The Boolean Isomorphism Problem

Manindra Agrawal [*]
Dept. of Computer Science
Indian Institute of Technology
Kanpur 208016, India

Thomas Thierauf [†]
Abt. Theoretische Informatik
Universität Ulm
89069 Ulm, Germany

March 12, 1996

Abstract

We investigate the computational complexity of the *Boolean Isomorphism problem (BI)*: on input of two Boolean formulas F and G decide whether there exists a permutation of the variables of G such that F and G become equivalent.

Our main result is a one-round interactive proof for $\overline{\mathrm{BI}}$, where the verifier has access to an NP oracle. To obtain this, we use a recent result from learning theory by Bshouty et.al. that Boolean formulas can be learned probabilistically with equivalence queries and access to an NP oracle. As a consequence, BI cannot be Σ_2^p complete unless the Polynomial Hierarchy collapses. This solves an open problem posed in [BRS95].

Further properties of BI are shown: BI has And- and Or-functions, the counting version, #BI, can be computed in polynomial time relative to BI, and BI is self-reducible.

1 Introduction

An interesting computational issue is to decide the equivalence of two given programs with respect to some computational model. While the problem is undecidable for computational models such as Turing machines, LOOP-programs, or context-free languages (see [HU79]), it is coNP complete for LOOP(1)-programs (no nested loops), circuits, branching programs, and Boolean formulas. Moreover, it can be efficiently solved for one-time-only branching programs by a randomized algorithm [BCW80]. For regular languages it can be efficiently solved deterministically (see [HU79]).

In this paper, we consider the complexity of a generalized version of the equivalence problem: decide whether two given programs become equivalent via some bijective transformation of the input. One of the simplest such transformation is an *isomorphism*, i.e., a permutation of the input bits. A slightly more general transformation is the *congruence*, where the permutation of the variables is composed with a *negation mapping* which maps each variable either to itself or to its complement. The congruence problem for Boolean functions is in fact a very old problem that has already been investigated in the last century. To motivate the name, the Boolean congruence relation can be seen as a geometrical congruence. There are 2^n assignments for n variables, forming the nodes of a n-dimensional cube in $\mathbf{R}^n$. The assignments where a Boolean formula F evaluates to one form a subgraph of the cube, *the n-dimensional geometrical cube*

[*]Research done while visiting the university of Ulm, Germany. Supported in part by an Alexander von Humboldt fellowship.

[†]Supported in part by DAAD, Acciones Integradas 1995, 322-AI-e-dr.

representing F. Two formulas F and G are congruent if and only if the n-dimensional geometrical cubes representing F and G are geometrically congruent, that is, there is a distance-preserving bijection from one subgraph to the other.

The paper by Borchert et.al. [BRS95] gives extensive background and provides a list of early references on this problem. In recent years, these problems have been reconsidered with respect to their computational complexity [BR93, BRS95, CK91] for different repesentations like formulas or circuits.

We call the isomorphism and congruence problem for Boolean formulas the *Boolean Isomorphism (BI)* and *Boolean Congruence (BC)* problem, respectively. Although congruence is a broader notion than isomorphism, BC is many-one equivalent to BI [BRS95].

BI is coNP hard but not known to be in coNP. Therefore it is at least as difficult as the Boolean equivalence problem. As an upper bound on its complexity, BI is in the second level of the Polynomial Hierarchy, Σ_2^p. It is posed as an open problem by Borchert et.al. [BRS95] whether BI is complete for Σ_2^p. They conjectured that it is not. In this paper we will solve this question in the affirmative by showing that BI is *not* complete for Σ_2^p unless the Polynomial Hierarchy collapses. We also give a lower bound for BI: we show that the problem of deciding if a graph has a unique optimal clique—which is not known to be in the Boolean Hierarchy—many-one reduces to it.

The Boolean Isomorphism problem shares many similarities with the Graph Isomorphism problem, GI (see [Hof82] and [KST93] for a comprehensive study on Graph Isomorphism). Many of the results for GI carry over to BI with similar proofs, although with some crucial differences.

We can rewrite any permutation of n variables $\mathbf{x} = (x_1, \ldots, x_n)$ as a product of a permutation matrix $\mathbf{P}$ with $\mathbf{x}$ over GF(2). That is, an isomorphism can be written as $\mathbf{xP}$, and a congruence can be written as $\mathbf{xP} + \mathbf{c}$, for a vector $\mathbf{c} \in \{0,1\}^n$. A natural generalization of the above notions is therefore to consider linear and affine transformations $\mathbf{xA}$ and $\mathbf{xA} + \mathbf{c}$, respectively, where $\mathbf{A}$ has to be a bijection on $\{0,1\}^n$. We call two formulas *linear equivalent* or *affine equivalent* if they become equivalent after a linear or an affine transformation of the variables of one of the formulas, respectively. As in the case of isomorphism and congruence, the *Boolean Linear Equivalence* problem, BLE, and the *Boolean Affine Equivalence* problem, BAE, are many-one equivalent [BRS95]. Also, BI many-one reduces to BAE [BRS95].

Looking at circuits instead of Boolean formulas, we get the corresponding problems CI, CC, CLE, and CAE. CAE is the most complex problem we considered so far: all the other problems are many-one reducible to it. The above mentioned result for BI holds in fact more general for CAE. That is, CAE is not complete for Σ_2^p unless the Polynomial Hierarchy collapses.

The paper is organized as follows. In Section 3 we show that the complement of BI has an one-round interactive proof, where the verifier has access to an NP oracle. The interactive proof can be extended to the complement of CAE. From this we conclude the above mentioned non-completeness results. In Section 4 we show that BI has And- and Or-functions. This will provide us with a lower bound for BI: the Unique Optimal CLIQUE problem can be many-one reduced to it. In Section 5 we show that the counting version of BI can be solved in polynomial time relative to BI. This is an result that holds analogously for the Graph Isomorphism problem. Finally, in Section 6 we show that BI is self-reducible.

2 Preliminaries

An *n-ary Boolean function* $f = f(x_1, \ldots, x_n)$ is a mapping from $\{0,1\}^n$ to $\{0,1\}$. An *assignment* for (the variables of) f is a mapping $a : \{x_1, \ldots, x_n\} \mapsto \{0,1\}^n$.

Every Boolean function f can be expressed as a *Boolean formula* F over variables $\{x_1, \ldots, x_n\}$ using the 2-ary conjunction ($\wedge$) and disjunction ($\vee$) and the 1-ary negation ($\neg$) as basis. We will additionally use implication ($\rightarrow$) and equivalence($\leftrightarrow$). The semantic of a formula is defined as usual. By convention, we use small letters for functions and capital letters for formulas. Note that there can be several formulas representing the same function.

Two formulas F and G are *equivalent* if $f = g$. That is, the associated functions are identical. Since F and G are equivalent if and only if the formula $F \leftrightarrow G$ is a tautology, the problem of deciding whether two formulas are equivalent is coNP complete.

Two formulas F and G are *isomorphic*, denoted by $F \cong G$, if there exists a permutation φ on $\{x_1, \ldots, x_n\}$, such that $f = g \circ \varphi$. In this case, we call φ an *isomorphism between* F *and* G. The *Boolean Isomorphism problem* is $\mathrm{BI} = \{ \langle F, G \rangle \mid F \cong G \}$. It follows directly from the definition that $\mathrm{BI} \in \Sigma_2^p$, the second level of the Polynomial Hierarchy.

