UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCAS MARTINELLI TABAJARA

Synthesis of Boolean functions through Binary Decision Diagrams

Monograph presented in partial fulfillment
of the requirements for the degree of
Bachelor of Computer Science

Prof. Luís da Cunha Lamb
Advisor

Prof. Moshe Y. Vardi
Coadvisor

Porto Alegre, June 18th, 2015

CIP – CATALOGING-IN-PUBLICATION

Lucas Martinelli Tabajara,

Synthesis of Boolean functions through Binary Decision Diagrams /

Lucas Martinelli Tabajara. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2015.

66 f.: il.

Monograph – Universidade Federal do Rio Grande do Sul. Curso de Bacharelado em Ciência da Computação, Porto Alegre, BR–RS, 2015. Advisor: Luís da Cunha Lamb; Coadvisor: Moshe Y. Vardi.

1. Boolean functions. 2. Synthesis. 3. Binary Decision Diagrams. 4. Quantified Boolean Formulas. I. da Cunha Lamb, Luís. II. Y. Vardi, Moshe. III. Título.

ACKNOWLEDGMENTS

I would like to thank professors Luís Lamb and Moshe Vardi, my advisors on both sides of the globe, for their guidance and support during my academic journey.

I also extend my sincere thanks to Dror Fried, for his active participation in this work, our many discussions and his invaluable comments and suggestions.

I would also like to thank my girlfriend Laura, for cheering me on and being there for me, celebrating my accomplishments and helping me relax in times of stress.

Lastly, I would like to thank my family for helping and supporting me through all my endeavors and giving me the certainty that I have people I can count on no matter what.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ABSTRACT

Boolean functions are an integral component of computer science, from combinational circuits to satisfiability problems. However, although several problems can be encoded as Boolean formulas, often the most intuitive representation is not constructive but declarative, meaning that instead of a Boolean function describing how to obtain the outputs from the inputs we have a relation between Boolean inputs and outputs that must be satisfied. Although this provides a specification of the problem, in order to actually implement a circuit or program to solve it we first need a constructive function that adheres to this specification. This function can be much more complex than the specification, requiring expending time and effort in development and testing to obtain a correct implementation. If we have methods to synthesize constructive implementations of Boolean functions from a declarative specification, we can automate this process and obtain correct-by-construction implementations with little human effort. In this work we propose methods to do this based on the manipulation of Binary Decision Diagrams (BDDs), a data structure for representing Boolean functions. Starting from a BDD representing the specification, we first describe how to verify that this specification defines a total function. If it is the case, we convert it into a sequence of BDDs, each representing a function that computes one of the outputs. This sequence can also be interpreted as a multi-rooted BDD, and can later be converted into a practical implementation as a circuit or program. We also tested our methods by synthesizing operations over integers in binary with varying number of bits, to observe how they scale and to obtain insights on the behavior of BDDs when used for synthesis.

Keywords: Boolean functions, Synthesis, Binary Decision Diagrams, Quantified Boolean Formulas.

Síntese de funções Booleanas através de Diagramas de Decisão Binários

RESUMO

Funções Booleanas são uma parte integral da ciência da computação, de circuitos combinacionais a problemas de satisfatibilidade. No entanto, apesar de diversos problemas poderem ser codificados como fórmulas Booleanas, frequentemente a representação mais intuitiva não é construtiva mas declarativa, significando que, ao invés de uma função Booleana descrevendo como obter as saídas a partir das entradas, nós temos uma relação entre entradas e saídas Booleanas que deve ser satisfeita. Apesar de tal formato fornecer uma especificação do problema, para de fato implementar um circuito ou programa para resolvê-lo precisamos primeiro de uma função construtiva que adere a essa especificação. Essa função pode ser muito mais complexa que a especificação, necessitando dispender de tempo e esforço no desenvolvimento e teste para obter uma implementação correta. Se possuirmos métodos para sintetizar implementações construtivas de funções Booleanas a partir de uma especificação declarativa, poderemos automatizar esse processo e obter implementações garantidamente corretas com mínimo esforço humano. Neste trabalho propomos métodos para realizar essa síntese baseados na manipulação de Diagramas de Decisão Binários (BDDs), uma estrutura de dados para a representação de funções Booleanas. A partir de um BDD representando a especificação, primeiro descrevemos como verificar que tal especificação define uma função total. Se é o caso, a convertemos em uma sequência de BDDs, cada um representando uma função que computa uma das saídas. Essa sequência pode também ser interpretada como um BDD de múltiplas raízes, e pode depois ser convertida em uma implementação prática na forma de um circuito ou programa. Também testamos nossos métodos sintetizando operações sobre inteiros em binário variando o número de bits, para observar sua escalabilidade e obter uma intuição sobre o comportamento de BDDs quando usados para síntese.

Palavras-chave: Funções Booleanas, Síntese, Diagramas de Decisão Binários, Fórmulas Booleanas Quantificadas.

1 INTRODUCTION

Boolean functions stand at the core of computer science as one of its most important building blocks. From digital circuits to satisfiability problems, functions of Boolean variables appear in all levels of computing. In fact, even problems over different data types can be represented as Boolean formulas, from basic arithmetic operations implemented as logical circuits to different NP-complete problems encoded in SAT. However, there can be a large gap between specifying a problem as a Boolean formula and obtaining a function that solves the problem.

Often the most intuitive way of defining a Boolean function is not constructive, describing how the outputs can be computed from the inputs, but rather declarative, as a relation between input and output values that must be satisfied. Nevertheless, in order to implement a function in a practical format, such as in a circuit or program, a declarative definition is not enough, and a constructive description of how to compute the output from the input is necessary. The way to do this is not always clear, and often requires considering implementation details that are abstracted in the declarative definition. As such, this process can be much more time-consuming than simply specifying the behavior of the function as a relation. Furthermore, implementation mistakes can lead to a result that does not conform to the specification, requiring further effort to be applied in testing and debugging.

In order to address this problem, in this work we propose methods to automatically synthesize from a specification, given as a propositional formula relating inputs and outputs, a correct-by-construction implementation of the desired Boolean function. More formally, given a specification in the form of a characteristic function $f : \mathbb{B}^n \times \mathbb{B}^m \to \mathbb{B}$, where $f(\vec{y}, \vec{x}) = 1$ iff $\vec{x}$ is a correct output for the input $\vec{y}$, we synthesize an implementation $g : \mathbb{B}^n \to \mathbb{B}^m$ with the guarantee that $f(\vec{y}, g(\vec{y})) = 1$.

Our synthesis procedure is based on the manipulation of Binary Decision Diagrams (BDDs) (BRYANT, 1986), data structures designed for representing Boolean functions. BDDs provide easy-to-manipulate canonical representations of Boolean expressions in which Boolean operations can be implemented efficiently, and as such have found applications in a variety of problems, including model checking (BURCH et al., 1992) and satisfiability solving (PAN; VARDI, 2005). However, the size of the BDD for a certain formula is often difficult to predict, as BDDs can often blow up in size due to various factors (HU; DILL, 1993). Therefore, one of our objectives is to observe how BDD sizes behave in the synthesis process.

