


Transformations of Boolean Functions

Jeffrey M. Dudek 

Department of Computer Science, Rice University, Houston, TX, USA
jmd11@rice.edu

Dror Fried

Department of Mathematics and Computer Science, The Open University of Israel, Ra'anana, Israel
dfried@openu.ac.il

Abstract

Boolean functions are characterized by the unique structure of their solution space. Some properties of the solution space, such as the possible existence of a solution, are well sought after but difficult to obtain. To better reason about such properties, we define *transformations* as functions that change one Boolean function to another while maintaining some properties of the solution space. We explore transformations of Boolean functions, compactly described as Boolean formulas, where the property is to maintain is the number of solutions in the solution spaces. We first discuss general characteristics of such transformations. Next, we reason about the computational complexity of transforming one Boolean formula to another. Finally, we demonstrate the versatility of transformations by extensively discussing transformations of Boolean formulas to “blocks,” which are solution spaces in which the set of solutions makes a prefix of the solution space under a lexicographic order of the variables.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Boolean Formulas, Boolean Functions, Transformations, Model Counting

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2019.39

Funding *Jeffrey M. Dudek*: Supported by NSF grants DMS-1547433 and IIS-1527668, by the Big-Data Private-Cloud Research Cyberinfrastructure MRI-award funded by NSF under grant CNS-1338099, by the Ken Kennedy Institute Computer Science & Engineering Enhancement Fellowship funded by the Rice Oil & Gas HPC Conference, and by the Ken Kennedy Institute 2017/18 Cray Graduate Fellowship.

Acknowledgements We would like to thank Kuldeep S. Meel, Moshe Y. Vardi, and Rice’s Computer-Aided Verification Group for useful discussions.

1 Introduction

Boolean functions play an integral part in many areas in computer science, electrical engineering and more [22, 11]. For example by abstracting properties of a system as a true/false dichotomy, one can model such properties as a Boolean function where a positive (true) output of that formula means that the property appears in the system. Typically every Boolean function can be uniquely characterized by its *solution space*, also called a *truth table*, which is a table that assigns the true/false output of the function for every possible assignment to the Boolean inputs. Since the size of such a table can be very large, in particular exponential in the number of variables, more compact representations of Boolean functions are used, such as Boolean formulas, Karnaugh maps [15], and Boolean Decision Diagrams (BDDs) [8]. Such compact representations, however, come at a cost since reasoning about properties of the Boolean function such as whether a solution exists, or counting the number of solutions, becomes a challenging problem. A question to ask, therefore, is whether one can better reason about properties of a Boolean function by “transforming” the function to a different Boolean function while still preserving some of the original properties.



© Jeffrey M. Dudek and Dror Fried;
licensed under Creative Commons License CC-BY

39th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2019).

Editors: Arkadev Chattopadhyay and Paul Gastin; Article No. 39; pp. 39:1–39:14



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this work, we lay foundations for thematic exploration of such transformations of Boolean functions. In our setting we use Boolean formulas to describe Boolean functions, and the property of the solution space that we maintain is the number of solutions in the solution space. In general, counting the number of solutions is a problem of great importance [12, 5, 16], which is known to be a #P-hard problem [21] and there are numerous works, both theoretical [20, 13] and applied [19, 17] that address this problem. In all this, the structure of the solution space can play an important role in the attempts for obtaining efficient counting [9, 7, 2].

In our formulation, transformations are quantified Boolean formulas that describe bijections between the output columns of solution spaces with the same number of variables. Thus the result of applying such a transformation T to a Boolean formula φ is a Boolean formula ψ with the same number of variables and same number of solutions as φ has.

This paper can be separated into three parts. In the first part we define transformations and discuss properties of transformations such as closure under composition and the inverse operation. We ask whether every pair of Boolean formulas with the same number of variables always has an expressible, polynomially-sized transformation between them. We discuss this in the second part and give an affirmative answer if the number of alternating quantifiers is not limited. Moreover, we show that if the number of alternations is bounded then the question is equivalent to the collapse of the polynomial hierarchy.

In the third part of the paper we present various transformations and combination of transformations that demonstrate the versatility of our framework. For that, we focus on a specific solution space structure called a “block,” in which the set of solutions form a prefix of the solution space, under a lexicographic order of the variables. Boolean formulas with such a block-type solution space— for example, chain formulas [9]— have efficient counting. We describe several techniques to construct transformations that merge solutions together in order to transform a solution space to a block. Specifically, we describe a method to, for an arbitrary given Boolean formula φ , construct a transformation (possibly of exponential size) that can transform φ to a block. We then present classes of transformations that can transform certain specialized types of solution spaces into blocks. The transformations in this part are “parameterized,” in the sense that there are certain parameters for the transformations that are adjusted according to the given Boolean formula.

Finally, a divide-and-conquer approach for a solution is a general technique also used in studies of Boolean functions [14, 10]. We show a transformation that can efficiently transform specific Boolean formulas to a block by first finding transformations to blocks for each sub-formula in a divide-and-conquer manner.

2 Preliminaries

A *Boolean function* $f : \{0, 1\}^n \rightarrow \{0, 1\}$ is a function that assigns n input Boolean variables $\{x_1, \dots, x_n\}$ to a Boolean output 0 (*false*) or 1 (*true*). The *solution space*, also called a *truth table*, of f is the explicit description of f as a table of a size 2^n . A *solution* however is an assignment $\vec{\sigma}$ for which $f(\sigma) = 1$. Thus the solution space of f contains solutions and non-solutions.

Throughout this work, we generally assume that n is fixed and use \vec{x} to denote a sequence of n variables x_n, \dots, x_1 . Moreover, we fix the order of the sequence of variables, thus have a lexicographic order \leq_{lex} on the truth assignments to \vec{x} , where x_n is the most significant bit (msb) and x_1 the least significant bit (lsb). We also define a function

$bin : \{0, \dots, 2^n - 1\} \rightarrow \{0, 1\}^n$ that maps the integers $0 \leq c \leq 2^n - 1$ to their natural encoding as a corresponding assignment to n variables (e.g. for $n = 4$, $bin(3) = 0011$), and the corresponding inverse function $|\cdot| : \{0, 1\}^n \rightarrow \{0, \dots, 2^n - 1\}$ (then $|0011| = 3$).

One compact description of a Boolean function f is by a *Boolean formula* $\varphi_f(x_1, \dots, x_n)$ that is satisfied exactly when f is 1. When f is obvious or irrelevant, we omit f from the notation of φ_f . Every Boolean formula φ over n variables describes a unique solution space of size 2^n . The number of solutions to φ , i.e. the number of assignments that set φ to *true* is denoted by $\#\varphi$. The *size* of φ is defined as the length of φ and is denoted by $|\varphi|$. Two Boolean formulas φ, ψ that describe the same Boolean function are called *logically equivalent*, denoted $\varphi \equiv \psi$, and by definition such formulas have identical solution spaces.