$\mathrm{Iso}(F, G)$ denotes the set of isomorphism between F and G. An automorphism of a formula F is an isomorphism between F and F, $\mathrm{Aut}(F) = \mathrm{Iso}(F, F)$. $\mathrm{Aut}(F)$ is a group with composition $\circ$ as group operation. It is a subgroup of the permutation group (on n variables). The *Boolean Automorphism problem*, BA, is the set of formulas F that have a non-trivial automorphism, i.e., $|\mathrm{Aut}(F)| > 1$.

A *negation mapping* on n variables is a function ν such that $\nu(x_i) \in \{x_i, \overline{x}_i\}$ for $1 \leq i \leq n$. Two formulas F and G are *congruent*, if there exists a permutation φ and a negation mapping ν on $\{x_1, \ldots, x_n\}$, such that $f = g \circ \nu \circ \varphi$. The *Boolean Congruence problem*, BC, is the set of pairs of formulas that are congruent. Congruence is a more flexible notion than isomorphism, however, BC is clearly in Σ_2^p, moreover $\mathrm{BI} \equiv_m^p \mathrm{BC}$ [BRS95].

Formulas F and G are *affine equivalent* if there exists a non-singular $n \times n$ matrix $\mathbf{A}$ and a $1 \times n$ vector $\mathbf{c}$ over GF(2) such that for every $\mathbf{x} = (x_1, \ldots, x_n)$, we have $f(\mathbf{x}) = g(\mathbf{xA} + \mathbf{c})$ (here addition and multiplication is over GF(2)). The formulas are *linear equivalent* if they are affine equivalent with the vector c being zero. We use BAE and BLE to denote the set of pairs of formula that are respectively affine and linear equivalent.

The above definitions can be applied to circuits instead of formulas. We use CI, CC, CLE, and CAE to denote the set of circuit pairs that are isomorphic, congruent, linear equivalent, and affine equivalent, respectively. The reductions shown in [BRS95] can easily be seen to carry over to circuits. That is, we have $\mathrm{CI} \equiv_m^p \mathrm{CC} \leq_m^p \mathrm{CLE} \equiv_m^p \mathrm{CAE}$. Since a formula can easily be transformed to a circuit, each Boolean formula problem is many-one reducible to its corresponding circuit version. It follows that CLE and CAE are the computationally hardest problems we have defined here.

We will use (fairly standard) notions of complexity theory. We refer the reader to [BDG-I&II, HU79] for definitions of these. Here, we only give an informal description of a few notions.

Interactive proofs were defined in [GMR89]. Informally, a language has an interactive proof if there is a probabilistic polynomial-time verifier that accepts the strings in the language with the help of an all powerful prover with high probability, and rejects the strings not in the language with high probability irrespective of the prover. If the verifier needs at most k rounds of message exchanges with the prover to accept the language, we say that the language belongs to the class $\mathrm{IP}[k]$ (one round consists of a question by verifier and its answer by prover). *Arthur-Merlin games* were introduced in [Bab85]. These are similar to interactive proofs with Arthur being the verifier and Merlin, the prover except that here the verifier is obliged to make all its random bits also available to the prover. The class $\mathrm{AM}[k]$ is similarly defined.

A language L is in the class $\mathrm{BP} \cdot \mathcal{C}$ if there exists a language $A \in \mathcal{C}$, and a polynomial p such that for every $x \in L$, $(x, y) \in A$ for at least $2/3$ of the y's of length $p(|x|)$, and for every $x \notin L$, $(x, y) \notin A$ for at least $2/3$ of the y's of length $p(|x|)$. Similarly, one defines $\mathrm{NP} \cdot \mathcal{C}$.

3 An Interactive Proof for $\overline{\mathrm{BI}}$

We show that there is a one round interactive proof for the complement of the Boolean Isomorphism problem, $\overline{\mathrm{BI}}$, where the verifier has access to an NP oracle. It follows that $\overline{\mathrm{BI}}$ is in $\mathrm{BP}\cdot\Sigma_2^p$ and is therefore not Π_2^p complete unless the Polynomial Hierarchy collapses.

Our interactive proof is based on the one for $\overline{\mathrm{GI}}$ [GMR89] (see also [Sch88]), however, it differs from it in one crucial aspect. Let us first recall the protocol for $\overline{\mathrm{GI}}$:

> On input (G_1, G_2), the verifier randomly picks $i \in \{1,2\}$, applies a random permutation of the vertices of G_i to obtain a new graph H and sends H to the prover. The prover answers by sending $j \in \{1,2\}$ to the verifier. Finally, the verifier accepts if and only if $i = j$.

When the input graphs are *not* isomorphic, the prover can find out from which of G_1 or G_2 the graph H was obtained by the verifier, and can therefore make the verifier accept with probability one. On the other hand, when the graphs are isomorphic, then no prover can find out the graph that was chosen by the verifier to construct H. Therefore, the answer of any prover is correct with the probability at most $1/2$.

Unfortunately the analog protocol for $\overline{\mathrm{BI}}$ does not work. To see this, consider the above protocol on input (F_1, F_2), where

$$\begin{aligned} F_1 &= x_1 \wedge (\overline{x}_1 \vee \overline{x}_2), \quad \text{and} \\ F_2 &= \overline{x}_1 \wedge x_2. \end{aligned}$$

Note that F_1 and F_2 are isomorphic (exchange x_1 and x_2). The verifier randomly picks $i \in \{1,2\}$, obtains a formula G by randomly permuting F_i and sends it to the prover. However, even though F_1 and F_2 are isomorphic, the prover can easily detect from which one G has been obtained, because of the syntactic structure of G: any permutation of F_1 will have three literals and any permutation of F_2 will have two literals.

It seems as what we need is a normal form for equivalent Boolean formulas that can be computed by the verifier. (Recall that the verifier has access to an NP oracle.) For a formula F, let $[F]$ denote the set of all Boolean formulas that are equivalent to F. If we have a way to map every formula in $[F]$ to a particular formula in $[F]$, then the protocol works: the verifier can map the formula G in the above protocol to its normal form G', and then the prover cannot distinguish whether G' is coming from F_1 or F_2 if the formulas are isomorphic.

Clearly, we cannot simply go for DNF or CNF, because this might lead to exponentially longer formulas. Another obvious candidate for a normal form is the *smallest* equivalent Boolean formula (under some suitable total ordering). However, computing this seems to require a Σ_2^p oracle, and our verifier only has an NP oracle available.

To overcome this difficulty, we use a result from learning theory by Bshouty et.al. [BCGKT95].

Lemma 3.1 [BCGKT95] *There is a probabilistic polynomial-time algorithm having access to an NP oracle that learns a Boolean formula using equivalence queries.* [1]

The scenario is roughly as follows. There is a Boolean formula F given in a *black box*. A probabilistic polynomial-time machine, the *learner*, which cannot see F, has to compute with high probability a formula that is equivalent to F. The learner can use an NP oracle, and

[1]The result is stated there only for DNF formulas, but their proof works for general Boolean formulas too.

furthermore ask *equivalence queries* to a *teacher* who knows F. That is, the learner can send formulas G to the teacher. If F and G are equivalent, the learner has succeeded in learning F and the teacher will answer 'yes'. Otherwise, the teacher will send a counter example to the learner, that is, an x such that $f(x) \neq g(x)$.