We begin the synthesis by converting the specification f into a BDD. We then apply transformations to it in order to obtain a representation of a correct implementation g as a multi-rooted BDD, where each root represents a different output. To do this, we first verify that the specification does describe a total function, that is, that every possible input has an associated output. If so, the synthesis itself is performed to obtain the implementation, for which we present two different approaches. The first is based on quantifier elimination, abstracting the representation of f as a BDD, while the second was designed specifically to be used with this data structure, but requires some restrictions to be applied to the BDD representation of f.

1.1 Related work

There have been a number of works pursuing the synthesis of Boolean functions from a declarative specification. These have employed different approaches and data structures, and sought applications in both hardware and software design.

The work of (KUNCAK et al., 2010) has as its focus the synthesis of small program fragments. It proposes a new programming language construct called *choose*, which would allow a programmer to specify the values of variables not directly through assignment, but by providing constraints on their value. This construct would then be compiled into code that chose appropriate values to the variables according to the constraints. The work concerns itself primarily with the synthesis of arithmetic expressions, but it briefly describes a method employing BDDs that can be used to apply the same concept to Boolean expressions. Our second method is reminiscent of the one mentioned in this work, following a similar intuition, but our approach was designed in order to take further advantage of the convenient properties of BDDs.

Still in the domain of software, (MARI et al., 2011) is more directly related to Boolean synthesis, aiming to synthesize implementations of controllers in the C programming language. This work also chooses BDDs as the underlying data structure, and for the synthesis it follows a procedure presented in (TRONCI, 1998), which is similar to our own method based on quantifier elimination. However, compared to our work it is more technical and focused on a C implementation, instead of our goal of synthesizing general Boolean functions.

Moving into hardware, (KUKULA; SHIPLE, 2000) proposes a direct mapping from a BDD representing a relation between inputs and outputs to a circuit implementing the behavior specified by the relation. Differently from the other approaches mentioned here and from our own, it converts the specification BDD directly to a circuit implementation, instead of first computing a BDD representation of the implementation. This circuit is also designed to be able to select over multiple possible output values through control signals. Because of this, the resulting circuit is very complex and large, and the solution is specific to a hardware implementation.

Finally, (JIANG, 2009) has a very similar goal to ours, being concerned with extracting functions from Boolean relations. It employs the method of quantifier elimination based on function composition presented in (JIANG; LIN; HUNG, 2009), effectively a process for finding witnesses to existentially-quantified Boolean formulas. This approach also has similarities to our own quantifier elimination method, but using a different method to obtain witnesses based on the concept of Craig interpolation. They also choose to use a different data structure as an alternative to BDDs, called And-Inverter

Graphs. The work of (HOFFEREK et al., 2013) generalizes this approach in order to synthesize several witnesses simultaneously. However, their approach is particularly focused on implementing Boolean control signals, rather than general functions, and the specifications used by them follow a different, specialized format reflecting this. They also dispense with data structures such as BDDs or AIGs to instead use interpolation on proofs obtained by specialized solvers.

In general, these works test their methods by applying them to the synthesis of specific benchmarks of the area. In contrast, the experimental portion of our work will instead focus on testing scalability, observing how the methods proposed here behave when the size of the specification grows. This way, we intend to obtain important general intuitions about the use of BDDs in the synthesis of Boolean functions.

1.2 Structure of this work

We start by providing basic terminology and theoretical concepts necessary for understanding the rest of this work, including an overview of BDDs, their properties and basic operations. This is presented in Chapter 2. Next, we formally describe the problem and introduce definitions specific to it in Chapter 3. The synthesis process itself and the different methods proposed for it are described in Chapter 4. Experiments and the results obtained from them are described in Chapter 5. Finally, conclusions and future work are discussed in Chapter 6.

2 CONCEPTUAL BASIS

In this chapter we introduce some important concepts in understanding our methodology, including an overview of Binary Decision Diagrams, which we use as our underlying data structure.

In what follows we denote by $\mathbb{B} = \{0, 1\}$ the set of Boolean values. The symbols $\neg$, $\wedge$, $\vee$ and $\oplus$ denote Boolean negation (NOT), conjunction (AND), disjunction (OR) and exclusive disjunction (XOR), respectively.

2.1 Shannon expansion

Let $f(x_1, \ldots, x_n)$ be a Boolean function of multiple variables. The positive and negative cofactors of f in respect to variable x_i, denoted respectively by f_{x_i} and $f_{\neg x_i}$, are defined as $f_{x_i}(x_1, \ldots, x_{i-1}, x_i, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 1, \ldots, x_n)$ and $f_{\neg x_i}(x_1, \ldots, x_{i-1}, x_i, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 0, \ldots, x_n)$. It is easy to verify that, for every variable x_i, the identity $f = (x_i \wedge f_{x_i}) \vee (\neg x_i \wedge f_{\neg x_i})$ holds. This decomposition of a Boolean function into positive and negative factors is called *Shannon expansion*.

In a way, Shannon expansion is representative of the concept of satisfiability of Boolean expressions. To obtain an assignment to a Boolean expression, each variable is selected to receive either a positive or negative assignment and then it is appropriately substituted by its assigned value in the expression. A formula is satisfiable if for each variable either a positive or negative assignment leads to it evaluating to 1.

As an example, take the function $f(x_1, x_2, x_3) = (x_1 \wedge \neg x_2) \vee x_3$. By expanding it in respect to the variable x_1 we obtain:

$$\begin{aligned}(x_1 \wedge \neg x_2) \vee x_3 &= (x_1 \wedge ((1 \wedge \neg x_2) \vee x_3)) \vee (\neg x_1 \wedge ((0 \wedge \neg x_2) \vee x_3)) \\ &= (x_1 \wedge (\neg x_2 \vee x_3)) \vee (\neg x_1 \wedge (0 \vee x_3)) \\ &= (x_1 \wedge (\neg x_2 \vee x_3)) \vee (\neg x_1 \wedge x_3)\end{aligned}$$

Note that the resulting formula is not only equivalent but is also in a format in which we can clearly see, given an assignment to x_1, what assignment to x_2 and x_3 must be picked to satisfy the formula.

2.2 Quantified Boolean Formulas

A Quantified Boolean Formula, or QBF, is a generalization of propositional logic where Boolean variables can be universally or existentially quantified. Although this allows for quantifiers inside Boolean expressions, it can be shown that every QBF formula can be represented in the form $Q_1x_1Q_2x_2 \dots Q_nx_n\varphi$, where $Q_1, Q_2, \dots, Q_n$ are quantifiers and φ is a formula in propositional logic. This form, where quantifiers are grouped in the outside of the formula, is called *Prenex normal form*. From now on, it will be assumed that any QBF formula is in Prenex normal form.