In our settings, we also reason about partial solution spaces (also called *subspaces*). For a solution space S , the partial solution $S[\sigma_n, \dots, \sigma_{i+1}]$ of size 2^i is obtained by assigning $\sigma_j \in \{0, 1\}$ to the variable x_j for each j such that $i + 1 \leq j \leq n$. We denote by

$$S[x_n, \dots, x_{i+1}] = \{S[\sigma_n, \dots, \sigma_{i+1}] \mid (\sigma_n, \dots, \sigma_{i+1}) \in \{0, 1\}^{n-i}\}$$

the set of all solution spaces of size 2^i obtained by fixing x_n, \dots, x_{i+1} to every assignment. When $\vec{\sigma}$ is a complete assignment for the variables $x_n \dots x_1$, then $S[\vec{\sigma}]$ denotes the value of the assignment $\vec{\sigma}$ in S (i.e. *true* or *false*). A *null* solution space is a solution space in which all its assignments are valued to 0.

A *block* is a solution space whose set of solutions make a prefix under the fixed lexicographic order \leq_{lex} . That is, a (possibly partial) solution space S of size 2^i is a block if, for every pair of assignments $\vec{\sigma}, \vec{\sigma}' \in \{0, 1\}^i$ where $S[\vec{\sigma}] = 1$ and $\vec{\sigma}' \leq_{lex} \vec{\sigma}$, it holds that $S[\vec{\sigma}'] = 1$. We can also describe a block S by its output column as $1^k 0^{2^i - k}$ for some positive integer k . In this case, k is exactly the number of solutions in S . Finally for every $0 \leq c \leq 2^n$, we define $block_c$ to be the formula over n variables whose solution space is the block with c solutions. That is, $block_c(\vec{x}) \equiv (\vec{x} \leq_{lex} bin(c)) \wedge (\vec{x} \neq bin(c))$. Note that $block_c$ can be written as a Boolean formula (in particular, as a *chain formula* [9]). When obvious from the context we sometimes refer to the formula $block_c$ as a block with c solutions.

3 Definitions and properties

Given a function $g : \{0, 1\}^n \rightarrow \{0, 1\}^m$ for some integers n, m , we say that a quantified Boolean formula $F(\vec{x}, \vec{y})$ *describes* g if \vec{x} (called the *domain variables*) is of size n , \vec{y} (called the *range variables*) is of size m , and for every assignments \vec{a}, \vec{b} for \vec{x}, \vec{y} respectively we have $F(\vec{a}, \vec{b}) = true$ iff $g(\vec{a}) = \vec{b}$.

We now define transformations as follows:

► **Definition 1.** A transformation $T(\vec{x}, \vec{y})$ is a quantified Boolean formula over $2n$ free variables that describes a bijection from $\{0, 1\}^n$ to $\{0, 1\}^n$.

We assume without loss of generality that all transformations are described in a prenex normal form. The *size* of a transformation T is the number of symbols in the underlying QBF formula, denoted $|T|$. Although we allow arbitrary alternation of quantifiers in transformations, one might consider restricting to transformations in Σ_k^P (for some fixed k) in order to limit the number of quantifier alternations and hence ease reasoning. In particular, restricting to transformations in Σ_1^P (i.e. using only existential quantifiers), allows reasoning on such transformations by using SAT solvers, while still maintaining some expressiveness, e.g. by using “carry” bits as we see in Section 5.1. Unless mentioned otherwise, for the rest of the paper we assume that all the Boolean formulas have n free variables and all transformations have $2n$ free variables.

$x_1 \vee x_3$	$XOR_{0,1,1}(x_1 \vee x_3)$
0	1
1	0
0	1
1	0
1	1
1	1
1	1
1	1

■ **Figure 1** The truth tables of the formula $x_1 \vee x_3$ and of the result after applying the transformation $XOR_{0,1,1}$. Each truth table is given in lexicographic order from top (where $x_3 = x_2 = x_1 = 0$) to bottom (where $x_3 = x_2 = x_1 = 1$).

We now define how to apply a transformation to a Boolean formula. Given a transformation $T(\vec{x}, \vec{y})$ that describes a bijection $g : \{0, 1\}^n \rightarrow \{0, 1\}^n$ and a Boolean formula $\varphi(\vec{z})$ over n free variables, we apply T to φ by constructing a new Boolean formula $Apply_{T,\varphi}$ over n free variables in which we identify the domain variables of T with the variables of φ :

$$Apply_{T,\varphi}(\vec{y}) \equiv \exists \vec{x} (T(\vec{x}, \vec{y}) \wedge \varphi(\vec{x})).$$

For convenience, we denote $Apply_{T,\varphi}$ by $T(\varphi)$. If ψ is a Boolean formula over n variables and $\psi \equiv T(\varphi)$, we say that T transforms φ into ψ . Note that we can also think of a transformation as a function from Boolean formulas to Boolean formulas. Also notice that $T(\varphi)$ is a Boolean formula with n free variables whose solution space is resulted by applying g (i.e., the bijection described by T) to the solution space of φ . That is, \vec{a} is a solution of φ if and only if $g(\vec{a})$ is a solution of $T(\varphi)$. Since g is a bijection, it follows that φ and $T(\varphi)$ indeed have the same number of solutions. Although transformations are defined over formulas, to ease the reading we sometimes say that we apply transformations to a solution space, in which case we mean that we apply the transformation to a formula with the mentioned solution space.

► **Example 2.** The simplest transformation is the *identity* transformation $id(\vec{x}, \vec{y}) \equiv \bigwedge_i x_i \leftrightarrow y_i$. For all Boolean formulas φ , $id(\varphi) \equiv \varphi$.

► **Example 3.** The transformation $T_1(x_1, x_2, y_1, y_2) \equiv (x_1 \leftrightarrow y_2) \wedge (x_2 \leftrightarrow y_1)$ is a transformation over 4 free variables. When applied to a Boolean formula $\varphi(x_1, x_2)$, T_1 syntactically switches between x_1 and x_2 . That is, $T_1(\varphi(x_1, x_2)) \equiv \varphi(x_2, x_1)$.

► **Example 4.** For a given vector $\vec{a} \in \{0, 1\}^n$, the bijection that maps every $\vec{b} \in \{0, 1\}^n$ to $\vec{b} \oplus \vec{a}$ is represented by the XOR transformation:

$$XOR_{\vec{a}}(\vec{x}, \vec{y}) \equiv \bigwedge_i (y_i \leftrightarrow (x_i \oplus a_i)).$$

Figure 1 describes an example of the XOR transformation.

3.1 Properties of transformations

We consider transformations as combinatorial objects that can be used to construct other, more complicated transformations. For that, we describe a few simple algebraic properties of transformations and then define the composition of two transformations.

