universität bern
institut für informatik
und angewandte mathematik

Probabilistic Equivalence Checking with Propositional DAGs

Michael Wachter & Rolf Haenni

iam-06-001

April 2006

Address:
Institut für Informatik und angewandte Mathematik
Universität Bern
Neubrückstrasse 10
CH-3012 Bern

e-mail: office@iam.unibe.ch
Phone: 0041 (0)31 631 86 81
Fax: 0041 (0)31 631 32 62

Probabilistic Equivalence Checking with Propositional DAGs*

Michael Wachter & Rolf Haenni
University of Berne
Institute of Computer Science and Applied Mathematics
CH-3012 Berne, Switzerland
{wachter,haenni}@iam.unibe.ch

Abstract

The canonical representation of Boolean functions offered by OBDDs (ordered binary decision diagrams) allows to decide the equivalence of two OBDDs in polynomial time with respect to their size. It is still unknown, if this holds for other more succinct supersets of OBDDs such as FBDDs (free binary decision diagrams) and d-DNNFs (deterministic, decomposable negation normal forms), but it is known that the equivalence test for these is probabilistically decidable in polynomial time. In this paper, we show that the same probabilistic equivalence test is also possible for cd-PDAGs (decomposable, deterministic propositional DAGs). cd-PDAGs are more succinct than d-DNNFs and therefore strictly more succinct than FBDDs and OBDDs.

CR Categories and Subject Descriptors:
B.6.3. [Logic Design]: Design Aids—verification

General Terms:
Algorithms, Design, Theory, Verification

*Research supported by the Swiss National Science Foundation, Project No. PP002102652/1.

1 Introduction

Symbolic model checking is one of the most common formal verification techniques used in hardware design today. One of its primary goals is to formally prove that two distinct representations of a Boolean function are equivalent. This problem arises in various situations. As an example, consider the problem of checking if a manually modified circuit design functionally corresponds to the original one. Another typical example is the question of whether a gate-level implementation meets its functional specification.

Equivalence checking is known to be a hard problem in general. Most techniques today make use of some sort of canonical representation, for which the equivalence check is performed in polynomial time with respect to the representation size. The problem is that the representation size of some ill-suited problems is already exponential in the number of variables. The best strategy thus is to choose the most succinct representation that allows a polynomial equivalence check [6].

The most prominent way of canonically representing Boolean functions is by means of *ordered binary decision diagrams* (OBDD) and their derivatives [2]. The idea thus is to transform functional specifications or circuit designs into corresponding OBDDs, and to see if they are structurally identical. In practice, this works well for a broad class of problems, and that's the key to the huge success of OBDDs in the practical applications of formal verification. Unfortunately, certain important problems such as integer multiplication lead to intractably large OBDDs. In his original paper on OBDDs, Bryant commented this limitation with the following statement [2]:

> "*Given the wide variety of techniques used in implementing multipliers* [...]*, a canonical form for Boolean functions* [...] *that could efficiently represent multiplication would be of great interest for circuit verification. Unfortunately, these functions seem especially intractable.*"

To overcome these difficulties, some authors proposed more succinct *languages* such as FBDDs (free binary decision diagrams) or d-DNNFs (deterministic, decomposable negation normal forms). "More succinct" means that every Boolean functions with a tractable representation in the second language has also a tractable representation in the first language. The price these languages have to pay is the abandonment of the canonical form. If this implies that equivalence can no longer be checked in polynomial time is still an open question for both FBDDs and d-DNNFs [6, 4]. Moreover, it has been shown that any FBDD representation of an integer multiplier grows exponentially with the number of bits [9], something that has not yet been been discussed in the literature for d-DNNFs.