Generally, the term QBF denotes a formula with no free variables, i.e. where every variable is either universally or existentially quantified. Formulas that allow free variables are called *open* QBF (KLIEBER et al., 2013). In contrast, QBF with no free variables can be called *closed* QBF when it is necessary to differentiate them from the open version. While closed QBFs are always either true or false, open QBFs depend on an assignment to their free variables to be evaluated. Therefore they can be classified according to criteria like satisfiability and validity, the same way as propositional formulas.

2.2.1 Quantifier elimination

Resolution of QBF formulas is performed through quantifier elimination. Each step of quantifier elimination removes one variable from the formula, such that once this is repeated for all quantifiers only free variables remain. If the formula is closed (has no free variables), it is reduced to either 0 or 1, indicating the truth value of the QBF.

Because Boolean variables can only assume two values, 0 or 1, quantifier elimination for QBF is easier than for generic first-order logic formulas. $\forall x(f(x))$ is equivalent to $f(0) \wedge f(1)$, and $\exists x(f(x))$ is equivalent to $f(0) \vee f(1)$. Therefore, given a QBF $Q_1x_1Q_2x_2 \dots Q_nx_n(f(x_1, x_2, \dots, x_n))$ we can start with the innermost quantifier Q_n and eliminate it by performing $Q_1x_1 \dots Q_{n-1}x_{n-1}(f(x_1, \dots, x_{n-1}, 0) \diamond f(x_1, \dots, x_{n-1}, 1))$, where $\diamond$ is $\wedge$ if Q_n is $\forall$ and $\vee$ if Q_n is $\exists$. Then we can repeat the process for the rest of the quantifiers. Although this process is simple, it is still computationally demanding, since each eliminated quantifier doubles the size of the formula.

Notice the similarity between quantifier elimination, particularly for existential quantifiers, and Shannon expansion. The two branches produced are precisely the positive and negative cofactors of the formula. This happens because satisfiability of a propositional formula $f(x_1, \dots, x_n)$ is equivalent to the truth value of its existentially quantified version $\exists x_1 \dots \exists x_n f(x_1, \dots, x_n)$.

2.2.2 Witnesses

The following definition can be used for any first-order formula, but in this work we are concerned specifically with the case for QBF. Let φ be a formula of the form $\exists x(f(x))$. A witness for φ is a value v such that $f(v)$ is true. In other words, a witness is an example of a value that proves the satisfiability of an existential formula. This definition can be generalized by saying that for a formula φ of the form $\exists x_1, \dots, x_n(f(x_1, \dots, x_n))$, v_i is a witness for x_i in φ if $\exists x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n(f(x_1, \dots, x_{i-1}, v_i, x_{i+1}, \dots, x_n))$ is true. If a witness is dependent on outside variables, then we will say it is defined by a *witness function*.

2.3 Binary Decision Diagrams

A *Binary Decision Diagram*, or BDD, is a data structure for representing a Boolean function that follows from the idea of Shannon expansion (BRYANT, 1986). A BDD takes the form of a directed acyclic graph where each internal node represents a variable, and following a path from the root to the leaves represents evaluating the function for a specific assignment.

The BDDs employed in this work are more precisely Reduced Ordered Binary Decision Diagrams (ROBDDs), which is the variety most commonly used in the literature. However, for simplicity we will refer to ROBDDs simply as BDDs throughout this work. The properties of reduction and ordering will be discussed later in this section.

To understand BDDs, it is useful to start from a related data structure called a *binary decision tree*, also used to encode a Boolean function, and apply some modifications to convert it into a BDD. The binary decision tree T representing a given Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ is defined inductively as follows:

1. If $n = 0$, f must be a Boolean constant $b \in \mathbb{B}$. Then T is a single leaf node labeled by the constant b, called a *terminal node*.

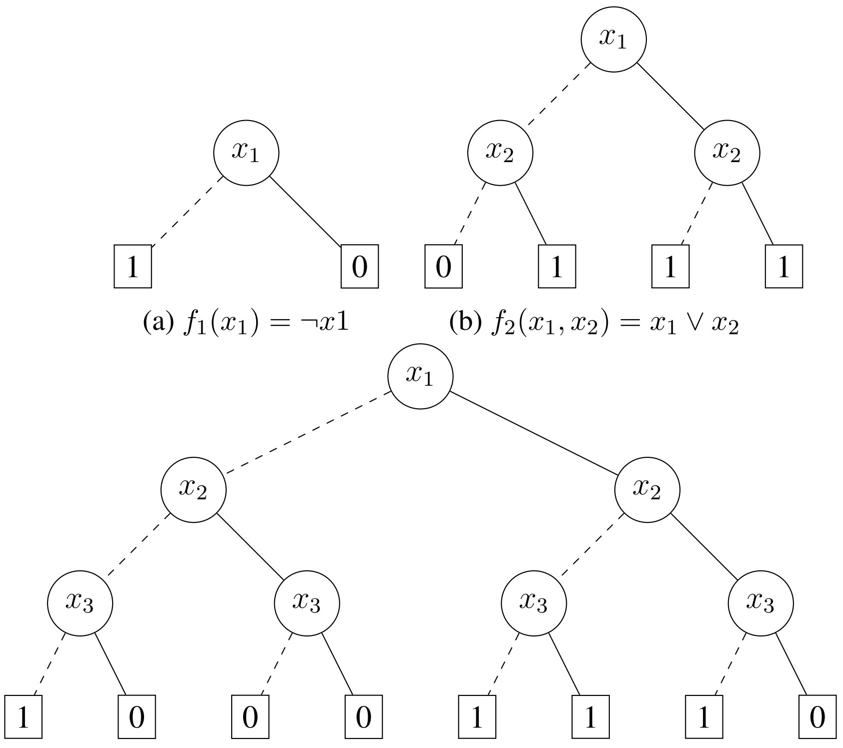
2. Otherwise, if $n > 0$, T is a tree rooted in a node labeled by the first variable x_1 with two outgoing edges labeled respectively by 0 and 1. Let T_b be the sub-tree to which the b-labeled edge leads. T_b is a binary decision tree representing the function $f_b : \mathbb{B}^{n-1} \to \mathbb{B}$ defined as $f_b(x_2, \ldots, x_n) = f(b, x_2, \ldots, x_n)$. T_0 will be referred to as the *negative sub-tree*, and T_1 as the *positive sub-tree* of T.

Fig. 2.1 shows some examples of binary decision trees. We will use the convention that 1-labeled edges will be solid and 0-labeled edges will be dashed.

It is clear that given a decision tree T for a function $f : \mathbb{B}^n \to \mathbb{B}$ and an assignment $\vec{b} \in \mathbb{B}^n$ for the variables in f, $\vec{b}$ describes a path in T to a terminal node, by starting at the root and deciding for each node which edge to follow based on the assignment to that variable according to $\vec{b}$. The terminal node reached by this path is the resulting value of f for the assignment.

There is a clear relation as well between a binary decision tree for a Boolean function f and the Shannon expansion of f. In the binary decision tree T representing f with the root labeled by the variable x_1, the positive sub-tree is the binary decision tree for f_{x_1}, and the negative sub-tree is the binary decision tree for $f_{\neg x_1}$.