We begin by listing a few simple properties of transformations, which follow directly from the fact that transformations describe bijections:

▷ **Claim 5.** Let T be a transformation and let φ and ψ be Boolean formulas. Then:

1. $T(\varphi \vee \psi) \equiv T(\varphi) \vee T(\psi)$
2. $T(\varphi \wedge \psi) \equiv T(\varphi) \wedge T(\psi)$
3. $T(\neg\varphi) \equiv \neg(T(\varphi))$

We next consider compositions of transformations. Let T_1 and T_2 be transformations that describe bijections g_1 and g_2 . We define the *composition* of T_1 and T_2 , denoted $T_2 \circ T_1$, to be a transformation that describes the composition of g_1 and g_2 (which is the bijection $g_2 \circ g_1$ that maps each $\vec{a} \in \{0, 1\}^n$ to $g_2(g_1(\vec{a}))$). Note that composition can be described by a simple syntactic construction of $T_2 \circ T_1$ from T_1 and T_2 as the following claim shows.

▷ **Claim 6.** $(T_2 \circ T_1)(\vec{x}, \vec{y}) \equiv \exists \vec{z} (T_1(\vec{x}, \vec{z}) \wedge T_2(\vec{z}, \vec{y}))$

Naturally, we also have that for an arbitrary Boolean formula φ , applying $T_2 \circ T_1$ to φ is logically equivalent to applying T_1 to φ followed by applying T_2 :

▷ **Claim 7.** $(T_2 \circ T_1(\varphi))$ is logically equivalent to $T_2(T_1(\varphi))$.

Proof. We have that:

$$\begin{aligned} T_2(T_1(\varphi))(\vec{z}) &\equiv \exists \vec{y} (T_2(\vec{y}, \vec{z}) \wedge T_1(\varphi)(\vec{y})) \\ &\equiv \exists \vec{y} (T_2(\vec{y}, \vec{z}) \wedge \exists \vec{x} (T_1(\vec{x}, \vec{y}) \wedge \varphi(\vec{x}))) \\ &\equiv \exists \vec{x} (\exists \vec{y} (T_1(\vec{x}, \vec{y}) \wedge T_2(\vec{y}, \vec{z})) \wedge \varphi(\vec{x})) \equiv (T_2 \circ T_1)(\varphi)(\vec{z}) \quad \triangleleft \end{aligned}$$

Notice that if T_1 and T_2 are both in Σ_k^P for some k , then Claim 6 proves that their composition $T_2 \circ T_1$ is in Σ_k^P as well. We will specifically use this fact for the merge-rotation transformations described in Section 5.2 which are in Σ_1^P . A transformation constructed by composition of many Σ_1^P transformations is also in Σ_1^P and so can still be reasoned about with a SAT solver.

Along the same lines, we define the *inverse transformation* of a transformation T (that describes a bijection g) to be a transformation T^{-1} that describes the inverse bijection g^{-1} . As with composition, there is a simple syntactic construction of T^{-1} from T by swapping the domain and range variables:

▷ **Claim 8.** $T^{-1}(\vec{x}, \vec{y}) \equiv T(\vec{y}, \vec{x})$

Proof. Let g be the bijection described by T . Notice that for every assignment \vec{a} and \vec{b} to \vec{x} and \vec{y} we have that $T(\vec{b}, \vec{a}) = \text{true}$ if and only if $g(\vec{b}) = \vec{a}$, which occurs if and only if $g^{-1}(\vec{a}) = \vec{b}$. Hence $T(\vec{y}, \vec{x})$ indeed describes g^{-1} . \triangleleft

As a direct result from Claim 8, we get that the inverse transformation is indeed an inverse under the composition operator as follows.

▶ **Corollary 9.** Let T be a transformation and φ be a Boolean formula. Then $(T \circ T^{-1})(\varphi) \equiv (T^{-1} \circ T)(\varphi) \equiv \varphi$.

4 Transformations and the polynomial hierarchy

In this work the transformations that we define do not affect the number of solutions of a formula. In particular, if a transformation transforms a Boolean formula φ_1 into φ_2 then the number of solutions of φ_1 and φ_2 must be the same. Perhaps surprisingly, the converse is also true: if two Boolean formulas φ_1 and φ_2 over the same number of variables n have the same number of solutions then there must be a transformation of size polynomial in $\max\{n, |\varphi_1|, |\varphi_2|\}$ that transforms φ_1 into φ_2 . We give this result as the following theorem.

► **Theorem 10.** *There is a polynomial $p : \mathbb{N}^3 \rightarrow \mathbb{N}$ such that, if φ_1 and φ_2 are two Boolean formulas with n variables each and the same number of solutions, then there is a transformation T of size no more than $p(n, |\varphi_1|, |\varphi_2|)$ such that $T(\varphi_1) \equiv \varphi_2$.*

Proof. For a given Boolean formula φ of n variables and an assignment $\{0, 1\}^n$, let $H_\varphi^1(\vec{a})$ be the set of solutions strictly smaller, under \leq_{lex} than \vec{a} . That is $H_\varphi^1(\vec{a}) = \{\vec{c} \mid \vec{c} <_{lex} \vec{a} \wedge \varphi(\vec{c}) = 1\}$. Similarly, let $H_\varphi^0(\vec{a}) = \{\vec{c} \mid \vec{c} <_{lex} \vec{a} \wedge \varphi(\vec{c}) = 0\}$ be the set of non-solutions strictly smaller than \vec{a} .

Now let $\mathcal{L} \subseteq \{0, 1\}^n \times \{0, 1\}^n$ be the set of $(\vec{a}, \vec{b}) \in \{0, 1\}^n \times \{0, 1\}^n$ such that either: (i) $\varphi_1(\vec{a}) = \varphi_2(\vec{b}) = 1$, and $|H_{\varphi_1}^1(\vec{a})| = |H_{\varphi_2}^1(\vec{b})|$, or; (ii) $\varphi_1(\vec{a}) = \varphi_2(\vec{b}) = 0$, and $|H_{\varphi_1}^0(\vec{a})| = |H_{\varphi_2}^0(\vec{b})|$.

Since for an arbitrary $\vec{a} \in \{0, 1\}^n$ and an arbitrary Boolean formula φ , both $|H_\varphi^1(\vec{a})|$ and $|H_\varphi^0(\vec{a})|$ can be computed by using a single $\#P$ query, \mathcal{L} belongs to $P^{\#P}$ and consequently to $PSPACE$. Therefore there is a polynomially-sized QBF formula T with $2n$ free variables whose solution space is \mathcal{L} .