An alternative to the missing *exact* equivalence check is the *probabilistic* test proposed in [1, 8] for FBDDs and in [5] for d-DNNFs. This test is analogue to the well-known and widely applied class of probabilistic prime number tests of a candidate prime number [11, 10]. If the result of the test is negative, the two representations are certainly not equivalent (resp. the candiate is not prime). In the case of a positive test result, the two representations appear to be equivalent (resp. the candidate appears to be prime), but this conclusion is subject to a certain failure probability $\pi \in [0, 1]$. Repeating the test under different conditions, let's say t times, leads to a reduced total failure probability of π^t. For a quick convergence towards zero, π is usually supposed to be smaller than $\frac{1}{2}$. An adequate choice of t allows then to make the equivalence check arbitrarily accurate.

This paper shows how to adopt the above-mentioned probabilistic test for cd-PDAGs (decomposable, deterministic propositional DAGs). This is an appealing new technique for the

representation of Boolean functions [13]. The corresponding language includes all possible d-DNNFs, FBDDs, and OBDDs, and is therefore the most succinct language among all of them.

2 Boolean Functions and Propositional DAGs

In this section, we will review the graph-based language for representing Boolean functions called *propositional DAGs* [13]. We will use PDAG to refer to the language of propositional DAGs and PDAG to refer to an element of PDAG. Consider a set V of r propositional variables and a Boolean function (BF) $f : \{0,1\}^r \to \{0,1\}$. Such a function f can also be viewed as the set of r-dimensional vectors $\mathbf{x} \in \{0,1\}^r$ for which f evaluates to 1. This is the so-called *satisfying set* or *set of models* $S_f = \{\mathbf{x} \in \{0,1\}^r : f(\mathbf{x}) = 1\}$ of f, for which an efficient representation has to be found [3].

A PDAG is a rooted, directed acyclic graph in which each leaf node is represented by $\bigcirc$ and labeled with $\top$ (true), $\bot$ (false), or $x \in V$. Each non-leaf node is represented by $\triangle$ (logical and), $\triangledown$ (logical or), or $\diamond$ (logical not). Fig. 1 depicts two examples.

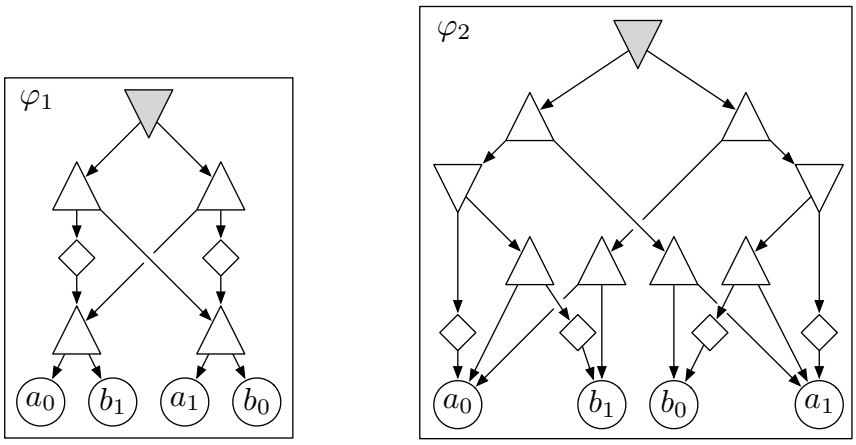

Figure 1: Two distinct PDAGs representing the same Boolean function.

Leaves labeled with $\top$ ($\bot$) represent the constant BF which evaluates to 1 (0) for all $\mathbf{x} \in \{0,1\}^r$. A leaf labeled with $x \in V$ is interpreted as the assignment $x = 1$, i.e. it represents the BF which evaluates to 1 iff $x = 1$. The BF represented by a $\triangle$-node is the one that evaluates to 1, iff the BFs of all its children evaluate to 1. Similarly, a $\triangledown$-node represents the BF that evaluates to 1, iff the BF of at least one child evaluates to 1. Finally, a $\diamond$-node represents the complementary BF of its child, i.e. the one that evaluates to 1, iff the BF of its child evaluates to 0. The BF of an arbitrary $\varphi \in$ PDAG will be denoted by f_φ. Two PDAGs $\varphi, \psi \in$ PDAG are *equivalent*, iff $f_\varphi = f_\psi$. This is denoted by $\varphi \equiv \psi$. Note that the two examples of Fig. 1 are equivalent.