The most important thing to note is that the output of the learner does not depend on the specific input formula F: because of the black box approach the learner makes the same outputs on every $F' \in [F]$ as input. Note also that this does not provide us with a normal form because the learner produces possibly several equivalent formulas depending on the random choices. However, on each random path the output remains the same on any $F' \in [F]$. This will suffice for our purposes.

Furthermore, the teacher can in fact be replaced by an NP oracle: an equivalence query can be simulated deterministically by the learner, since testing equivalence of two formulas is a coNP problem and computing counter examples can easily be done with several queries to NP.

Therefore, the above lemma gives a functional $\mathrm{ZPP}^{\mathrm{NP}}$-type algorithm to learn Boolean formulas. More precisely, we have the following.

Lemma 3.2 [BCGKT95] (restated) *There is a probabilistic polynomial-time algorithm that has access to an NP oracle such that on input of a Boolean formula F, the algorithm*

- *outputs a Boolean formula that is equivalent to F with probability at least* $2/3$*,*
- *never outputs a Boolean formula that is not equivalent to F, and*
- *uses the input F only to test its equivalence to some formula or to find counter examples via the NP oracle.*

Now the idea should be clear. The verifier, instead of directly sending the randomly produced formula G to the prover, first learns G via the above algorithm and then sends the formula it has learned. We give the full protocol below.

Theorem 3.3 $\overline{\mathrm{BI}} \in \mathrm{IP}[1]^{\mathrm{NP}}$.

Proof. The following IP-protocol accepts $\overline{\mathrm{BI}}$.

Input (F_1, F_2) with both the formulas being over variables $x_1, \ldots, x_n$.

Question Is F_1 not isomorphic to F_2?

Protocol The verifier randomly picks $i \in \{1, 2\}$ and a random permutation φ on n variables. Let $G = F_i \circ \varphi$. Now, the verifier uses the algorithm of Lemma 3.2 on input G to obtain an equivalent Boolean formula G' and sends G' to the prover. On those paths where the algorithm does not make an output, the verifier directly accepts.

The prover answers by sending $j \in \{1, 2\}$ to the verifier. Finally, the verifier accepts if $i = j$, and rejects otherwise.

We show that the above protocol works correctly. If F_1 is *not* isomorphic to F_2, a prover can determine which of F_1 and F_2 formula G' is isomorphic to, and tell it to the verifier. Also, on the random paths where no equivalent formula is produced the verifier accepts. Therefore, the verifier accepts with probability one.

Now consider the case when F_1 is isomorphic to F_2. Then formula G is isomorphic to both, F_1 and F_2. Since the algorithm of Lemma 3.2 has the same (set of) outputs on any formula in $[G]$, any G' that is sent to the prover has the same probability irrespective of whether G was obtained from F_1 or F_2. Hence, the prover has no way of finding out which of F_1 and F_2 were used to construct G'. Thus, the best way for the prover is to answer randomly and so its answer is correct with probability at most $1/2$. Therefore, the verifier will accept with probability at most $1/2 + 1/3 = 5/6$ (the $1/3$ comes from the paths on which the verifier accepts without asking the prover).

The verifier can execute the above protocol in parallel to obtain exponentially small bounds on the error. This proves the theorem. □

We remark that the same idea can be used to give a *perfect zero-knowledge* interactive proof for BI, where the verifier has access to an NP oracle.

It is known that the private coins of an IP protocol can be made public with only two more rounds [GS89]. Moreover, a constant round AM protocol can be reduced to a single round [Bab85]. Both results hold in the presence of an NP oracle as well. Therefore, we have

$$\overline{\mathrm{BI}} \in \mathrm{AM}^{\mathrm{NP}}.$$

Finally, it is known that $\mathrm{AM} = \mathrm{BP}\cdot\mathrm{NP}$, because in an AM protocol, *Arthur* can be replaced by a BPP machine that just passes the result of the coin tosses to *Merlin*. After receiving the answer, the final decision is a polynomial-time computation. Now, in $\mathrm{AM}^{\mathrm{NP}}$ the final decision is in P^{NP}. Therefore we have $\mathrm{AM}^{\mathrm{NP}} = \mathrm{BP}\cdot\mathrm{NP}\cdot\mathrm{P}^{\mathrm{NP}} = \mathrm{BP}\cdot\Sigma_2^p$.

Corollary 3.4 $\overline{\mathrm{BI}} \in \mathrm{BP}\cdot\Sigma_2^p$.

Schöning [Sch88] gives a direct proof that the graph isomorphism problem is in AM by using hash functions. We remark that we can extend Schöning's proof by our technique to directly obtain Corollary 3.4

Schöning [Sch89] showed that a Π_2^p complete set cannot be in $\mathrm{BP}\cdot\Sigma_2^p$ unless the Polynomial Hierarchy collapses.

Corollary 3.5 *If* BI *is* Σ_2^p*-complete then* $\mathrm{PH} = \Sigma_3^p$.

Bshouty et.al. [BCGKT95] show the analog result to Lemma 3.2 for circuits. Therefore we can adapt the interactive proof for $\overline{\mathrm{BI}}$ for the Circuit Isomorphism problem, CI which gives $\overline{\mathrm{CI}} \in \mathrm{IP}[1]^{\mathrm{NP}}$.

Corollary 3.6 *If* CI *is* Σ_2^p*-complete then* $\mathrm{PH} = \Sigma_3^p$.

We can extend the interactive proof for BI even to the linear and affine equivalence problems. The only difference is when the verifier randomly generates a permutation. Now, the verifier must randomly generate an affine transformation. To achieve this, the verifier randomly generates an n-bit vector and an $n \times n$ 0-1 matrix. If the matrix is singular, the verifier accepts. Otherwise, the protocol proceeds as in the case of a permutation.

Observe that we get an extra error because some random paths of the verifier might lead to singular matrices. However, the next lemma ensures that a constant fraction of all matrices are non-singular. Therefore, the verifier can repeat the experiment to find a non-singular matrix a few number of times so that the probability of not finding one is very small.

Lemma 3.7 *At least* $1/4$ *of the* $n \times n$ *matrices over* GF(2) *are non-singular.*

Proof. We successively choose the column vectors of a $n \times n$ matrix such that the next column vector is linearly independent of the previous ones. The first column can be chosen arbitrary, except that it can't be zero. So there are $2^n - 1$ choices.

Any k linearly independent vectors in $\mathrm{GF}(2)^n$ span a vector space of size 2^k. Therefore, when we choose the $(k+1)$-st column, we have $2^n - 2^k$ choices.

In total, $\prod_{k=0}^{n-1}(2^n - 2^k)$ of the 2^{n^2} $n \times n$ matrices over GF(2) are non-singular. Thus, their proportion is

$$\begin{aligned}
\frac{1}{2^{n^2}} \prod_{k=0}^{n-1}(2^n - 2^k) &= \prod_{k=1}^{n}(1 - \frac{1}{2^k}) \\
&= \frac{1}{2} \prod_{k=2}^{n}(1 - \frac{1}{2^k}) \quad (\text{for } n \geq 2) \\
&\geq \frac{1}{2} \prod_{k=2}^{n}(1 - \frac{1}{k^2}) \quad (\text{for } n \geq 6) \\
&= \frac{1}{2}\,(\frac{1}{2}\,\frac{n+1}{n}) \\
&\geq \frac{1}{4},
\end{aligned}$$

where the second line from bottom follows by induction on n. For values of n smaller than 6 also the above bound holds, as can be checked directly. □

Corollary 3.8 $\overline{\mathrm{CAE}} \in \mathrm{IP}[1]^{\mathrm{NP}}$.