Finally, recall that φ_1 and φ_2 have the same number of solutions. For every $\vec{a} \in \{0, 1\}^n$, there is therefore exactly one $\vec{b} \in \{0, 1\}^n$ such that $(\vec{a}, \vec{b}) \in \mathcal{L}$. Thus T describes a bijection and so T is indeed a transformation. Together with the fact that, by definition $\varphi_1(\vec{a}) = \varphi_2(\vec{b})$ for every $(\vec{a}, \vec{b}) \in \mathcal{L}$, we have that that $T(\varphi_1) \equiv \varphi_2$. ◀

The transformation T obtained by Theorem 10 may have arbitrarily nested quantifiers. A natural question to ask is whether it is possible to generalize Theorem 10 while limiting the number of alternating quantifiers. This leads us to the following conjecture, which restricts the number of quantifier alternations to some $k \geq 1$.

► **Conjecture 1** (Transformation Conjecture at k). *There is an integer $k \geq 1$ and a polynomial $p : \mathbb{N}^3 \rightarrow \mathbb{N}$ such that, if φ_1 and φ_2 are two Boolean formulas with n variables each and the same number of solutions, then there is a transformation $T \in \Sigma_k^P$ of size no more than $p(n, |\varphi_1|, |\varphi_2|)$ such that $T(\varphi_1) \equiv \varphi_2$.*

As we next show via the following two lemmas, our conjecture is equivalent to open unsolved questions in computational complexity.

We first generalize our proof of Theorem 10 to the restricted setting of the conjecture. In order to obtain Σ_k^P transformations, our proof requires the stronger, open assumption that the polynomial hierarchy collapses at or before Σ_k^P (in place of the fact used in Theorem 10 that $P^{\#P} \subseteq PSPACE$). We state this result as the following lemma.

► **Lemma 11.** *For all $k \geq 1$, if $P^{\#P} \subseteq \Sigma_k^P$ then the Transformation Conjecture at k holds.*

Proof. Let Boolean formulas φ_1 and φ_2 be Boolean formulas over n variables each, with the same number of solutions. Consider the language $\mathcal{L} \subseteq \{0, 1\}^n \times \{0, 1\}^n$ defined in the proof of Theorem 10. In particular, if $P^{\#P} \subseteq \Sigma_k^P$ then \mathcal{L} belongs to Σ_k^P . It follows that there is a polynomially-sized Σ_k^P formula T with $2n$ free variables whose solution space is \mathcal{L} . Hence, as in the proof of Theorem 10, T is a transformation and $T(\varphi_1) \equiv \varphi_2$. ◀

We next show in Lemma 12 that on the other hand the Transformation Conjecture implies that the polynomial hierarchy collapses at or before the level Σ_{k+4}^P . Tightening the result to prove the exact converse of Lemma 11, which would be that the Transformation Conjecture implies the collapse of the polynomial hierarchy at or before Σ_k^P , remains for future work.

► **Lemma 12.** *For all $k \geq 1$, if the Transformation Conjecture at k holds then $P^{\#P} \subseteq \Sigma_{k+4}^P$.*

Proof. Let $p : \mathbb{N}^3 \rightarrow \mathbb{N}$ be the polynomial from the conjecture. It suffices to prove that, given a Boolean formula φ over n variables and an index $1 \leq i \leq n + 1$, we can construct (in polynomial time) a Σ_{k+4}^P Turing Machine M that takes φ and i as input and accepts if and only if the i -th bit of $\#\varphi$ is 1. This decision problem is complete for $P^{\#P}$.

Our Turing Machine M first guesses a formula $T \in \Sigma_k^P$ (in prenex normal form) over $2n$ variables and an integer $0 \leq c \leq 2^n$ and makes a sequence of Π_{k+3}^P queries to verify that: (1) T is a transformation, (2) T transforms φ into a block with c solutions, and (3) the i -th bit of c is 1. M accepts if and only if these three conditions hold for some guess T and c , where T has size no more than $p(n, |\varphi|, |block_c|)$ and $0 \leq c \leq 2^n$.

Intuitively, c is our verified guess of the number of solutions for φ , so that M can check if the i -th bit of $\#\varphi$ is 1 just by consulting c . The transformation T is used to verify c .

The first property (that T is a transformation) can be verified by making a Π_{k+3}^P query followed by a Π_{k+1}^P query:

$$\begin{aligned} \alpha(T) &\equiv \forall \vec{x} \exists \vec{y} \forall \vec{z} (T(\vec{x}, \vec{z}) \leftrightarrow (\vec{y} = \vec{z})) \\ \beta(T) &\equiv \forall \vec{y} \exists \vec{x} (T(\vec{x}, \vec{y})). \end{aligned}$$

In particular, $\alpha(T)$ evaluates to true if and only if T describes some function g , and $\beta(T)$ then evaluates to true if and only if g is invertible. Thus $\alpha(T)$ and $\beta(T)$ together hold if and only if T describes a bijection g , i.e. if and only if T is a transformation.

The second property (that T transforms φ into a block with c solutions) can be verified by a single Π_{k+1}^P query:

$$\gamma(T, c) \equiv \forall \vec{x} \exists \vec{y} (T(\vec{x}, \vec{y}) \wedge (\varphi(\vec{x}) \leftrightarrow block_c(\vec{y})))$$

Recall from Section 2 that, for $0 \leq c \leq 2^n$, $block_c(\vec{x}) \equiv (\vec{x} \leq bin(c)) \wedge (\vec{x} \neq bin(c))$ is the Boolean formula whose solution space is a block with c solutions.

Finally, the third property (that the i -th bit of c is 1) can be verified simply by reading the bits of c .

Since M makes a single polynomially-sized guess (of T and c) followed by three Π_k^P queries, M is indeed a Σ_{k+4}^P Turing Machine. Since transformations preserve the number of solutions and $\#block_c = c$, then if M accepts it means that φ must have c solutions. Moreover, if M accepts then the i -th bit of c is 1. Thus the i -th bit of $\#\varphi$ is indeed 1. Conversely, consider the case where the i -th bit of $\#\varphi$ is 1. By the Transformation Conjecture there exists a polynomially-sized transformation $T' \in \Sigma_k^P$ that transforms φ into $block_{\#\varphi}$. Thus M will accept with $T = T'$ and $c = \#\varphi$. ◀

5 Transformations to blocks

In this section we give examples of how to use the transformations definitions and properties defined in Section 3 to construct and combine various transformations in order to manipulate the solution space to a specific structure. For that, we choose the structure of a block solution space and we focus on a specific type of transformations that transform a given formula into a block.

In general, Boolean formulas with a block solution space, such as *block_c* or chain formulas [9] have efficient counting by a simple binary-search method. To see this, assume that φ is a formula with n variables and a block solution space. Then for every assignment $\vec{\sigma}$ to the variables of φ , we have that $\varphi(\vec{\sigma}) = 1$ if and only if $\#\varphi \geq |\vec{\sigma}|$. Therefore $\#\varphi$ can be found by at most n such queries. Thus we have that a hypothetical efficient transformation of a given formula to a block can lead to efficient counting.