By replacing the non-terminal nodes of $\varphi \in$ PDAG by corresponding logical gates, one can look at it as a *digital circuit* implementing f_φ. We prefer to make a distinction between PDAGs and digital circuits to emphasize their respective purposes. Digital circuits are mainly used to *implement* BFs, whereas PDAGs are useful to *represent*, *manipulate*, and answer *queries* about BFs. Examples of such queries are satisfiability, validity, entailment, equivalence, model counting, model and counter-model enumeration, and so on [6].

Our convention is to denote PDAGs by lower-case Greek letters such as φ, ψ, or the like. The

set of variables included in a sub-PDAG α of φ is denoted by $vars(\alpha) \subseteq V$. The number of edges of a PDAG φ is called its *size* and denoted by $|\varphi|$. The *depth* of φ, denoted by $depth(\varphi)$, is the maximal number of edges from the root to some leave. PDAGs can have a number of properties [6, 13]. In the context of this paper, only two of them are relevant:

- *Decomposability*: this property holds, if the sets of variables of the children of each $\wedge$-node α in φ are pairwise disjoint (i.e. if $\beta_1, \ldots, \beta_l$ are the children of α, then $vars(\beta_i) \cap vars(\beta_j) = \emptyset$ for all $i \neq j$);

- *Determinism*:[1] this property holds, if the children of each $\vee$-node α in φ are pairwise logically contradictory (i.e. if $\beta_1, \ldots, \beta_l$ are the children of α, then $\beta_i \wedge \beta_j \equiv \bot$ for all $i \neq j$).

A decomposable and deterministic PDAG is called cd-PDAG. Note that both examples shown in Fig. 1 are cd-PDAGs. We will refer to the corresponding language by `cd-PDAG`, a sub-language of `PDAG`. Other sub-languages are obtained from considering further properties: `d-DNNF` is the sub-language of `cd-PDAG` satisfying *simple-negation*, `FBDD` is the sub-language of `d-DNNF` satisfying *read-once*, and `OBDD` is the sub-language of `FBDD` satisfying *ordering* [6, 13]. It is shown in Chapter 11.1 of [14] that it is sufficient for a language to offer a probabilistic equivalence check, if both decomposability and determinism hold.

A language $\mathtt{L}_1$ is *more succinct* than another language $\mathtt{L}_2$, $\mathtt{L}_1 \preceq \mathtt{L}_2$, if any sentence $\alpha_2 \in \mathtt{L}_2$ has an equivalent sentence $\alpha_1 \in \mathtt{L}_1$ whose size is polynomial in the size of α_2. A language $\mathtt{L}_1$ is *strictly* more succinct than another language $\mathtt{L}_2$, $\mathtt{L}_1 \prec \mathtt{L}_2$, iff $\mathtt{L}_1 \preceq \mathtt{L}_2$ and $\mathtt{L}_2 \not\preceq \mathtt{L}_1$. With respect to the above-mentioned languages, we have the following proven relationships:

$$\mathtt{PDAG} \prec \mathtt{cd\text{-}PDAG} \preceq \mathtt{d\text{-}DNNF} \prec \mathtt{FBDD} \prec \mathtt{OBDD}. \tag{1}$$

Whether `cd-PDAG` is strictly more succinct than `d-DNNF` remains an open question.

3 Probabilistic Equivalence Testing

Let f_φ and f_ψ be the BFs of two PDAGs φ and ψ, respectively. Recall that $\varphi \equiv \psi$ is defined by $f_\varphi = f_\psi$, which in turn can be expressed by $S_{f_\varphi} = S_{f_\psi}$ in terms of their satisfying sets. Of course, comparing their satisfying sets is by no means a tractable way of checking the equivalence of two PDAGs, but this is the starting point of this section, which will lead us to a general probabilistic equivalence test for BFs, for which we will propose two possible implementations. These general versions of the test are then adapted for the particular case of cd-PDAG representations.