Proof. Let C_1 and C_2 be the input circuits with variables $\mathbf{x} = (x_1, \ldots, x_n)$. We have already described the protocol of the interactive proof. It remains to show that the prover cannot detect which of the two input circuits was used to obtain the circuit he got, when C_1 and C_2 are affine equivalent.

Let $\mathbf{xA} + \mathbf{c}$ be the affine transformation so that $C_2(\mathbf{xA} + \mathbf{c})$ is equivalent to C_1. For a random affine transformation, say $\mathbf{xR} + \mathbf{r}$, applied to C_2, we get $C_2(\mathbf{xR} + \mathbf{r})$. Applied to C_1, we get $C_1(\mathbf{xR} + \mathbf{r})$ which is equivalent to $C_2(\mathbf{xAR} + \mathbf{cR} + \mathbf{r})$. Now note that $\mathbf{x} \mapsto \mathbf{xAR} + \mathbf{cR} + \mathbf{r}$ is still a random affine transformation for fixed $\mathbf{A}$ and $\mathbf{c}$. □

Corollary 3.9 *If* CAE *is* Σ_2^p*-complete then* $\mathrm{PH} = \Sigma_3^p$.

4 BI has AND and OR Functions

The complexity of sets can be compared by reductions. It two sets are equivalent with respect to some reduction, they are considered as having similar complexity, where this similarity increases the more restrictive the reduction is. As an example, in the Turing degree of SAT one can already solve NP optimization problems while this *might* not be possible in the many-one degree of SAT. On the other hand, a conjecture of Berman and Hartmanis is that the many-one degree of SAT collapses to its isomorphism degree.

In this section we show that for BI, the disjunctive and conjunctive truth-table degree collapses to the many-degree of BI. This is a consequence of BI having And- and Or-functions.

Definition 4.1 *An* And-function *for a set A is a function* $\mathsf{And} : \Sigma^* \times \Sigma^* \mapsto \Sigma^*$ *such that for any $x, y \in \Sigma^*$, we have $x \in A$* and *$y \in A$ if and only if* $\mathsf{And}(x, y) \in A$*. Similar, an* Or-function *d for A fulfills $x \in A$* or *$y \in A$ if and only if* $\mathsf{Or}(x, y) \in A$.

Before we can define the And- and Or-functions, we need some technical lemmas providing us with some *marking* or *labelling mechanism* for the variables of a Boolean formula such that a labelled variable is a *fixpoint of any automorphism* of the formula. It is not clear whether there exists such labellings that are efficiently computable. However, the following weaker labelling often suffices.

Let $F = F(x_1, \ldots, x_n)$ be a Boolean formula. We call variables x_i and x_j *equivalent* (with respect to F) if for any assignment a that satisfies F, we have $a(x_i) = a(x_j)$. Let $E(x_i)$ denote the set of variables that are equivalent to x_i. Now consider any automorphism $\varphi \in \mathrm{Aut}(F)$. If φ maps x_i to x_k, then φ must map all variables equivalent to x_i to variables that are equivalent to x_k, i.e., $\varphi(E(x_i)) = E(x_k)$. We conclude that, in this case, $E(x_i)$ and $E(x_k)$ must be of the same size. Thus, we can label variable x_i by taking n new variables $y_1, \ldots, y_n$, and make them equivalent to x_i as follows. Define

$$\begin{aligned} F_{[i]} &= F \wedge L(x_i, y_1, \ldots, y_n), \text{ where} \\ L(x_i, y_1, \ldots, y_n) &= \bigwedge_{j=1}^{n} (x_i \leftrightarrow y_j). \end{aligned}$$

Then x_i has more equivalent variables (with respect to $F_{[i]}$) than any other variable $x_k \notin E(x_i)$, and hence, any automorphism of $F_{[i]}$ stabilizes $E(x_i)$. Moreover, we can modify any automorphism to *pointwise* stabilize $E(x_i)$ and still have an automorphism of $F_{[i]}$. The new variables y_i of $F_{[i]}$ are referred to as the *labelling variables*.

Lemma 4.2 *For all $\varphi \in \mathrm{Aut}(F_{[i]})$,*

1. *$\varphi(E(x_i)) = E(x_i)$.*
2. *Define φ' to coincide with φ on all variables not in $E(x_i)$ and to be the identity on $E(x_i)$. Then $\varphi' \in \mathrm{Aut}(F_{[i]})$.*

Now let $G = G(x_1, \ldots, x_n)$ be a second formula. We label variable x_j with the same label as x_i, namely $L(x_j, y_1, \ldots, y_n)$ and define $G_{[j]} = G \wedge L(x_j, y_1, \ldots, y_n)$. Then any isomorphism for $(F_{[i]}, G_{[j]})$ must map all the variables equivalent to x_j in $G_{[j]}$ to the variables equivalent to x_i in $F_{[i]}$.

Corollary 4.3 *For all $\varphi \in \mathrm{Iso}(F_{[i]}, G_{[j]})$,*

1. *$\varphi(E_G(x_j)) = E_F(x_i)$.*
2. *Define φ' to coincide with φ on all variables not in $E_G(x_j)$ and to map x_j to x_i, be the identity on $y_1, \ldots, y_n$, and arbitrary on the remaining variables of $E_G(x_j)$. Then $\varphi' \in \mathrm{Iso}(F_{[i]}, G_{[j]})$.*

It follows that when two formulas $F_{[i]}$ and $G_{[j]}$ as above are isomorphic, we know that there is an isomorphism that maps x_j to x_i and keeps the new variables from the labelling process on themselves. We will therefore omit to explicitly mention the new variables in a label and will simply write $L(x_i, n)$ when we label x_i with n variables that do not yet occur in the considered formula.

A more general task is to force automorphisms to stabilize a set of variables. Let $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{y} = (y_1, \ldots, y_m)$, let $F = F(\mathbf{x}, \mathbf{y})$ and suppose we want to consider only automorphisms of F that map x- to x-variables and y- to y-variables. Let $n \leq m$. Extending the technique from Lemma 4.2, we could simply label variable y_i with $L(y_i, m)$, for $i = 1, \ldots, m$. However, the formula we get could increase quadratically in size. In order of being able to do this process iteratively a logarithmic number of times, the size of the formula we obtain should increase only linearly in size. Here is a trick to achieve this. Define

$$S(\mathbf{x}, \mathbf{y}, s, M) \quad = \quad (\bigvee_{i=1}^{n} x_i \rightarrow s) \quad \wedge \quad (\bigvee_{i=1}^{m} y_i \rightarrow \overline{s}) \quad \wedge \quad L(s, M).$$

Let $S = S(\mathbf{x}, \mathbf{y}, s, n+m)$. S has the following property: let a be a satisfying assignment of S. If a assigns a one to any of the x-variables, then $a(s) = 1$ which implies that $a(\overline{s}) = 0$, and therefore a must assign zero to all the y-variables. Symmetrically, if a assigns a one to any of the y-variables then $a(\overline{s}) = 1$ which implies that $a(s) = 0$, and therefore a must assign zero to all the x-variables.