Moreover, in the setting of transformations, the following claim, which follows directly from the transformation properties discussed in Section 3, shows that by exploring transformations to blocks we can also get a better understanding on transformations between every two formulas.

▷ **Claim 13.** Let φ_1, φ_2 be Boolean formulas with the same number of solutions, and with transformations T_1, T_2 respectively to blocks. Then $T_2^{-1} \circ T_1(\varphi_1) \equiv \varphi_2$.

Proof. Assume that $\#\varphi_1 = \#\varphi_2 = c$. Then the transformations T_1, T_2 transform φ_1 and φ_2 respectively to a block of c solutions. Then $T_1(\varphi_1) \equiv T_2(\varphi_2) \equiv \text{block}_c(\vec{x})$. Then from the transformation properties we have that $T_2^{-1}(\text{block}_c(\vec{x})) \equiv \varphi_2$, thus $T_2^{-1} \circ T_1(\varphi_1) \equiv \varphi_2$. ◁

We first describe a type of transformations, possibly of exponential size to the size of the input, that can block any Boolean formula. These transformations are based on merging the solutions in the solution space together. We then explore a different technique, more efficient size-wise, of transformations, called *merge-rotate* that merges subspaces, that are already blocks, into a single block. We show how by iterating the merge-rotate transformations we can transform more sophisticated solution spaces to a block. Finally we demonstrate the use of the iterative approach to efficiently transform a specific type of formulas that are conjunction of two variable-disjoint sub-formulas, once their transformations to blocks are found. The transformations that we describe here are “parameterized” in the sense that the transformations use additional parameters that are depended on the given input formula. Exploring so called “oblivious” transformations that do not have such parameters is left for future work.

5.1 Transforming general formulas to blocks

A general description of a solution space S for every Boolean formula φ is as a sequence of alternating intervals of all solutions and all non-solutions. That is, $S = (1^{k_1}0^{k_2} \dots 1^{k_{\ell-1}}0^{k_\ell})$ where $0 \leq k_i \leq 2^n$ for every $i \leq \ell$ for some even ℓ , and $\sum_i k_i = 2^n$. In this section, we show a general Σ_1^P transformation, of size polynomial in $\max\{\ell, n\}$, that blocks S .

For that, we first describe *addition* as a way to “shift” whole intervals in a solution space. Let $\psi_{+,i}(\vec{y}, \vec{a}, \vec{b})$ be the following Σ_1^P formula:

$$\exists z_1 \dots \exists z_i \left(\neg z_1 \wedge \bigwedge_{j=1}^i (y_j \leftrightarrow a_j \oplus b_j \oplus z_j) \wedge \bigwedge_{j=1}^{i-1} (z_{j+1} \leftrightarrow ((z_j \wedge a_j) \vee (z_j \wedge b_j) \vee (a_j \wedge b_j))) \right)$$

The z variables represent the carry in the addition of the \vec{a} and \vec{b} variables. Recall from Section 2 that $|\cdot| : \{0, 1\}^n \rightarrow \{0, 1, \dots, 2^n - 1\}$ produces the n -bit positive integer corresponding to an assignment. Then we have the following.

▷ **Claim 14.** For all integers $0 < i \leq n$ and assignments $\vec{y}, \vec{a}, \vec{b} \in \{0, 1\}^n$, $\psi_{+,i}(\vec{y}, \vec{a}, \vec{b}) = \text{true}$ if and only if $|\vec{y}| = |\vec{a}| + |\vec{b}| \pmod{2^i}$.

Now let $interval_{(c,d)}(\varphi(x)) \equiv (bin(c) \leq_{lex} \vec{x} <_{lex} bin(d))$ be the formula that is true for every assignment $\vec{\sigma}$ over the n variables for which $|\vec{\sigma}| \in [c, d)$. Denote by $k'_i = \sum_{h \leq i} k_h$ the last index of the i 'th interval and set $k'_0 = 0$. Then let $Merge_{(k_1, \dots, k_\ell)}(\vec{x}, \vec{y})$ be the following transformation:

$$Merge_{(k_1, \dots, k_\ell)}(\vec{x}, \vec{y}) = \bigwedge_{j=0}^{\ell/2-1} \left(interval_{(k'_{2j}, k'_{2j+1})}(\vec{x}) \rightarrow \psi_{+,n}(\vec{y}, \vec{x}, 2^n - \sum_{1 \leq h \leq j} k_{2h}) \wedge \right. \\ \left. interval_{(k'_{2j+1}, k'_{2j+2})}(\vec{x}) \rightarrow \psi_{+,n}(\vec{y}, \vec{x}, \sum_{j < h < \ell/2} k_{2h+1}) \right)$$

Note that $Merge_{(k_1, \dots, k_\ell)}(\vec{x}, \vec{y})$ is of size polynomial in $\max\{\ell, n\}$.

▷ **Claim 15.** Let φ be a Boolean formula with a solution space described as $S = (1^{k_1} 0^{k_2} \dots 1^{k_{\ell-1}} 0^{k_\ell})$ where $0 \leq k_i \leq 2^n$ for all $0 \leq i \leq \ell$ and $\sum_i k_i = 2^n$. Let $k = \sum_{i=0}^{\ell/2-1} k_{2i+1}$. Then $Merge_{(k_1, \dots, k_\ell)}$ is a Σ_1^P transformation that transforms φ to the block $(1^k 0^{2^n-k})$.

Proof. The transformation $Merge_{(k_1, \dots, k_\ell)}$ simply shifts the j -th odd interval (which are all 1) to be the j -th interval in the lexicographic order by shifting the interval past all earlier 0 blocks, and the j -th even interval (which are all 0) to be the $\ell/2 + j$ -th interval in the lexicographic order by shifting the interval past all later 1 blocks. Thus all 0 blocks occur lexicographically after all 1 blocks following the transformation. ◁

When ℓ is exponential in n , the size of $Merge_{(k_1, \dots, k_\ell)}$ is exponential in n as well. In Sections 5.2 and 5.3 we explore ways to maintain efficient size transformations for certain solution spaces with an exponential number of intervals.