Although binary decision trees can be used to represent Boolean functions and provide an easy way to evaluate them, they are usually not helpful in practice because their size is exponential on the number of variables. However, by observing their structure one will notice there is a lot of redundancy in their representation of a Boolean function. For example, in Fig. 2.1c the sub-trees rooted on the leftmost and rightmost x_3 nodes are identical. This opens up the possibility of storing this sub-tree only once and having the negative edge of the leftmost x_2 node and the positive edge of the rightmost x_2 node both lead to the same node. Fig. 2.2a shows the result of this operation. Because there are two edges pointing to the same node, the structure is no longer a tree, but a directed acyclic graph, or DAG.



(a) $f_1(x_1) = \neg x1$ (b) $f_2(x_1, x_2) = x_1 \vee x_2$

(c) $f_3(x_1, x_2, x_3) = (x_1 \wedge \neg x_2) \vee ((x_1 \leftrightarrow x_2) \wedge \neg x_3)$

Figure 2.1: Examples of binary decision trees representing Boolean functions.

Looking at the DAG, one can find further redundancy upon noticing that the other two x_3 nodes have both outgoing edges leading to the same constant (0 in one case, 1 in the other). This means the result of the function in these paths is independent of the variable x_3. Therefore, the structure can be further reduced by removing the two x_3 nodes and having the edges that lead to them redirected to point directly to the leaves. Fig. 2.2b shows the final DAG after this second modification. Notice that the nodes representing the constants 0 and 1 also appear only once, for the same reason the x_3 nodes were merged.

The resulting DAG obtained by applying these operations is the BDD for the function represented by the original binary decision tree. The significant reduction in size is clear, going from 7 internal nodes to only 4. If the nodes for the Boolean constants are considered, the reduction is even more evident, decreasing from 15 nodes in total to only 6.

This example illustrates the two properties that differentiate a BDD from a binary decision tree, guaranteeing that the BDD is reduced:

Non-redundancy No nodes can have both outgoing edges pointing to the same child. Any edges that would point to such a node point to the child instead.

Uniqueness There must not be more than one copy of a node with the same label, positive child and negative child. All edges leading to a node with label x_i, positive child B_1 and negative child B_0 must point to the same node.

The reduction operations performed to obtain these properties are illustrated in Fig. 2.3.

With this we arrive at the following definition for BDDs:

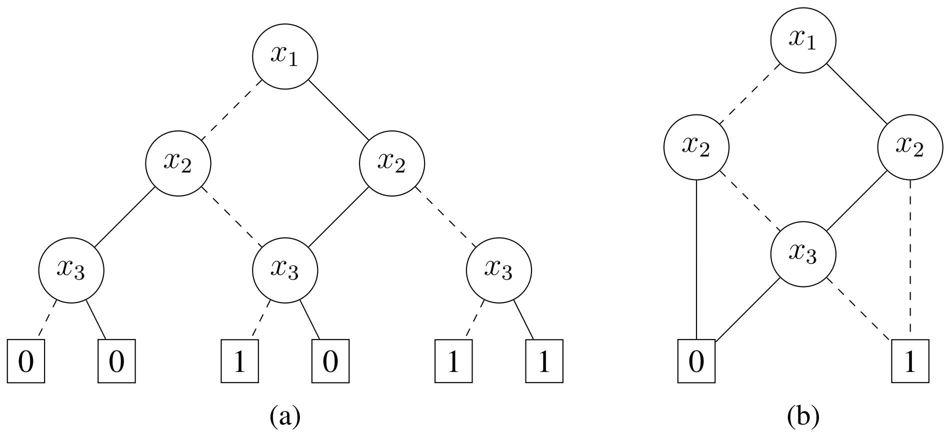


Figure 2.2: Result of applying reduction operations to eliminate redundancies in the binary decision tree from Fig. 2.1c.

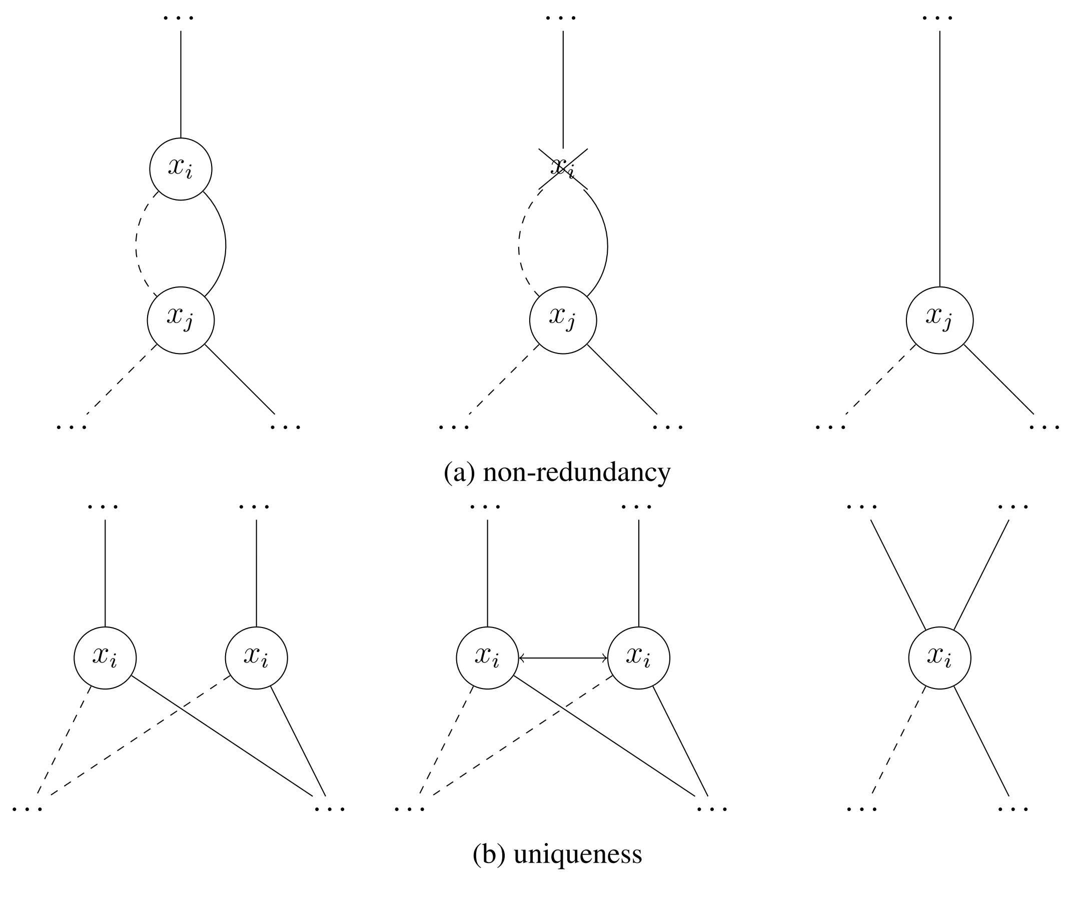


Figure 2.3: Reduction properties of BDDs.