Now consider $F \wedge S$. We claim that any automorphism of $F \wedge S$ must map x- to x-variables and y- to y-variables, unless they are equivalent.

Lemma 4.4 *Let F be a formula as above such that the all-zero assignment does not satisfy F. Let $\varphi \in \mathrm{Aut}(F \wedge S)$.*

1. *If $\varphi(x_i) = y_j$ for some i and j, then x_i and y_j are equivalent with respect to F.*
2. *Define φ' to coincide with φ on all variables where φ maps x- to x-variables and y- to y-variables, and to be the identity on the remaining variables. Then $\varphi' \in \mathrm{Aut}(F \wedge S)$.*

Proof. Any automorphism φ of $F \wedge S$ must stabilize s because of its label. Let a be an assignment that satisfies $F \wedge S$ and let x_i be a variable such that $a(x_i) = 1$ (recall that the all zero assignment does not satisfy F). Since $x_i \rightarrow s$ we have that $a(s) = 1$. Since $y_j \rightarrow \overline{s}$, we have that $a(y_j) = 0$ for $j = 1, \ldots, m$. Therefore, φ cannot map x_i to some y_j in order of $\varphi(a)$ to satisfy $F \wedge S$. We conclude that φ must map x_i to some x_j. It follows that whenever φ maps, say, x_i to y_j, then any satisfying assignment of $F \wedge S$ assigns zero to both, x_i and y_j. This means that x_i and y_j are equivalent (with respect to $F \wedge S$). □

We extend the lemma to isomorphisms. Let $G = G(\mathbf{x}, \mathbf{y})$. Then any isomorphism of $(F \wedge S, G \wedge S)$ must map x- to x-variables and y- to y-variables, unless they are equivalent.

Corollary 4.5 *Let F and G be formulas as above such that the all-zero assignment does not satisfy F or G. If $(F, G) \in \mathrm{BI}$ then there is a $\varphi \in \mathrm{Iso}(F \wedge S, G \wedge S)$ that maps all the x- to x-variables and all the y- to y-variables.*

For a last generalization step, consider again $F = F(\mathbf{x}, \mathbf{y})$ where $n = m$. Now we want to allow automorphisms of F to map x- variables to y-variables in the following way: either x-variables are only mapped to x-variables or x-variables are only mapped to y-variables. We can achieve this as follows. Define

$$ST(\mathbf{x}, \mathbf{y}, s, t, M) \quad = \quad (\bigvee_{i=1}^{n} x_i \rightarrow s) \quad \wedge \quad (\bigvee_{i=1}^{n} y_i \rightarrow \overline{s}) \quad \wedge \quad (s \leftrightarrow \overline{t}) \quad \wedge \quad L(s, M) \quad \wedge \quad L(t, M).$$

Let $ST = ST(\mathbf{x}, \mathbf{y}, s, t, 2n)$. As above for S, a satisfying assignment of ST can assign a one to either any of the x-variables or any of the y-variables, but not to both. The difference to S is that now an automorphism of ST can interchange s and t because they have the same label.

Now consider $F \wedge ST$. We claim that any automorphism of $F \wedge ST$

(i) either maps x- to x-variables and y- to y-variables,

(ii) or interchanges x- and y-variables,

unless they are equivalent.

Lemma 4.6 *Let F be a satisfiable formula as above such that the all-zero assignment does not satisfy F. For all $\varphi \in \mathrm{Aut}(F \wedge ST)$,*

(i) either $\varphi(s) = s$ and then we have that if $\varphi(x_i) = y_j$ for some i and j, then x_i and y_j are equivalent with respect to F,

(ii) or $\varphi(s) = t$ and then we have that if $\varphi(x_i) = x_j$ for some i and j, then x_i and x_j are equivalent with respect to F.

Furthermore, we can modify φ to an automorphism of $F \wedge ST$ that keeps x- on x-variables in case (i), and interchanges x- and y-variables in case (ii).

Proof. We have to distinguish two cases according to whether an automorphism φ maps s to s or t. In the case that $\varphi(s) = s$ we can directly use the proof of Lemma 4.4. If $\varphi(s) = t$, then by the same argument again, φ has to interchange all the x- and y-variables as well. □

Corollary 4.7 *Let F and G be formulas as above such that the all-zero assignment does not satisfy F or G. If $(F, G) \in$ BI then there is a $\varphi \in \mathrm{Iso}(F \wedge ST, G \wedge ST)$ that*

(i) either maps x- to x-variables and y- to y-variables,

(ii) or interchanges x- and y-variables,

Theorem 4.8 BI *has And- and Or-functions.*

Proof. Let (F_1', F_2') and (G_1', G_2') be two instances for BI, the F_i''s have variables $x_1, \ldots, x_{n-1}$ and the G_i''s have variables $y_1, \ldots, y_{m-1}$, and let $n \le m$.

Our first step is to switch to formulas

$$\begin{aligned} F_i &= F_i \wedge x_n \text{ and} \\ G_i &= G_i \wedge y_m, \end{aligned}$$

for $i = 1, 2$. The new formulas have the following properties.

(a) $(F_1', F_2') \in \mathrm{BI} \iff (F_1, F_2) \in \mathrm{BI}$ and $(G_1', G_2') \in \mathrm{BI} \iff (G_1, G_2) \in \mathrm{BI}$, and

(b) 0^n and 0^m are not satisfying assignments of F_1, F_2 and G_1, G_2, respectively.

Property (b) is obvious. To see (a) note that any satisfying assignment for F_1 or F_2 must set x_n to one. Thus, any isomorphism for (F_1, F_2) must stabilize $E(x_n)$, and hence there is an isomorphism that stabilizes x_n. This gives an isomorphism for (F_1', F_2').

For constructing the And-function, we simply combine the formulas by or-ing together F_1 and G_1 on one side, and F_2 and G_2 on the other side. However, we have to make sure that we don't get automorphisms that map x-variables to y-variables. For this we use formula $S = S(\mathbf{x}, \mathbf{y}, s, M)$ from above, where $M = n + m$. Define

$$\begin{aligned} \mathsf{And}((F_1, F_2), (G_1, G_2)) &= (C_1, C_2), \quad \text{where} \\ C_1 &= (F_1 \vee G_1) \ \wedge \ S \quad \text{and} \\ C_2 &= (F_2 \vee G_2) \ \wedge \ S \end{aligned}$$

If $(F_1, F_2) \in$ BI and $(G_1, G_2) \in$ BI, then clearly $(C_1, C_2) \in$ BI. For the reverse direction assume that $(C_1, C_2) \in$ BI. By Corollary 4.5 there is an isomorphism φ that maps x- to x-variables and y- to y-variables, i.e., φ can be written as $\varphi = \varphi_x \cup \varphi_y \cup \varphi_S$, where φ_x is a permutation on $\{x_1, \ldots, x_n\}$, φ_y on $\{y_1, \ldots, y_n\}$, and φ_S on the extra variables from formula S. Moreover, φ_x has to be an isomorphism for (F_1, F_2): we know that $g_1(0^m) = 0$, and $g_2(\varphi_y(0^m)) = g_2(0^m) = 0$. Therefore, we must have $f_1(a_x) = f_2 \circ \varphi_x(a_x)$ for any assignment a_x of the x-variables.

By a symmetric argument we have that φ_y must be an isomorphism for (G_1, G_2).