5.2 The merge-rotate transformation

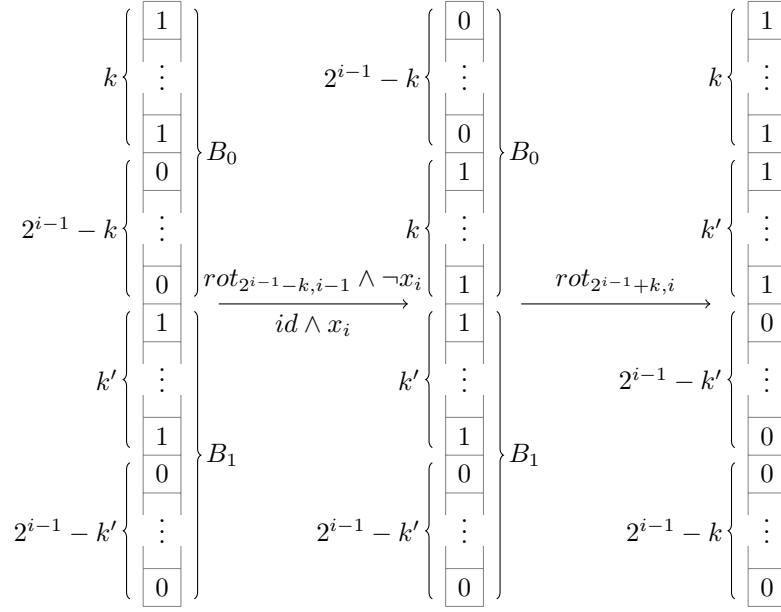
We next turn our attention to a different technique of transformations to blocks called the *merge-rotate* transformation. For merge-rotate we assume that a given solution subspace B is “halved” into an “upper” subspace B_0 and a “lower” subspace B_1 that are already in block forms. The transformation merge-rotate (as its name implies) rotates B_0 in order to merge its solutions with B_1 , then rotates the entire subspace B to turn B to a block. We describe the merge-rotate technique, then see how to extend merge-rotate to an iterative process that can handle more complicated solution spaces.

We start from the ψ_+ formula, defined in Section 5.1, upon which we define the following \mathbf{rot}_c transformation for a given integer $0 \leq c < 2^n$.

► **Definition 16.** For given $0 \leq c < 2^n$ and $0 < i \leq n$, let $\mathbf{rot}_{c,i}(\vec{x}, \vec{y}) = \psi_{+,i}(\vec{y}, \vec{x}, bin(\vec{c})) \wedge \bigwedge_{j=i+1}^n (x_j \leftrightarrow y_j)$.

By applying $\mathbf{rot}_{c,i}$ to a Boolean formula φ we “rotate” each subspace in $S[x_n, \dots, x_{i+1}]$ of size 2^i of φ by c steps (mod 2^i).

Next let $\sigma_n, \dots, \sigma_{i+1}$ be an assignment to x_n, \dots, x_{i+1} and assume that the subspaces $B_0 = S[\sigma_n, \dots, \sigma_{i+1}, 0]$ and $B_1 = S[\sigma_n, \dots, \sigma_{i+1}, 1]$ are already in block forms, with number of solutions k and k' respectively. The overall subspace $B = S[\sigma_n, \dots, \sigma_{i+1}]$ has the form $(1^k 0^{2^{i-1}-k} 1^{k'} 0^{2^{i-1}-k'})$. We show how to use the rotation transformation on these blocks, to transform the subspace B a block of the form $(1^{k+k'} 0^{2^i-(k+k')})$. For that, we need to restrict the rotation only to B_0 in order to merge the solutions of B_0 and B_1 . We then use rotation on the entire subspace B to rotate B to a block form.



■ **Figure 2** The transformation to blocks *MergeRotate* is a composition of two *rot* transformations. Each solution subspace of the form $B = (1^k 0^{2^{i-1}-k} 1^{k'} 0^{2^{i-1}-k'})$ is transformed on B_0 by $rot_{2^{i-1}-k, i-1}$ and on B_1 by the identity transformation to $(0^{2^{i-1}-k} 1^{k+k'} 0^{2^{i-1}-k'})$ and then by $rot_{2^{i-1}+k, i}$ to $(1^{k+k'} 0^{2^{i-1}-(k+k')})$.

This results in the following transformation which we call *MergeRotate*, also depicted in Figure 2. Note that *id* is the identity transformation as defined in Section 3.

► **Definition 17.** Let $k, i \leq n$ be given. The transformation *MergeRotate* is defined as:

$$\text{MergeRotate}(k, i) \equiv rot_{2^{i-1}+k, i} \circ ((rot_{2^{i-1}-k, i-1} \wedge \neg x_i) \vee (id \wedge x_i))$$

Then we have the following claim, whose proof follows from the definition of *MergeRotate*.

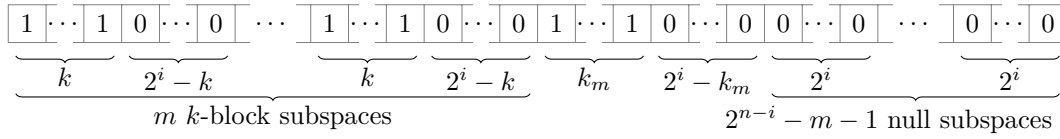
▷ **Claim 18.** Let $B_0 = S[\sigma_n, \dots, \sigma_{i+1}, 0]$ and $B_1 = S[\sigma_n, \dots, \sigma_{i+1}, 1]$ for some index $i \leq n$ and $(\sigma_n, \dots, \sigma_{i+1}) \in \{0, 1\}^{n-i}$. Assume that B_0 and B_1 are blocks of size k and k' respectively. Then the transformation *MergeRotate*(k, i) transforms $B = S[\sigma_n, \dots, \sigma_{i+1}]$ to a block of $k + k'$ solutions.

Note that k' , the number of solutions in B_1 , is not required for *MergeRotate*. Also note that *MergeRotate* is in *NP* and in size polynomial to n .

We next give a technical lemma, which we make use of in Section 5.3, that shows that when using *MergeRotate* we do not always need to have the rotation as the exact size of the block of B_0 , as long as B_1 is a null solution space.

► **Lemma 19.** Let $B_0 = S[\sigma_n, \dots, \sigma_{i+1}, 0]$ and $B_1 = S[\sigma_n, \dots, \sigma_{i+1}, 1]$ for some index $i \leq n$ and $(\sigma_n, \dots, \sigma_{i+1}) \in \{0, 1\}^{n-i}$. Assume that B_0 is a block of size k' and that B_1 is a null solution space (note that it means that the overall subspace $B = (1^{k'} 0^{2^{i-1}-k'} 0^{2^{i-1}})$ is already a block of k' solutions). Then for every $k' \leq k \leq 2^{i-1}$ the transformation *MergeRotate*(k, i) maintains $B = S[\sigma_n, \dots, \sigma_{i+1}]$ as a block of k' solutions.

Proof. Note that when applying *MergeRotate*, we first rotate only B_0 by $2^{i-1} - k$. This results in a space $B' = (0^{2^{i-1}-k} 1^{k'} 0^{k-k'} 0^{2^{i-1}})$. Now the second rotation rotates B' by $2^{i-1} + k$ which makes $B'' = (1^{k'} 0^{k-k'} 0^{2^{i-1}} 0^{2^{i-1}-k}) = (1^{k'} 0^{2^i-k'})$ as required. ◀



■ **Figure 3** The truth table of a k -block i -suffix-null solution space, given in lexicographic order from left to right as described in Definition 20.