Definition 1. *A Binary Decision Diagram, or BDD, is the directed acyclic graph obtained by applying reductions to a binary decision tree such that the properties of non-redundancy and uniqueness are fulfilled.*

Therefore, one can construct a BDD by going through the process of building a binary decision tree but making sure these properties are maintained during the construction. The *uniqueness* property is usually obtained by keeping all BDD nodes in a table, so the existence of a node can be easily checked and a reference to the already-existing one can be used instead of creating a new one. *Non-redundancy* can be ensured by checking before creating a node if both its children are the same, and returning the child instead of creating the node if that is the case.

Algorithm 10 presents a procedure that returns a BDD given a label and positive and negative children, while maintaining the reduction properties. The notation (x_i, B_0, B_1) will be used to denote a BDD node with label x_i, negative child B_0 and positive child B_1.

```
Data: A label $x_i$ and two BDDs $B_0$ and $B_1$
Result: A BDD whose root is labeled by $x_i$ and has $B_0$ and $B_1$ as its negative and
        positive child, respectively
Procedure MkBDD ($x_i$, $B_0$, $B_1$)
    if $B_0$ and $B_1$ are the same BDD $B$ then
        return $B$
    else if a node $B = (x_i, B_0, B_1)$ already exists then
        return $B$
    else
        $B \leftarrow (x_i, B_0, B_1)$
        insert $B$ in table
        return $B$
    end
```

Algorithm 1: BDD node construction.

2.3.1 Ordering of a BDD

One very important factor to consider when working with BDDs is the *ordering* of their variables. In order to build a BDD from a Boolean expression, one must first specify a strict total order of the variables appearing in it. This will determine which variables will come before others in the BDD. A node labeled by x_i can have a child labeled by x_j only if $x_i < x_j$ according to the variable ordering. This means that the lowest variable in the ordering can only appear as root of the BDD, and the highest variable can only have terminal nodes as children. Note however that intermediary variables can also be the root, have terminals as children, both, or neither, since it is possible for variables to not appear in the BDD at all if the result of the expression is independent of them.

The variable ordering of a BDD has a major influence on its size, to the point that the BDD for a specific function might have linear size with one ordering and exponential size with another. Therefore, one of the most popular topics of research in BDDs is in finding heuristics to determine variable orderings that produce efficient BDDs. This can vary a lot depending on the formula, and is influenced by the dependencies between variables. For example, placing variables that appear together in the same sub-expression close to each other in the ordering tends to produce smaller BDDs. The reasoning is that variables that appear together tend to influence the assignment of one another, thus reducing the number of different possible paths. On the other hand, other variables that appear outside of the sub-expression will not be influenced by their individual assignments, but by the result of the sub-expression, thus reducing the number of branches.

Fig. 2.4 exemplifies the advantage of this intuition. Both BDDs depicted encode the expression $(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6)$, the first using the ordering $x_1 < x_3 < x_5 < x_2 < x_4 < x_6$ and the second the ordering $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$. In the first ordering, the variables x_1, x_3 and x_5, which are from different sub-expressions, were placed together. Because of that, up to the x_5 nodes, the BDD is as large as possible, with $2^3 = 8$ total edges leaving x_5 nodes, all leading to different nodes. This happens because, since these variables influence the value of different sub-expressions, every single assignment of $\langle x_1, x_3, x_5 \rangle$ leads to a different set of satisfying assignments to the rest of the variables, producing a fully branched BDD. On the other hand, the second ordering places together variables from the same sub-expression, for example x_1 and x_2. Because their influence in the formula is contained by their own sub-expression, there is as little branching as possible. No matter the assignments to the individual variables, their influence to the total formula is only on the evaluation of $(x_1 \wedge x_2)$, which can only assume values 0 or 1. The same is the case for the other variable pairings.

Of course, this heuristic becomes less effective when the same variable appears in several different sub-expressions together with different groups of other variables. Since this is the case for most of the formulas of practical interest, this heuristic alone may not always provide significant results. Several different heuristics have been developed (RICE; KULHARI, 2008), but not all functions will have good orderings as easy to find as this example, and some might not have orderings that lead to a linear BDD at all.

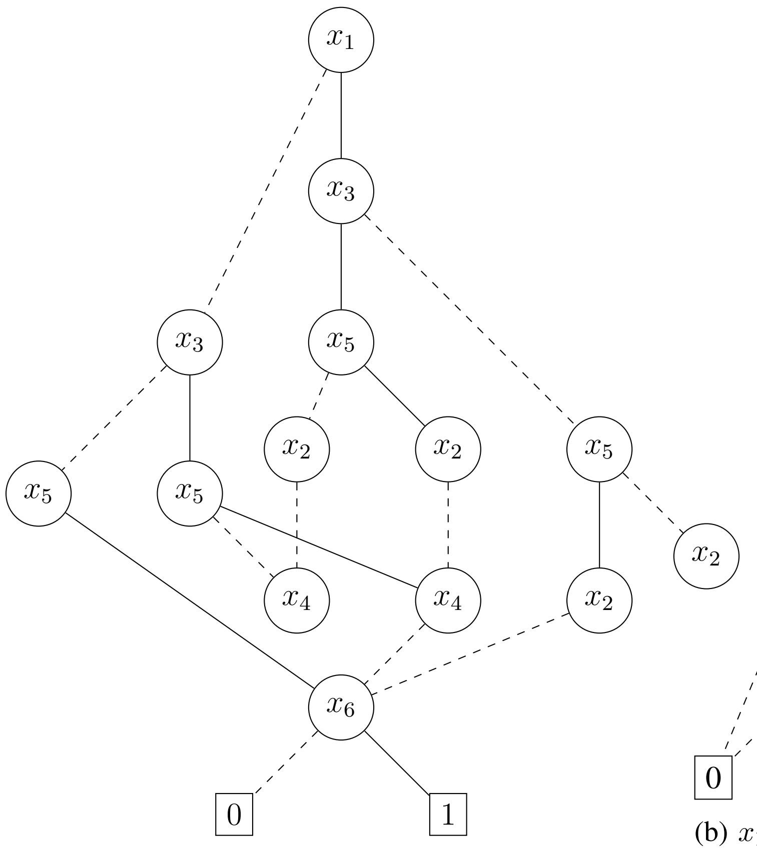
Given the difficulty in finding good heuristics for ordering BDDs, many applications resort instead to techniques for dynamic variable ordering citeprudell, which tries to progressively improve the ordering of the variables in the BDD during program execution by attempting small modifications and checking if they lead to an improvement on the size of the BDD.