The idea of probabilistically checking the equivalence of two BFs f_φ and f_ψ goes as follows. First, we assign to each BF an *attribute*, which is thus unambiguously determined by the corresponding satisfying set. If f_φ and f_ψ have different attributes, they are not equivalent, since this would imply different satisfying sets. If their attributes are the same, they may or may not be equivalent, since we might have assigned the same attribute to two distinct BFs. The key point is to define the attribute in a way that two BFs with the same attribute are likely to be equivalent.

This idea has been identified in [1] and further studied in [8, 5, 7]. For the purpose of this paper, we will focus on a concrete type of attribute. In accordance with [8], we will refer to it as the *hash code* of a BF, while other authors prefer to call it *signature*. Without loss of generality, we restrict our attention to the field $\mathbb{F}_p = \{0, \ldots, p-1\}$ of integers, where p is a prime

[1] Disjunctions satisfying this property are sometimes called *orthogonal partitions* [14].

number. This allows us to define hash codes in terms of arithmetic operations within $\mathbb{F}_p$, where all additions, subtractions, and multiplications are performed modulo p.

Definition 1. Let $V = \{v_1, \ldots, v_r\}$ be a set of r propositional variables and $\mathbf{h} = (h_1, \ldots, h_r) \in \mathbb{F}_p^r$ a r-dimensional vector of integers in $\mathbb{F}_p$.

- The hash code of a variable $v_i \in V$ is the corresponding coordinate of $\mathbf{h}$:

$$H_{\mathbf{h}}(v_i) \;:=\; h_i.$$

- The hash code of a truth assignment $\mathbf{x} = (x_1, \ldots, x_r) \in \{0,1\}^r$ is the following product:

$$H_{\mathbf{h}}(\mathbf{x}) \;:=\; \prod_{i=1}^{r} \begin{cases} H_{\mathbf{h}}(v_i), & \text{if } x_i = 1, \\ 1 - H_{\mathbf{h}}(v_i), & \text{if } x_i = 0. \end{cases}$$

- The hash code of a BF f is the sum of hash codes of its models:

$$H_{\mathbf{h}}(f) \;:=\; \sum_{\mathbf{x} \in S_f} H_{\mathbf{h}}(\mathbf{x}).$$

This definition implies that $H_{\mathbf{h}}(f) = 1$ if f is valid, $H_{\mathbf{h}}(f) = 0$ if f is inconsistent, and $H_{\mathbf{h}}(1 - f) = 1 - H_{\mathbf{h}}(f)$. We rephrase a theorem from [1] adjusted to our formulation.

Theorem 1. *If f_1 and f_2 are two distinct Boolean functions, then there are at least $(p-1)^r$ vectors $\mathbf{h} \in \mathbb{F}_p^r$ such that $H_{\mathbf{h}}(f_1) \neq H_{\mathbf{h}}(f_2)$.*

This theorem holds for arbitrary fields of size p, as proven in [1, 5]. For two non- equivalent BFs f_1 and f_2, we call $\mathbf{h} \in \mathbb{F}_p^r$ with $H_{\mathbf{h}}(f_1) \neq H_{\mathbf{h}}(f_2)$ a *witness* to the distinctness of f_1 and f_2. Otherwise, $\mathbf{h}$ is called a *liar*, as it is common in the literature on probabilistic algorithms. As a consequence of Theorem 1, we have at least $(p-1)^r$ witnesses and at most $p^r - (p-1)^r$ liars. Therefore, two BFs with identical hash codes appear to be equivalent with a failure probability

$$\pi \leq \frac{p^r - (p-1)^r}{p^r} = 1 - \frac{(p-1)^r}{p^r} \leq \frac{r}{p}, \text{ for } p \geq r.$$

To make π smaller than $\frac{1}{2}$, it is necessary to choose $p \geq 2r$. This is the usual minimal requirement, which guarantees that the total failure probability of a repeated test quickly converges towards 0. Another strategy is to choose p sufficiently large from the beginning, which guarantees a low failure probability after a single test. These are the underlying ideas of the two algorithms of the following subsection.