In the next sections we show where *MergeRotate* can be used iteratively to construct transformations of polynomial size to blocks for formulas with a specific solution space structure.

5.3 Iterating the merge-rotate transformations

Having defined the merge-rotate transformation, we would like to see how to use it to transform more complex solution spaces, with possibly an exponential number of intervals, into blocks. For that, a natural solution space that can demonstrate the iterative use of merge-rotate is a solution space S where, for some integer $0 \leq k \leq 2^i$ and every $(\sigma_n, \dots, \sigma_{i+1}) \in \{0, 1\}^{n-1}$, the subspace $S[x_n, \dots, x_{i+1}]$ is a block of size k . In fact, we can make a somewhat stronger statement on a more complicated solution space structure as defined below. This structure also appears later in Section 5.4.

► **Definition 20.** For a given $i \leq n$, and $0 \leq k \leq 2^i$, a solution space is said to be a k -block i -suffix-null if there exists an integer $0 \leq m \leq 2^{n-i}$ such that: (i) every solution space $S[\sigma_n, \dots, \sigma_{i+1}]$ where $(\sigma_n, \dots, \sigma_{i+1}) >_{lex} bin(m)$ is a null solution space; (ii) every solution space $S[\sigma_n, \dots, \sigma_{i+1}]$ where $(\sigma_n, \dots, \sigma_{i+1}) <_{lex} bin(m)$ is a block of size k ; (iii) $S[bin(m)]$ is a block of size $0 \leq k_m \leq k$.

Figure 3 depicts a k -block i -suffix-null solution space. Note that m can be 0 which means that the entire solution space is null, or can be 2^{n-i} in which all that the elements in $S[x_n, \dots, x_{i+1}]$ are k -blocks. Moreover, if $i = n$ then S is a block of size k .

We next show how to construct a transformation composed of $n - i$ merge-rotate transformations in order to block a k -block i -suffix-null solution space. For given $i < n$ and $k < 2^i$, let $ItrMergeRotate(k, i)$ be the following transformation:

$$MergeRotate(2^{n-i-1}k, n) \circ \dots \circ MergeRotate(2^{j-1}k, i + j) \circ \dots \circ MergeRotate(k, i + 1)$$

We then have the following.

► **Theorem 21.** For a given $i < n$ and $k < 2^i$, let S be a k -block i -suffix-null solution space. Then $ItrMergeRotate(k, i)$ transforms S into a block.

Proof. We prove by induction that for every $0 \leq j \leq n - i$, the solution space S after applying the transformation $MergeRotate(2^{j-1}k, i + j)$, is a $(2^j k)$ -block $(i + j)$ -suffix-null. It follows after applying $MergeRotate(2^{j-1}k, i + j)$ for $j = n - i$ that S is an ℓ -block n -suffix-null (for some $\ell \leq 2^{n-i}k$), i.e. S is a block.

In the base case $j = 0$ (i.e. before applying the first *MergeRotate*), S is by hypothesis a k -block i -suffix-null solution space. Assume by induction that, for some $0 \leq j \leq n - i - 1$ when applying $ItrMergeRotate(k, i)$ on S , after $MergeRotate(2^{j-1}k, i + j)$ we have that S is a $(2^j k)$ -block $(i + j)$ -suffix-null solution space. Then by definition, there is some m for

which the subspace $S[\sigma_n, \dots, \sigma_{i+j+1}]$ is a null block for every $(\sigma_n, \dots, \sigma_{i+j+1}) >_{lex} bin(m)$, a block of size $2^j k$ for every $(\sigma_n, \dots, \sigma_{i+j+1}) <_{lex} bin(m)$ and $S[bin(m)]$ is a k_m block for some $k_m \leq 2^j k$.

Now the transformation $MergeRotate(2^j k, i+j+1)$ is applied on S , as described in Definition 17, by merging and rotating the subspaces $S[\sigma_n, \dots, \sigma_{i+j+2}, 0]$ and $S[\sigma_n, \dots, \sigma_{i+j+2}, 1]$ for every $(\sigma_n, \dots, \sigma_{i+j+2}) \in \{0, 1\}^{n-i-j-1}$. This makes four cases to consider:

1. $(\sigma_n, \dots, \sigma_{i+j+2}, 1) < bin(m)$. Then both $S[\sigma_n, \dots, \sigma_{i+j+2}, 0]$ and $S[\sigma_n, \dots, \sigma_{n-i+j+2}, 1]$ are blocks of size $2^j k$, and therefore by Claim 18, applying $MergeRotate(2^j k, i+j+1)$ transforms $S[\sigma_n, \dots, \sigma_{i+j+2}]$ to a block of size $2^{j+1} k$.
2. $(\sigma_n, \dots, \sigma_{i+j+2}, 0) > bin(m)$. Then both $S[\sigma_n, \dots, \sigma_{i+j+2}, 0]$ and $S[\sigma_n, \dots, \sigma_{n-i+j+2}, 1]$ are null, and therefore applying $MergeRotate(2^j k, i+j+1)$ maintains $S[\sigma_n, \dots, \sigma_{i+j+2}]$ null as well.
3. $(\sigma_n, \dots, \sigma_{i+j+2}, 1) = bin(m)$. Then the subspace $S[\sigma_n, \dots, \sigma_{i+j+2}, 0]$ is a block of size $2^j k$, while the subspace $S[\sigma_n, \dots, \sigma_{i+j+2}, 1]$ is a block of size $k_m < 2^j k$. Then again by Claim 18, applying $MergeRotate(2^j k, i+j+1)$ transforms $S[\sigma_n, \dots, \sigma_{i+j+2}]$ to a block of size $2^j k + k_m$, where $2^j k + k_m \leq 2^{j+1} k$.
4. $(\sigma_n, \dots, \sigma_{i+j+2}, 0) = bin(m)$. Then the subspace $S[\sigma_n, \dots, \sigma_{i+j+2}, 0]$ is a block of size k_m , while the subspace $S[\sigma_n, \dots, \sigma_{i+j+2}, 1]$ is a null block. Since $k_m \leq 2^j k$ this case fits to the conditions of Lemma 19. Therefore we get that $S[\sigma_n, \dots, \sigma_{i+j+2}]$ is (still) a block of size $k_m \leq 2^{j+1} k$.

That shows that applying $MergeRotate(2^j k, i+j+1)$ on S in the j 'th iteration of $ItrMergeRotate(k, i)$ transforms S to a $2^{j+1} k$ -block $(i+j+1)$ -suffix-null solution space as required. ◀

In the next section we see how to make use of Theorem 21 when blocking specific conjuncted Boolean formulas.

5.4 Transforming conjuncted variable-disjoint formulas

Having defined the iterated merge-rotate method, we finally demonstrate how to combine it with existing transformations to blocks, in the specific formulation which we now describe.