In algorithms presented from now on, the strict partial order $\prec$ will be used to compare BDDs using the same ordering. This relation is defined as follows for two BDDs B_1 and B_2:

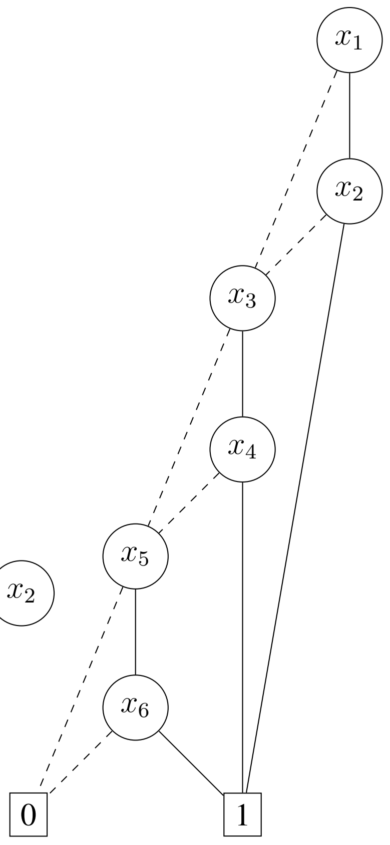
- If B_1 is not a terminal but B_2 is, $B_1 \prec B_2$.
- If both B_1 and B_2 are rooted, and x and y are respectively the variables of the roots of B_1 and B_2, $B_1 \prec B_2$ if $x < y$ according to the variable ordering, and $B_2 \prec B_1$ if $y < x$ according to the variable ordering.

2.3.2 BDD operations

Another way to construct a BDD is by combining basic BDDs using analogs of Boolean operations. For example, to build a BDD for the expression $x_1 \wedge \neg(x_2 \vee x_3)$ one would start with BDDs for x_2 and x_3, apply a $\vee$ operation to them, negate the result and combine it with a BDD for x_1 using a $\wedge$ operation. These operations are also useful when BDDs have to be constructed iteratively.



(a) $x_1 < x_3 < x_5 < x_2 < x_4 < x_6$



(b) $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$

Figure 2.4: BDDs for the expression $(x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6)$ for two different variable orderings, with missing positive edges understood to lead to the terminal 1 and missing negative edges to the terminal 0.

The following are the basic BDD building blocks and operations which are used in BDD construction and manipulation.

Constants Boolean constants are represented by terminal nodes labeled with 0 or 1.

Variables A BDD representing a single variable is comprised of a node labeled with that variable with the positive child pointing to the constant 1 and the negative child pointing to the constant 0.

Negation A BDD is negated by switching the 0 node by the 1 node and vice-versa. To avoid having to go down the whole BDD from the root to the terminals, some implementations employ a flag to indicate whether a BDD is negated or not.

Binary operations Binary operations such as $\wedge$, $\vee$ or $\oplus$ are obtained by recursively traversing the two BDD operands while maintaining the variable ordering. The general template for such operations is described in Algorithm 8. This general algorithm can still be optimized for specific operations. For example, the algorithm for applying the $\wedge$ operator to two BDDs can return immediately if one of the BDDs is the terminal 0, since the result will be 0 independently of the other BDD. The same can be done for $\vee$ if one of the BDDs is the terminal 1.

Functional composition Given two BDDs F and G representing functions $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n)$, the composition of F and G on variable x_i, denoted by $F|_{x_i=G}$, is a BDD encoding the function $h(x_1, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, g(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n)$. In the case when G is a terminal node, $F|_{x_i=G}$ encodes one of the cofactors of f in respect to x_i, the positive one if G is the terminal node 1 and the negative one if G is the terminal node 0. Algorithm 12 presents a method for performing BDD composition as shown in (SOMENZI, 1999). Note that this does not necessarily remove x_i from the BDD, since G might be dependent on x_i.

```
Data: An arbitrary Boolean binary operation denoted by $\diamond$ and two BDDs $X$ and $Y$
      to be combined according to it
Result: The BDD $X\check{\diamond}Y$, where $\check{\diamond}$ is the BDD equivalent of the operator $\diamond$
1 if $X$ and $Y$ are both constants then
2     return Value($X$) $\diamond$ Value($Y$)
3 else if $X \prec Y$ then
4     return MkBDD(Root($X$), $X_0\check{\diamond}Y$, $X_1\check{\diamond}Y$)
5 else if $X \succ Y$ then
6     return MkBDD(Root($Y$), $X\check{\diamond}Y_0$, $X\check{\diamond}Y_1$)
7 else // X and Y have the same root
8     return MkBDD(Root($X$), $X_0\check{\diamond}Y_0$, $X_1\check{\diamond}Y_1$)
```

Algorithm 2: Template for binary operations on BDDs.

```
Data: A BDD $F$, a label $x_i$ and a BDD $G$
Result: The BDD $F|_{x_i=G}$
1  Procedure Compose($F$, $x_i$, $G$)
2      if $F$ is a constant or Root($F$) $> x_i$ then
3          return $F$
4      else if Root($F$) = $x_i$ then
5          return $(\neg G\check{\wedge}F_0)\check{\vee}(G\check{\wedge}F_1)$
6      else if Root($F$) = Root($G$) then
7          return MkBDD(Root($F$), Compose($F_0$, $x_i$, $G_0$), Compose($F_1$, $x_i$,
           $G_1$))
8      else if Root($F$) < Root($G$) then
9          return MkBDD(Root($F$), Compose($F_0$, $x_i$, $G$), Compose($F_1$, $x_i$,
           $G$))
10     else
11         return MkBDD(Root($F$), Compose($F$, $x_i$, $G_0$), Compose($F$, $x_i$,
           $G_1$))
12     end
```

Algorithm 3: Functional composition on BDDs.

2.3.3 Canonicity of BDDs

One of the most significant properties of BDDs is that, given a certain variable ordering, they are a *canonical representation* of a Boolean function. This means that there is a single BDD that represents each function. Thus, if two Boolean functions are equivalent (that is, they produce the same result for every assignment), their BDDs will be identical. This property, combined with the fact that every BDD node is stored only once, means that for two Boolean functions f and g, given their respective BDD representations F and G, their equivalence can be determined in constant time simply by testing whether F and G refer to the same BDD.