For constructing the Or-function, we need a copy of the variables $\mathbf{x} = (x_1, \ldots, x_n)$ and $\mathbf{y} = (y_1, \ldots, y_m)$ used yet. Let $\mathbf{u} = (u_1, \ldots, u_n)$ and $\mathbf{v} = (v_1, \ldots, v_m)$. Define

$$\begin{aligned}
\mathsf{Or}((F_1, F_2), (G_1, G_2)) &= (D_1, D_2), \quad \text{where} \\
D_1 &= ((F_1(\mathbf{x}) \vee G_1(\mathbf{y})) \ \vee \ (F_2(\mathbf{u}) \vee G_2(\mathbf{v}))) \ \wedge R \quad \text{and} \\
D_2 &= ((F_2(\mathbf{x}) \vee G_1(\mathbf{y})) \ \vee \ (F_1(\mathbf{u}) \vee G_2(\mathbf{v}))) \ \wedge \ R, \quad \text{where} \\
R &= S((\mathbf{x}, \mathbf{u}), (\mathbf{y}, \mathbf{v}), s_0, M_0) \ \wedge \ ST((\mathbf{x}, \mathbf{y}), (\mathbf{u}, \mathbf{v}), s, t, M),
\end{aligned}$$

where $M_0 = n + m$ and $M = 2M_0$. (The additional brackets for the variables in formulas S and ST indicate the two groups of variables occuring in the definition of these formulas.)

The rough idea of this definition is that if one of (F_1, F_2) or (G_1, G_2) are isomorphic then we can use such an isomorphism for (D_1, D_2) extended by the identity mapping for the other, maybe non-isomorphic, pair. Formula R will ensure that we don't get more isomorphisms than the ones just described.

Suppose first that $(F_1, F_2) \in$ BI and let φ_x be an isomorphism. Let φ_u be φ_x^{-1} but on the u-variables. That is, define

$$\varphi_u(u_i) \ = \ u_j, \quad \text{if } \varphi_x^{-1}(x_i) = x_j.$$

Define φ as the union of φ_x and φ_u and the identity on all the other variables of D_2. Then it is straight forward to check that φ is an isomorphism of (D_1, D_2) which is therefore in BI.

Now assume that $(G_1, G_2) \in$ BI via isomorphism φ_y. Then we get an isomorphism φ for (D_1, D_2) as follows. Define

$$\begin{aligned}
\varphi(x_i) &= u_i, & & \qquad \varphi(s) = t, \\
\varphi(u_i) &= x_i, & & \qquad \varphi(t) = s, \\
\varphi(v_i) &= y_j, & \text{if } \varphi_y(y_i) = y_j, & \qquad \varphi(s_0) = s_0, \\
\varphi(y_i) &= v_j, & \text{if } \varphi_y^{-1}(y_i) = y_j. &
\end{aligned}$$

The remaining variables coming from the labelling process are mapped according to the variables they are equivalent to.

In summary, the isomorphisms we constructed have the following property:

(i) either they map s and t to itself, respectively, and map x- to x-variables, u- to u-variables, y- to y-variables, and v- to v-variables,

(ii) or they interchange s with t and interchange x- with u-variables and y- with v-variables.

Conversely, if there is an isomorphism for (D_1, D_2) fulfilling property (i) or (ii), then it is easy to see that we get an isomorphism for either (F_1, F_2) or (G_1, G_2) from it, respectively. We now show that every isomorphism for (D_1, D_2) must (almost) satisfy one of these two properties.

Assume that (D_1, D_2) are isomorphic. We consider formula R. By Corollary 4.5, its first part, $S((\mathbf{x},\mathbf{u}),(\mathbf{y},\mathbf{v}),s_0,M_0)$, implies that there is an isomorphism φ of (D_1, D_2) that can be written as a union of permutations $\varphi_{x,u}$ on x- and u-variables, $\varphi_{y,v}$ on y- and v-variables, $\varphi_{s,t}$ on s and t, and φ_L for the remaining variables from the labelling. Combined with its second part, $ST((\mathbf{x},\mathbf{y}),(\mathbf{u},\mathbf{v}),s,t,M)$, we get by Corollary 4.7, that $\varphi_{x,u}$, depending on $\varphi_{s,t}$, either maps all x-variables to x-variables or interchanges x- and u-variables. The same holds analogously for $\varphi_{y,v}$. Thus we get an isomorphism fulfilling property (i) or (ii) above. $\square$

We remark that we can extend the And- and Or- functions to more than two arguments by combining them in a binary tree like fashion with the above functions for two arguments. Since the size of the output of our And- and Or- function is linear in the size of the input formulas, it is polynomial when having more arguments.

Corollary 4.9 *If a set L is disjunctively or conjunctively reducible to* BI, *then* $L \leq_m^p$ BI.

We give two applications of Corollary 4.9. The Boolean Automorphism problem, BA, is disjunctively reducible to BI, because a formula $F = F(x_1, \ldots, x_n)$ is in BA if and only if for some pair $i, j \in \{1, \ldots, n\},\ i \neq j$ we have that $(F_{[i]}, F_{[j]})$ is in BI. It follows that BA is many-one reducible to BI.

Corollary 4.10 BA $\leq_m^p$ BI.

We have already seen that BI is coNP hard. *Unique Satisfiability* (USAT) [BG82] is the set of all Boolean formulas that have exactly one satisfying assignment. The function $F(x_1, \ldots, x_n) \mapsto (F \wedge z)\ \vee\ (\bigwedge_{i=1}^{n} x_i \wedge \overline{z})$ reduces unsatisfiable formulas to uniquely satisfiable ones. Therefore USAT is coNP hard. Since USAT can be written as the difference of two NP sets, USAT is in $\mathrm{D^P}$, the second level of the Boolean Hierarchy. On the other hand, USAT is not known to be NP hard. The function $F(x_1, \ldots, x_n) \mapsto (F, \bigwedge_{i=1}^{n} x_i)$ reduces USAT to BC, the Boolean Congruence problem. Since BI $\equiv_m^p$ BC [BRS95], USAT can be reduced to BI.

A seemingly harder problem than USAT is the *Unique Optimal Clique problem* (UOCLIQUE), that is, whether the largest clique of a given graph is unique. The standard reduction from SAT to CLIQUE also reduces USAT to UOCLIQUE. An upper bound for the complexity of UOCLIQUE is $\mathrm{P^{NP[log]}}$: with logarithmically many queries to an NP oracle one can compute the size of the largest clique of a given graph. Then, with one more query, one can find out whether there is more than one clique of that size. Papadimitriou and Zachos [PZ83] asked whether UOCLIQUE is complete for $\mathrm{P^{NP[log]}}$. This is still an open problem. Buhrman and Thierauf [BT96] provide strong evidence that UOCLIQUE is *not* complete for $\mathrm{P^{NP[log]}}$.

UOCLIQUE can be disjunctively reduced to USAT [BT96]. To see this let UCLIQUE be the unique CLIQUE version. That is, given a graph G and a integer k, decide whether G has a unique clique of size k. Now, observe that $G \in$ UOCLIQUE $\iff \exists k : (G, k) \in$ UCLIQUE. Since the Cook reduction is parsimonious, this provides a disjunctive reduction of UOCLIQUE to USAT.

Combined with the reduction from USAT to BI, we have that UOCLIQUE can be disjunctively reduced to BI. By Corollary 4.9, this can be turned into a many-one reduction.