3.1 General Algorithms

The proposed definition of hash codes makes them dependent of the satisfying set S_f of a BF f. As the size of the satisfying set may be exponential in the number of variables, this is computationally not tractable. However, it has been shown that the same hash codes can be computed more efficiently using appropriate OBDD, FBDD, or d-DNNF representations of f [1, 5]. In the following algorithms, we will thus consider representations of BFs as input parameters, not the BFs themselves. Note that this is our final goal, namely to check the equivalence of two different representations.

Consider the two algorithms `ProbEquivIter` and `ProbEquiv` shown below. Their input parameters are representations φ and ψ of two BFs f_φ and f_ψ, respectively, and the maximal tolerable failure probability $\varepsilon \in [0,1]$. Both algorithms implement the probabilistic equivalence test, but they follow different strategies.

The iterative version `ProbEquivIter` uses the smallest possible prime number which guarantees $\pi \leq \frac{1}{2}$. The test is then repeated until the total failure probability drops below ε. The maximal number of necessary runs for this is $t = -\lfloor \log_2(\varepsilon) \rfloor$.

```
Algorithm ProbEquivIter(φ, ψ, ε);
begin
    r ← |vars(φ) ∪ vars(ψ)|;
    p ← smallest prime number ≥ 2r;
    t ← −⌊log2(ε)⌋;
    while t ≥ 0 do
        h ← randomly selected from {0, ..., p − 1}^r;
        if H_h(f_φ) = H_h(f_ψ) then
            t ← t − 1;
        else
            return "not equivalent";
        end
    end
    return "probably equivalent";
end
```

Algorithm 1: The iterative version of the probabilistic equivalence test.

The non-iterative version `ProbEquiv` uses another strategy. It takes the smallest prime number $p \geq \frac{r}{\varepsilon}$ and then performs the equivalence test once. Since $\pi \leq \frac{r}{p}$ for $p \geq r$, we get $\pi \leq \varepsilon$ for $p \geq \frac{r}{\varepsilon}$ as required. In other words, the failure probability already drops below the tolerated rate ε after a single run.

```
Algorithm ProbEquiv(φ, ψ, ε);
begin
    r ← |vars(φ) ∪ vars(ψ)|;
    p ← smallest prime number ≥ r/ε;
    h ← randomly selected from {0, ..., p − 1}^r;
    if H_h(f_φ) = H_h(f_ψ) then
        return "probably equivalent";
    else
        return "not equivalent";
    end
end
```

Algorithm 2: The non-iterative version of the probabilistic equivalence test.

We postpone the discussion of the complexity of these algorithms until we have shown how to perform the probabilistic equivalence test for cd-PDAGs.

3.2 Adaption to cd-PDAGs

To adjust the general algorithms from the previous subsection to cd-PDAGs, we have to give an alternative definition of hash codes for elements of the language `cd-pdag`. The goal of course is to make this new definition compatible with the original one. The key properties for this are determinism and decomposability, the ones that are characteristic for the language `cd-pdag`. We will prove that we get to the same results, no matter if we compute the hash code with respect to a cd-PDAG φ or its BF f_φ. As before, we restrict our attention to the field $\mathbb{F}_p$ defined by a prime number p.