► **Theorem 22.** *Let φ be a Boolean formula such that $\varphi = \varphi_1 \wedge \varphi_2$ where φ_1 and φ_2 have disjoint variables. Furthermore, assume that T_1 and T_2 are Σ_1^P transformations that transform φ_1 and φ_2 respectively to blocks. Then there is a Σ_1^P transformation of size polynomial in $|T_1| + |T_2| + |ItrMergeRotate|$ that transforms φ to a block.*

Proof. We denote the φ_1 variables by $x_n \dots x_{i+1}$ for some i and the φ_2 variables by $x_i \dots, x_1$. We set an order on x_n, \dots, x_1 where x_n is the msb. Denote the number of solutions in φ_1 by k_1 and the number of solutions of φ_2 by k_2 (we can obtain the k_i 's by performing the transformations to blocks on φ_i 's and use binary search.). Note that every solution subspace $S[\sigma_n, \dots, \sigma_{i+1}]$ of $S[x_n, \dots, x_{i+1}]$ is either null (when $\varphi_1(\sigma_n, \dots, \sigma_{i+1}) = 0$) or is an identical copy of the solution space of φ_2 (when $\varphi_1(\sigma_n, \dots, \sigma_{i+1}) = 1$).

We first apply $T_2' = T_2 \wedge \bigwedge_{j>i} (x_j \leftrightarrow y_j)$ on φ . This effectively applies T_2 to every copy of the solution space of φ_2 and so transforms every subspace of $S[x_n, \dots, x_{i+1}]$ that is not null to a block of size k_2 . Next, we apply $T_1' = T_1 \wedge \bigwedge_{j \leq i} (x_j \leftrightarrow y_j)$ to $T_2(\varphi)$. This transforms the solution space S to a k_2 -block i -suffix-null. Finally we make use of Theorem 21 and apply $ItrMergeRotate(k_2, i)$ to transform S to a block. Thus the resulting composition $ItrMergeRotate(k_2, i) \circ T_1' \circ T_2'$ is a transformation that transforms φ to a block. Moreover, this composition is in Σ_1^P since all components are in Σ_1^P . ◀

Theorem 22 shows that there are cases in which a divide-and-conquer approach, in the sense of a syntactical decomposition of a Boolean formula into separate conjuncts, followed by pursuing a transformation for each sub formula separately to a block, can lead to an efficient transformation for the original formula to a block. This approach also follows recent methods in decomposition of Boolean formulas, see for example [10].

6 Conclusion

The transformations that we explored in this work transform Boolean functions, described by Boolean formulas, while maintaining the number of solutions. Manipulations of the structure of the solution space through Boolean formulas were done before on various occasions. One classical example in the theoretical setting is Sipser’s proof of $BPP \subseteq \Sigma_2^P \cap \Pi_2^P$ [3] that makes use of what we call the XOR transformation. In a more applied setting, the SAT community uses transformations in an ad-hoc manner as preprocessing steps; although some preprocessing techniques change the number of solutions, many techniques do not [4]. In addition, in the area of approximate model counting, two very recent works [2, 7] suggest the use of transformations to create a high degree of separation (in terms of Hamming distance) between solutions in the solution space in order to improve practical approximate counting. It would be interesting in future work to see how our work on transformations can be applied with this goal. Finally, it is also worth mentioning a similar line of work that studies the Formula Isomorphism Problem, which asks if there exists a bijection between variables such that two Boolean formulas become equivalent [1, 6, 18].

To the best of our knowledge, this work is the first that formally defines and studies the general concept of transformations of Boolean functions described as Boolean formulas. Among the results that we presented here are not only takeaways on the computational complexity limitations of using transformations, but also definitions, properties, and foundational techniques that express the versatility and the usability in which transformations can be used to combinatorially manipulate various solution spaces.

References

- 1 Manindra Agrawal and Thomas Thierauf. The formula isomorphism problem. *SIAM Journal on Computing*, 30(3):990–1009, 2000.
- 2 S. Akshay and Kuldeep S. Meel. Scalable Approximate Model Counting via Concentrated Hashing. Under submission.
- 3 Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.
- 4 Armin Biere. Preprocessing and Inprocessing Techniques in SAT. In *Proceedings of HVC*, page 1, 2011. doi:10.1007/978-3-642-34188-5_1.
- 5 Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS Press, 2009.
- 6 Elmar Böhler, Edith Hemaspaandra, Steffen Reith, and Heribert Vollmer. Equivalence and isomorphism for Boolean constraint satisfaction. In *International Workshop on Computer Science Logic*, pages 412–426. Springer, 2002.
- 7 Michele Boreale and Daniele Gorla. Approximate Model Counting, Sparse XOR Constraints and Minimum Distance, 2019. arXiv:1907.05121.
- 8 Randal E Bryant. Graph-based algorithms for boolean function manipulation. Technical report, Carnegie-Mellon University, 2001.
- 9 Supratik Chakraborty, Dror Fried, Kuldeep S Meel, and Moshe Y Vardi. From weighted to unweighted model counting. In *Proceedings of IJCAI*, 2015.

39:14 Transformations of Boolean Functions

- 10 Supratik Chakraborty, Dror Fried, Lucas M. Tabajara, and Moshe Y. Vardi. Functional Synthesis via Input-Output Separation. In *Proceedings of FMCAD*, pages 1–9, 2018.
- 11 Yves Crama and Peter L Hammer. *Boolean functions: Theory, algorithms, and applications*. Cambridge University Press, 2011.
- 12 Carmel Domshlak and Jörg Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *J. of AI Research*, 30:565–620, 2007.
- 13 Lance Fortnow. Counting complexity. *Complexity theory retrospective II*, pages 81–107, 1997.
- 14 Dror Fried, Axel Legay, Joël Ouaknine, and Moshe Y. Vardi. Sequential Relational Decomposition. In *Proceedings of LICS*, pages 432–441, 2018.
- 15 Maurice Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 72(5):593–599, 1953.
- 16 Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan Marcu, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. *AI Magazine*, 28(3):13–13, 2007.
- 17 Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In *Proceedings of IJCAI*, pages 3141–3148, 2015.
- 18 B.V. Rao and M.N. Sarma. Isomorphism testing of read-once functions and polynomials. In *Proceedings of FSTTCS*, 2011.
- 19 Tian Sang, Fahiem Bacchus, Paul Beame, Henry A Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Proceedings of SAT*, pages 20–28, 2004.
- 20 Larry Stockmeyer. The complexity of approximate counting. In *Proceedings of STOC*, pages 118–126. ACM, 1983.
- 21 Leslie G Valiant. The complexity of enumeration and reliability problems. *SIAM J. on Computing*, 8(3):410–421, 1979.
- 22 Ingo Wegener. *The complexity of Boolean functions*. BG Teubner, 1987.