2.3.4 Paths in a BDD

A *path* in a BDD is a sequence of alternating nodes and edges $\pi = \langle v_1, e_1, v_2, e_2, \ldots, e_k, v_{k+1}\rangle$ such that for all $i \leq k$ the edge e_i connects the node v_i with a child node v_{i+1}. The *length* of the path is the number of edges in it. We say that a path π

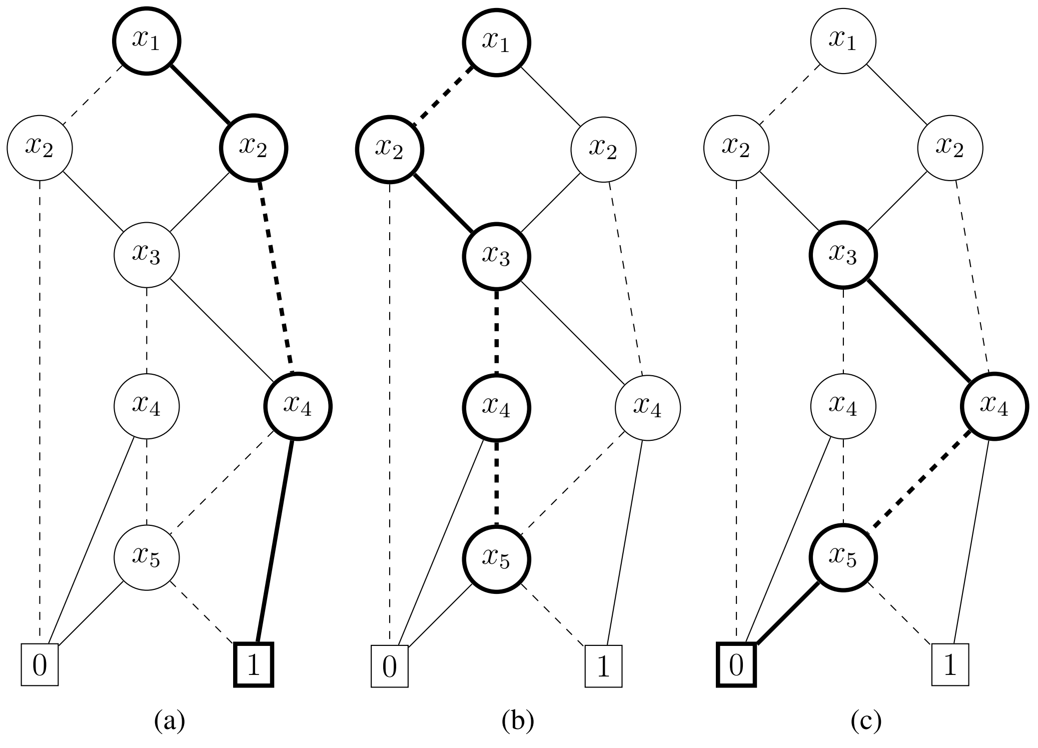


Figure 2.5: Examples of paths on a BDD.

is *complete* if v_1 is the root of the BDD and v_k is a terminal node.

Every path in a BDD defines a partial assignment to the variables in the BDD. For a given path π of length k, for all $i \leq k$, if v_i is labeled by variable x_j and e_i is labeled by a Boolean b, then x_j is assigned b in the assignment defined by the path. Note that this partial assignment might skip variables that have been suppressed by the reduction rules.

Similarly, take a partial assignment $\nu = \langle x_i \mapsto b_i, x_{i+1} \mapsto b_{i+1}, \ldots, x_{i+k} \mapsto b_{i+k}\rangle$ where $x_i, x_{i+1}, \ldots, x_{i+k}$ are consecutive variables in the ordering of the BDD. Then ν, together with a starting node v labeled by a variable x', $x_i \leq x' \leq x_{i+k}$, defines a path, obtained by starting at v and for every node labeled by x_j taking the edge labeled by b_j. If the starting node is not specified, it is assumed to be the root.

We say that a path π is *satisfiable* if it does not end in the terminal node 0. This is equivalent to saying there exists a path π' such that the last node of π is the starting node of π' and the last node of π' is the terminal node 1. We say that a path π is *satisfying* if it ends in the terminal node 1. Note that if π is satisfiable and complete then it is necessarily satisfying.

Fig. 2.5 shows examples of paths in BDDs. They respectively define the partial assignments $\langle x_1 \mapsto 1, x_2 \mapsto 0, x_4 \mapsto 1\rangle$, $\langle x_1 \mapsto 0, x_2 \mapsto 1, x_3 \mapsto 0, x_4 \mapsto 0\rangle$, and $\langle x_3 \mapsto 1, x_4 \mapsto 0, x_5 \mapsto 1\rangle$. The path represented in (a) begins in the root and ends in the terminal node 1, making it a complete path. The path in (b), on the other hand, ends in a variable node, and therefore is not complete. The path in (c), although ending in a terminal node, is not complete because it does not begin in the root.

2.3.5 BDD applications

Since their introduction, BDDs have become a widely used tool for solving a variety of problems, with their main contribution in the field of formal verification. In model checking, BDDs can be used as a symbolical representation of the state space of a system (BURCH et al., 1992). Due to being a canonical representation of Boolean functions, they can be used to test circuit equivalence (BRAND, 1993). They can also be used for solving many problems related to satisfiability, such as enumerating, counting or randomly generating solutions (KNUTH, 2009). These examples merely graze the surface of the applications of BDDs. Many BDD variants have also been proposed that specialize in different types of problems. For example, Zero-suppressed Decision Diagrams (ZDDs) (MINATO, 1993) use different reduction rules which make them efficient representations of families of sets, allowing them to be used for example in representing sets of clauses in CNF (CHATALIC; SIMON, 2000).

3 PROBLEM DEFINITION

The main problem we wish to solve is the following:

Given a relation between two vectors of Boolean variables represented by the characteristic function $f : \mathbb{B}^n \times \mathbb{B}^m \to \mathbb{B}$, obtain a function $g : \mathbb{B}^n \to \mathbb{B}^m$ such that for any $\vec{y} \in \mathbb{B}^n$, $f(\vec{y}, g(\vec{y})) = 1$.

In the context of this problem, f is called the *specification*, while g is called the *implementation*. The specification is interpreted as describing a desired relationship between inputs and outputs of a function, and the implementation describes how to obtain an output from an input such that this relationship is maintained. Therefore, in the expression $f(\vec{y}, \vec{x})$, $\vec{y} = \langle y_1, \ldots, y_n \rangle$ are called the *input variables*, and $\vec{x} = \langle x_1, \ldots, x_m \rangle$ the *output variables*.

Note that the relation between f and g is not necessarily one-to-one. In fact, for a given function f there are three possibilities:

1. For each $\vec{y} \in \mathbb{B}^n$ there is a single $\vec{x} \in \mathbb{B}$ such that $f(\vec{y}, \vec{x}) = 1$. In this case, there is only one function g that preserves $f(\vec{y}, g(\vec{y})) = 1$.

2. There is at least one $\vec{y} \in \mathbb{B}^n$ for which there are $\vec{x}, \vec{x}' \in \mathbb{B}^m$, $\vec{x} \neq \vec{x}'$, such that $f(\vec{y}, \vec{x}) = 1$ and $f(\vec{y}, \vec{x}') = 1$. In this case, there can be two functions g and g' such that $g(\vec{y}) = \vec{x}$ and $g'(\vec{y}) = \vec{x}'$ and both are a correct implementation of f.

3. There is at least one $\vec{y} \in \mathbb{B}^n$ for which there is no $\vec{x} \in \mathbb{B}^m$ such that $f(\vec{y}, \vec{x}) = 1$. In this case, f specifies only a partial function. Since the definition of the problem requires g to be a total function, an implementation cannot be synthesized.

For the first case there is no problem. Any correct synthesis procedure will generate the same implementation. For the second case, depending on choices made during the synthesis process one option will be produced from a set of equally correct implementations.