Corollary 4.11 UOCLIQUE $\leq_m^p$ BI.

As for UOCLIQUE, it is not known whether BI is NP hard.

5 The Counting Version of BI

Mathon [Mat79] showed that the counting version of the Graph Isomorphism problem can be (truth-table) reduced to the decision version. Thus GI behaves differently than the known NP complete problems. This was historically the first hint that GI might *not* be NP complete.

We will show an anlogous result for BI. The proof follows essentially the one for GI, however, there is again a technical difficulty to get around. For GI, the idea is as follows.

> First of all it is enough to compute the number of automorphism of a graph G, because there are exactly as many isomorphism to any graph, G is isomorphic to. In the beginning, label all nodes of G (with pairwise different labels), so that the identity is the only automorphism of the resulting graph. Then successively take away the labels. If i is the node where the label was cancelled last, compute the orbit of i by asking queries to GI. When all labels are taken away, the number of automorphisms of G is the product of the orbit sizes constructed during this procedure.

In the above algorithm one needs to construct a graph $G_{[I]}$, where $I \subseteq \{1, \ldots, n\}$ such that any automorphism of $G_{[I]}$ pointwise stabilizes the nodes in I. Correspondingly, given a formula F in n variables, we need to construct a formula $F_{[I]}$ whose set of automorphisms (roughly) corresponds to the pointwise stabilizer of I in $\mathrm{Aut}(F)$. More precisely, $F_{[I]}$ must retain exactly those automorphisms of F that map variables in $E(x_i)$ to themselves, for all $i \in I$. Observe that the labelling method given in Section 4 takes m new variables to label a variable in a formula with m variables. Thus, starting with n variables, we would get $n(2^{|I|} - 1)$ new variables to carry out the marking which is exponential in n when $|I| = \Theta(n)$. This is clearly too much in general.

Here, we give a method of computing $F_{[I]}$ that works in $\mathrm{FP}^{\mathrm{NP}}$. Recall that BI is coNP hard. Therefore we can especially use this method when having BI as an oracle.

Lemma 5.1 *$F_{[I]}$ is computable in* $\mathrm{FP}^{\mathrm{NP}}_{||}$.

Proof. Recall from Section 4 that for any automorphism φ of F, we have $|E(x_i)| = |E(\varphi(x_i))|$ for any variable x_i. The trick in Lemma 4.2 was to make $|E(x_i)|$ unique by appending enough equivalent variables to x_i. Now, with an NP oracle, we can actually compute sets $E(x_i)$ by finding out whether F is equivalent with $F \wedge (x_i \leftrightarrow x_j)$, for all j. Then, if we want to label x_i, we take the smallest label such that x_i gets a unique number of equivalent variables. This will keep the number of new variables needed small.

We construct formula $F_{[I]}$ by successively labelling the variables in I. Let $F_0 = F$. Suppose that we have already labelled the first k variables of I and obtained the formula F_k, for some $k \geq 0$. Say that x_i is the next variable to label, for some $i \in I$. That is, x_i is not equivalent to any of the variables already labelled. Now, let t be the smallest number such that, with respect to F_k, we have

$$|E(x_i)| + t \neq |E(x_j)| \quad \text{for any} \quad x_j \notin E(x_i),$$

and define $F_{k+1} = F_k \wedge L(x_i, t)$. This ensures that any automorphism of the new formula F_{k+1} stabilizes the set $E(x_i)$. Thus F_{k+1} has the desired property. Furthermore, the only equivalence class of variables that changed in this process is $E(x_i)$ which has now all the new variables added to it. Therefore, no equivalence class will have size more than n during any stage of the above construction, and thus the number t will be at most n. It follows that in total at most n^2 new variables are introduced, and the construction works in polynomial time. □

Using the labelling technique from Lemma 5.1, we can compute the number of isomorphisms of a Boolean formula in polynomial time relative to BI.

Theorem 5.2 $\#\mathrm{BI} \in \mathrm{FP}_{||}^{\mathrm{BI}}$.

Mathon's algorithm for computing the number of isomorphisms of two graphs is an inductive process, where at each intermediate stage one knows the number of isomorphisms of the labelled graphs that are considered. This observation led to the result that GI $\in$ LWPP [KST92]. Since LWPP is low for PP, that is, $\mathrm{PP}^{\mathrm{LWPP}} = \mathrm{PP}$ [FFK94], it follows that GI is low for PP.

Using Lemma 5.1, an analogous argument shows that BI $\in \mathrm{LWPP}^{\mathrm{NP}}$. Since $\mathrm{PP}^{\mathrm{LWPP}^{\mathrm{NP}}} = \mathrm{PP}^{\mathrm{NP}}$, we get kind of a lowness result for BI.

Theorem 5.3 $\mathrm{PP}^{\mathrm{BI}} = \mathrm{PP}^{\mathrm{NP}}$.

6 BI is self-reducible

A set A is self-reducible if, very informally, the decision problem whether a given instance x is in A can be reduced to the same problem but for smaller (under some ordering) instances. Self-reducibility is a very useful property of a set. For example, if an NP set is self-reducible then the (seemingly more complex) construction problem, i.e., constructing a witness, can be reduced to the decision problem. In this section we show that BI is self-reducible.

Definition 6.1 *A partial ordering $\prec$ on Σ^* is* polynomially related *if $\prec$ is decidable in polynomial time and there is a polynomial p such that*

(i) for any $x, y \in \Sigma^*$, we have $x \prec y \Longrightarrow |x| \leq p(|y|)$,

(ii) any chain is polynomially length bounded, that is, if $x_1 \prec x_2 \prec \cdots \prec x_k$, then $k \leq p(|x_k|)$.

Definition 6.2 *A set A is* self-reducible *if there is a polynomially related partial ordering $\prec$ and a deterministic polynomial-time Turing machine M such that*

(i) M^A accepts A and

(ii) on input x and any oracle B, M^B queries only strings y such that $y \prec x$.

Theorem 6.3 BI *is self-reducible.*

Proof. We first give an informal description of the self-reducing machine for BI. For two formulas F and G, we have that

$$(F, G) \in \mathrm{BI} \Longleftrightarrow \exists i, j : \ (F_{[i]}, G_{[j]}) \in \mathrm{BI}.$$

Thus, the self-reducing machine simply constructs these formulas for all values of i and j and queries the oracle. However, we need to define a polynomially related partial ordering according to these queries. There are some subtle points that one has to take care of.

The self-reducing machine uses the labelling scheme from Lemma 5.1, which yields a polynomially sized formula even after several labellings. To ensure polynomially-sized chains, the machine must check if some variables in F and G are already labelled, and, in this case, not relabel them. This is possible since a label is easily detectable: it is of the form $x \leftrightarrow y$. We use the variable with the smallest index as a representative for a label and call it a *basic variable* of F. Note that the variables that don't have a label are also basic variables. The other variables we refer to as *labelling variables*. The *label size* of a basic variable is the number of labelling variables labelling it.

When all the basic variables of F and G have a unique label size, they define a permutation on the variables which the machine can use to permute the variables of F. Then it checks if the permuted formula F is equivalent to G. This checking is done by setting the first variable of both formulas to zero and one, respectively, and then verifying that both pairs of the resulting formulas are equivalent. Note that all tests can be done with BI as an oracle.