Definition 2. Let $V = \{v_1, \ldots, v_r\}$ be a set of r propositional variables and $\mathbf{h} = (h_1, \ldots, h_r) \in \mathbb{F}_p^r$ a r-dimensional vector of integers in $\mathbb{F}_p$. The hash code $H_\mathbf{h}(\varphi)$ of φ is recursively defined by:

- $H_\mathbf{h}(\varphi) := \prod_i H_\mathbf{h}(\beta_i)$, if φ is a $\wedge$-node with children β_i;
- $H_\mathbf{h}(\varphi) := \sum_i H_\mathbf{h}(\beta_i)$, if φ is a $\vee$-node with children β_i;
- $H_\mathbf{h}(\varphi) := 1 - H_\mathbf{h}(\beta)$, if φ is a $\diamond$-node with child β;
- If φ is a $\bigcirc$-node, then the hash code depends on its label:

$$H_\mathbf{h}(\varphi) := \begin{cases} 1, & \text{if } \varphi \text{ is labeled with } \top, \\ 0, & \text{if } \varphi \text{ is labeled with } \bot, \\ h_i, & \text{if } \varphi \text{ is labeled with } v_i \in V. \end{cases}$$

The following theorem associates both ways of computing hash codes. This allows us to replace $H_\mathbf{h}(f_\varphi) = H_\mathbf{h}(f_\psi)$ by $H_\mathbf{h}(\varphi) = H_\mathbf{h}(\psi)$ in both versions of the algorithm.

Theorem 2. *For all $\varphi \in$ `cd-PDAG`, we have $H_\mathbf{h}(f_\varphi) = H_\mathbf{h}(\varphi)$.*

Proof. We will use the principle of mathematical induction to prove this theorem. For $depth(\varphi) = 0$, φ is a $\bigcirc$-node. If φ is labeled with $v_i \in V$, we have $H_\mathbf{h}(\varphi) = h_i = H_\mathbf{h}(f_\varphi)$. If φ is labeled with $\top$ (f_φ is valid) or with $\bot$ (f_φ is inconsistent), the definition exploits the corresponding facts about valid or inconsistent BFs, respectively.

For $depth(\varphi) = j$ we have to consider three kinds of nodes. First, if φ is a $\diamond$-node φ with child α, we have $f_\varphi = 1 - f_\alpha$ and $H_\mathbf{h}(\alpha) = H_\mathbf{h}(f_\alpha)$, since $depth(\alpha) < j$. So $H_\mathbf{h}(\varphi) = 1 - H_\mathbf{h}(\alpha)$ exploits the the fact $H_\mathbf{h}(f_\varphi) = H_\mathbf{h}(1 - f_\alpha) = 1 - H_\mathbf{h}(f_\alpha)$, from which we conclude $H_\mathbf{h}(f_\varphi) = H_\mathbf{h}(\varphi)$.

Without loss of generality, we assume for the two other node types that the children of φ are α_1 and α_2. Since $depth(\alpha_i) < j$, we also have $H_\mathbf{h}(\alpha_i) = H_\mathbf{h}(f_{\alpha_i})$.

If φ is a $\vee$-node, since it is deterministic, we know that α_1 and α_2 have no common element in the corresponding satisfying sets. This leads to $H_\mathbf{h}(\varphi) = H_\mathbf{h}(\alpha_1) + H_\mathbf{h}(\alpha_2) = H_\mathbf{h}(f_{\alpha_1}) + H_\mathbf{h}(f_{\alpha_2}) = H_\mathbf{h}(f_\varphi)$.

If φ is a $\wedge$-node, since itis decomposable, we know that each variable in $vars(\varphi)$ occurs either in α_1 or in α_2. This implies $S_\varphi = S_{\alpha_1}|_{V_1} \times S_{\alpha_2}|_{V_2}$, where $V_i = vars(\alpha_i)$. This leads to $S_{f_\varphi} = S_{f_{\alpha_1}}|_{V_1} \times S_{f_{\alpha_2}}|_{V_2}$. With $H_\mathbf{h}(f_{\alpha_i}) = H_\mathbf{h}(f_{\alpha_i}|_{V_i})$, we get $H_\mathbf{h}(\varphi) = H_\mathbf{h}(\alpha_1) \cdot H_\mathbf{h}(\alpha_2) = H_\mathbf{h}(f_{\alpha_1}) \cdot H_\mathbf{h}(f_{\alpha_2}) = H_\mathbf{h}(f_\varphi)$.

□ □

The hash code computation of a cd-PDAG is illustrated in Fig. 2. With respect to $V = \{a, b, c\}$, φ represents the BF f_φ with the satisfying set $S_{f_\varphi} = \{(1,1,1), (1,1,0), (0,1,1)\}$. If we assume

that $\mathbf{h} = (s_a, s_b, s_b)$ is the randomly selected vector of integers in the field $\mathbb{F}_p$, we get

$$\begin{aligned} H_{\mathbf{h}}(f_\varphi) &= s_a s_b s_c + s_a s_b (1 - s_b) + (1 - s_a) s_b s_c \\ &= s_b (s_a + s_c - s_a s_c) \\ &= H_{\mathbf{h}}(\varphi) \end{aligned}$$

for both the hash code of f_φ and the hash code of φ. The computation in the cd-PDAG is illustrated in Fig. 2.

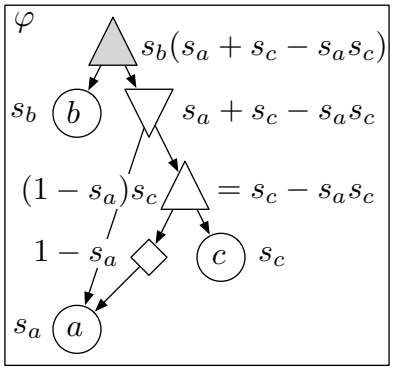

Figure 2: Computing the hash code of a cd-PDAG.

3.3 Complexity Analysis

To conclude this section, let's say a few words about the complexities of these algorithms. According to the observations from [1], it is essential to first look at the complexity of single additions and multiplications. Within the field $\mathbb{F}_p$, the hash codes do not exceed $\log p$ bits. According to [12], this restricts the complexity of addition to $\mathcal{O}(\log p)$ and the classical[2] multiplication to $\mathcal{O}(\log^2 p)$.

Performing all the arithmetic operations modulo p is very important for the complexity. By using the finite field only to define the hash codes of the variables, but not to perform the arithmetic operations, we would unnecessarily blow up the complexity to $\mathcal{O}(r \log p)$ for additions and to $\mathcal{O}(r^2 \log^2 p)$ for multiplications. This is the problem of the procedure described in [5], which does not improve the theoretical failure probability π.

Let $\varphi_1, \varphi_2 \in$ `cd-PDAG` be two arbitrary cd-PDAGs, and let S be the sum of their sizes, $S = |\varphi_1| + |\varphi_2|$. Since the complexity depends on the number of additions and multiplications, and since multiplication is more complex than addition, we assume the worst case in which all the necessary arithmetic operations are multiplications. Furthermore, we neglect the computational costs for finding appropriate prime numbers and randomly selecting a vector $\mathbf{h}$.

In the case of the iterative version `ProbEquivIter`, we choose p to be the smallest prime number $\geq 2r$. This implies $\log p \geq \log 2r = \log 2 + \log r = 1 + \log r$, from which we get $\mathcal{O}(\log^2 r)$ for the complexity of multiplication. With this, we get $\mathcal{O}(S \cdot \log^2 r \cdot (-\log \varepsilon))$ for the time complexity of `ProbEquivIter`.

[2]Karatsuba's and Schönhage & Strassen's multiplication algorithms reduce the complexity to $\mathcal{O}(\log^{1.59} p)$ and $\mathcal{O}(\log p (\log\log p)(\log\log\log p))$, respectively.

In the non-iterative version `ProbEquiv`, we choose p to be the smallest prime number $\geq \frac{r}{\varepsilon}$. This implies $\log p \geq \log \frac{r}{\varepsilon} = \log r - \log \varepsilon$, and the complexity of multiplication is $\mathcal{O}((\log r - \log \varepsilon)^2)$. Therefore, we obtain $\mathcal{O}(S \cdot (\log r - \log \varepsilon)^2)$ for the time complexity of `ProbEquiv`.