The underlying polynomially related partial ordering $\prec$ is defined in the following way. By True_n we denote a fixed formula over n variables that is a tautology (e.g., $\bigwedge_{i=1}^{n}(x_i \vee \bar{x}_i)$). In the following, we drop the subscript n and simply write True to denote such a formula when the number of variables is clear from the context. For formulas F, G, F', G', we define $(F, G) \prec (F', G')$ if $|F|$ and $|G|$ is bounded by a fixed polynomial in $|F'|$ and $|G'|$ respectively, and one of the following conditions hold:

(i) the number of basic variables in F is less than that in F', and either $G = \mathsf{True}$ or the number of basic variables in G is less than or equal to that in G',

(ii) the number of basic variables in G is less than that in G', and either $F = \mathsf{True}$ or the number of basic variables in F is less than or equal to that in F', or

(iii) the number of basic variables in F and G equals that in F' and G', and more basic variables in F and G have unique label sizes than in F' and G', respectively.

We now describe the self-reducing machine, M, in detail. At several places, M has to test whether a Boolean formula T is a tautology. This is done by checking that $(T[0], \mathsf{True})$ and $(T[1], \mathsf{True})$ belong to BI, where by $T[b]$, $b \in \{0, 1\}$, we denote the formula obtained from T by setting its first basic variable—and the labelled variables associated with it—to the value b.

Let F and G be formulas over variables $x_1, \ldots, x_n$.

> M on input (F, G) first checks if any of F and G equals True. If both of them do, then they are isomorphic and so M accepts. If exactly one of them, say G, does, then F and G are isomorphic if F is also a tautology. This can be checked using the scheme described above. M accepts iff F is a tautology.
>
> If none of F and G equals True, M does the following. It detects the basic variables of F and G, and finds out if any two basic variables of F are equivalent by checking if the formula $T_{F,i,j}$ is a tautology, where
>
> $$T_{F,i,j} = (F \wedge (x_i \leftrightarrow x_j)) \leftrightarrow F,$$
>
> for every pair of basic variables x_i and x_j of F. If $T_{F,i,j}$ is a tautology then x_i and x_j are equivalent in F, otherwise not. If there are x_i and x_j in F that are equivalent then M accepts iff $(F', G) \in$ BI where F' is obtained from F by replacing all occurrences of x_j in F by x_i. Then we 'and' the formula $x_i \leftrightarrow x_j$ to the resulting formula. By this transformation, F and F' are equivalent and F' has one less basic variables than F. If F has no equivalent basic variables, then the above is repeated for G instead of F.
>
> If no two basic variables of F or G are equivalent, M computes, for each basic variable, its label size. It then checks whether these numbers of F and G match. If not, then M rejects as there cannot be any isomorphism between F and G.
>
> If the numbers match, and all basic variables of F are uniquely labelled, then M constructs a permutation φ of variables of F such that $\varphi(x_i) = x_j$, where the label

size of x_i in F and x_j in G are the same. φ also maps labelled variables associated with x_i to those associated with x_j. Now, M permutes the variables of F using φ to obtain the formula $F \circ \varphi$ and then checks whether $F \circ \varphi \leftrightarrow G$ is a tautology, and accepts in this case.

Finally, if there are some basic variables of F with identical label sizes, for every such variable x_i of F, and for every basic variable x_j of G that has the same label size, M queries the oracle whether $(F_{[i]}, G_{[j]}) \in$ BI. It accepts iff at least one of these pairs belong to BI.

It is straight forward to see that M respects the partial order defined above, works in polynomial-time, and accepts BI. □

7 Open Problems

In the known examples, isomorphism does not appear to add full NP power to the corresponding equivalence problem, as in the case of graphs and Boolean formulas. Note that for graphs, the equivalence asks for equality, which is trivial. The equivalence of two deterministic finite automatons (DFA) can be decided in P. It is not hard to see that the isomorphism problem for DFA's, where one can permute the states of a DFA, is still in P.

The equivalence problem for one-time-only branching programs is known to be in coRP. Therefore, the corresponding isomorphism problem is in NP·coRP. We ask for some better bound on the complexity of the isomorphism problem for one-time-only branching programs.

Acknowledgements

We benefited from discussions with V. Arvind, Bernd Borchert, Jin-yi Cai, Toni Lozano, and Lance Fortnow.

References

[Bab85] L. Babai. Trading group theory for randomness. In *17th ACM Symposium on Theory of Computing*, 421-429, 1985.

[BDG-I&II] J. Balcázar, J. Díaz, and J. Gabarró. Structural Complexity I & II. EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1988 und 1991.

[BCGKT95] N. Bshouty, R. Cleve, R. Gavaldà, S. Kannan, C. Tamon. Oracles and queries that are sufficient for exact learning. ECCC TR95-015, 1995. Available via: http://www.eccc.uni-trier.de/eccc/

[BG82] A. Blass, Y. Gurevich. On the unique satisfiability problem. *Information and Control 55*, 80-88, 1982.

[BR93] B. Borchert, D. Ranjan. The Subfunction Relations are Σ_2^p–complete, Technical Report MPI-I-93-121, MPI Saarbrücken, 1993.

[BRS95] B. Borchert, D. Ranjan, F. Stephan. On the Computational Complexity of some Classical Equivalence Relations on Boolean Functions. Forschungsberichte Mathematische Logik, Universität Heidelberg, Bericht Nr. 18, Dezember 1995.

[BCW80] M. Blum, A. Chandra, M. Wegman. Equivalence of free Boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters 10(2)*, 80-82, 1980.

[BT96] H. Buhrman, T. Thierauf. The complexity of generating and checking proofs of membership. In *Symposium on Theoretical Aspects of Computer Sience (STACS) '96* Springer Verlag, Lecture Notes in Computer Sience 1046, 75-87,1996.

[CK91] P. Clote, E. Kranakis. Boolean functions, invariance groups, and parallel complexity. *SIAM Journal on Computing 20(3)*, 553-590, 1991.

[FFK94] S. Fenner, L. Fortnow, and S. Kurtz, Gap-definable counting classes, *J. Comput. System Sci.* **48** (1994), 116–148.

[GMR89] S. Goldwasser, S. Micali, C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing 18*, 186-208, 1989.

[GS89] S. Goldwasser, M. Sipser. Privat coins versus public coins in interactive proof systems. *Advances in Computing Research. Vol. 5: Randomness and Computation*, S. Micali (Ed.), JAI Press, 73-90, 1989

[Hof82] C. Hoffmann. Group-theoretic algorithms and graph isomorphism. Springer Verlag, Lecture Notes in Computer Sience 136, 1982.

[HU79] J. Hopcroft, J. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 1979.

[KST92] J. Köbler, U. Schöning, J. Toran. Graph Isomorphism is low for PP. *Journal of Computational Complexity 2*, 301-330, 1992.

[KST93] J. Köbler, U. Schöning, J. Toran. The Graph Isomorphism Problem: Its Structural Complexity. Birkhäuser Verlag, 1993.

[Mat79] R. Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters 8*, 131-132, 1979.

[PZ83] C. Papadimitriou and D. Zachos. Two remarks on the power of counting. In *6th GI Conference on TCS*, Springer Verlag LNCS 145, pages 269–276, 1983.

[Sch88] U. Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences 37*, 312-323, 1988.

[Sch89] U. Schöning. Probabilistic complexity classes and lowness. *Journal of Computer and System Sciences 39*, 84-100, 1989.