The question arises which algorithm should be preferred. To analyze this, let's move to the perspective of $x = \log r$ and $y = -\log \varepsilon$. Based on the above-mentioned complexity results, it follows then that `ProbEquivIter` is preferred for $x^2(1-y) + 2xy + y^2 > 0$ and `ProbEquiv` is preferred for $x^2(1-y) + 2xy + y^2 < 0$. We have no preference for $x^2(1-y) + 2xy + y^2 = 0$, i.e. it does not matter which algorithm we use.

Since the number of variables r is normally fix, we can not change x. So if we consider the above inequality as a function $g(y) = y^2 - (x^2 - 2x)y + x^2$, we get two solutions for $g(y) = 0$ and $x \geq 4$, namely $y_1 = \frac{x}{2}\left(\sqrt{x^2 - 4x + 4} - \sqrt{x^2 - 4x}\right)$ and $y_2 = \frac{x}{2}\left(\sqrt{x^2 - 4x + 4} + \sqrt{x^2 - 4x}\right)$. This is another interesting result: if $y_1 \leq y \leq y_2$, we prefer `ProbEquiv`, otherwise we prefer `ProbEquivIter`.

4 Conclusions

The language `cd-PDAG` already supports a number of useful queries in polynomial time. The most important ones are satisfiability, validity, clause entailment, term implication, model counting, and probability computation [13]. While sentence entailment is known to be infeasible in the language `cd-PDAG`, it is still unknown if there is a polynomial equivalence test or not.

The probabilistic equivalence tests proposed in this paper are an alternative to the missing exact equivalence test. We have shown that for an adequate choice of parameters, the failure probabilities of these tests converge quickly towards 0. As long as the existence of an exact test is still an open question, we propose to use these probabilistic tests instead. This seems to be an interesting alternative to the techniques used in hardware design, which are mostly based on the language `OBDD` and its derivatives. The advantage of using the language `cd-PDAG` instead of `OBDD` is its succinctness.

References

[1] M. Blum, A. K. Chandra, and M. N. Wegman. Equivalence of free boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters*, 10(2):80–82, 1980.

[2] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[3] P. Clote and E. Kranakis. *Boolean Functions and Computation Models*. Springer, 1998.

[4] A. Darwiche. A compiler for deterministic, decomposable negation normal form. In *AAAI'02, 18th National Conference on Artificial Intelligence*, pages 627–634. AAAI Press, 2002.

[5] A. Darwiche and J. Huang. Testing equivalence probabilistically. Technical Report D–123, Computer Science Department, UCLA, Los Angeles, USA, 2002.

[6] A. Darwiche and P. Marquis. A knowlege compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[7] E. Dubrova and H. Sack. Probabilistic verification of multiple-valued functions. In *ISMVL'00, 30th IEEE International Symposium on Multiple-Valued Logic*, pages 460–466, Portland, USA, 2000. IEEE Computer Society.

[8] J. Jain, J. Bitner, D. S. Fussell, and J. A. Abraham. Probabilistic design verification. In *ICCAD'91, International Conference on Computer-Aided Design*, pages 468–471, Santa Clara, USA, 1991.

[9] S. Ponzio. A lower bound for integer multiplication with read-once branching programs. In *STOC'95, 27th Annual ACM Symposium on Theory of Computing*, pages 130–139, Las Vegas, USA, 1995. ACM Press.

[10] M. O. Rabin. Probabilistic algorithm for primality testing. *Journal of Number Theory*, 12:128–138, 1980.

[11] R. Solovay and V. Strassen. A fast monte-carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, 1977.

[12] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, USA, 2003.

[13] M. Wachter and R. Haenni. Propositional DAGs: a new graph-based language for representing Boolean functions. In *KR'06, 10th International Conference on Principles of Knowledge Representation and Reasoning (accepted)*, Lake District, U.K., 2006.

[14] I. Wegener. *Branching Programs and Binary Decision Diagrams*. Society for Industrial and Applied Mathematics, 2000.