The biggest problem is the case when f does not describe a total function. In this case, the synthesis cannot be performed and an error will have to be signaled. The following definition serves to identify such a case:

Definition 2. *A Boolean function $f(\vec{y}, \vec{x})$ is said to be* realizable *if $\exists \vec{x}(f(\vec{y}, \vec{x}))$ is valid, which is equivalent to saying that $\forall \vec{y} \exists \vec{x}(f(\vec{y}, \vec{x}))$ is true.*

Since realizability is a requirement to perform synthesis, it will be necessary to be able to identify when this property is violated. Therefore, we will be concerned not only with the synthesis itself, but also with verifying realizability.

4 METHODOLOGY

As stated in Chapter 3, our goal is to synthesize a total function $g(\vec{y})$ that implements a given specification $f(\vec{y}, \vec{x})$, if it exists. We will use BDDs to represent this function. Recall that, while the return value of $g(\vec{y})$ is a vector of Booleans, a BDD encodes a function that outputs a single Boolean variable. Therefore, one BDD is not enough to encode g. Instead, our synthesis methods will return a sequence of BDDs $\vec{W} = \langle W_1, \ldots, W_m \rangle$, where each BDD W_i represents a function $g_i(\vec{y})$ that returns a satisfying value for output variable x_i. Because $g_i(\vec{y})$ can be seen as a witness function for x_i in the formula $\exists \vec{x}(f(\vec{y}, \vec{x}))$, we call W_i a *witness BDD* for x_i. In certain cases we will use an intermediary representation for a witness for x_i where it is not dependent only on the input variables, but also on the assignment of other output variables. We call those *partial witness functions*, and the sequence of BDDs $\vec{W}' = \langle W'_1, \ldots, W'_m \rangle$ representing them *partial witness BDDs*. When the meaning is made clear by the context we will use the term *witness* to refer to both witness functions and witness BDDs.

We will start by presenting a method for traversing a BDD as a graph. BDDs are usually manipulated as representations of Boolean functions, using analogs to typical Boolean operations. However, internally they are still represented as DAGs, and as such can be traversed recursively. In some of the algorithms we use we will need to perform this traversal, so we describe a general method that can be used later in different situations. Next, we will introduce a specific BDD ordering that is particularly relevant to this work, as some of the algorithms presented in this chapter require the BDD to follow this ordering. Then, we will present a novel method of quantifier elimination that will be employed in our algorithms. Finally, we will describe our methods for testing realizability of a specification and for performing the synthesis of the implementation from it.

4.1 About BDD traversal

During this chapter we will propose different algorithms for testing realizability and performing synthesis. Some of them will work by traversing the BDD as a DAG rather than abstracting its structure and treating it like a Boolean function. This can be done by a recursive traversal similar to one that would be performed in a binary tree. However, since in a DAG different paths can lead to the same node, performing this traversal naively would lead to the result for the same node being computed several times. To avoid this, we use a dynamic programming solution, implementing a cache that stores the results for nodes that have already been visited, in the form of a hash table.

Since this caching operation is purely a practical concern, and it would obscure the logic of the algorithms that use it, we will assume that every procedure *Traversal* that needs to traverse a BDD is accompanied by supporting operations *Traversal_Init* and *Traversal_Rec* that follow the model presented in Algorithm 4. *Traversal_Init* should be called to initiate the traversal, clearing the hash table and then calling the first recursion step. *Traversal_Rec* should be called instead of the main procedure on every recursive call. It verifies if the current BDD node has been visited already, returning the cached result if so, and otherwise continuing the traversal and caching the result after it is finished.

```
Data: A BDD $B$ and extra arguments to be passed to the Traversal function
Result: The output of the Traversal function
Procedure Traversal_Init($B, \ldots$)
    clear table
    return Traversal_Rec($B, \ldots$)
Procedure Traversal_Rec($B, \ldots$)
    if table contains an entry for $B$ then
        return result stored in the entry for $B$
    else
        $v \leftarrow$ Traversal($B, \ldots$)
        store entry in table associating $B$ to $v$
        return $v$
    end
```

Algorithm 4: Supporting operations for BDD recursive traversal.

4.2 Input-first BDDs

As mentioned in Chapter 2, the structure of a BDD can be influenced greatly by the ordering used for the variables in it. There is a specific ordering which is particularly relevant in the development of our methods, as presented in the following definition:

Definition 3. *A BDD B for a specification is said to use an* input-first *order if for every input variable y and every output variable x, $y < x$ in the ordering used for B.*

Informally, an input-first BDD has all input variables appearing higher in the BDD than the output variables. This is an intuitive ordering that supports synthesis approaches. We expect the output variables to depend on the input variables, therefore it makes sense to decide on their values after knowing the assignment for the input. However, this is not necessarily an efficient ordering, and in fact can lead to intractably large BDDs depending on the case.

4.3 Quantifier elimination

4.3.1 Self-substitution

Recall from Chapter 2.2.1 that quantifier elimination in QBFs is usually performed by expanding a formula into positive and negative cofactors and combining them using conjunction or disjunction depending on whether the quantifier is universal or existential.

	$\forall x'(f(\vec{x},x'))$	$\exists x'(f(\vec{x},x'))$
Shannon expansion	$f(\vec{x},0) \wedge f(\vec{x},1)$	$f(\vec{x},0) \vee f(\vec{x},1)$
Self-substitution	$f(\vec{x},f(\vec{x},0))$	$f(\vec{x},f(\vec{x},1))$

Table 4.1: Equivalent formulas using each method of quantifier elimination.

We will call this technique the *Shannon expansion method* for quantifier elimination, due to its similarity to the theorem of the same name.

Our first contribution in this work is an alternative technique for quantifier elimination in QBFs, which we call the *self-substitution* method. This technique is based on the following lemma:

Lemma 1. *Let $\varphi = Qx'(f(\vec{x},x'))$ be an open QBF formula, where Q is either a universal or existential quantifier. Let q be 0 if Q is universal and 1 if Q is existential. Then, $Qx'(f(\vec{x},x'))$ is logically equivalent to $f(\vec{x},f(\vec{x},q))$.*

Proof. If Q is a universal quantifier, then we will prove that for any assignment $\vec{\sigma}$, $\forall x'(f(\vec{\sigma},x')) = 1$ iff $f(\vec{\sigma},f(\vec{\sigma},0)) = 1$:

($\Rightarrow$) If $\forall x'(f(\vec{\sigma},x')) = 1$, this means that both $f(\vec{\sigma},0) = 1$ and $f(\vec{\sigma},1) = 1$. Therefore, $f(\vec{\sigma},f(\vec{\sigma},0)) = f(\vec{\sigma},1) = 1$.

($\Leftarrow$) If $f(\vec{\sigma},f(\vec{\sigma},0)) = 1$, then it cannot be the case that $f(\vec{\sigma},0) = 0$ otherwise $f(\vec{\sigma},f(\vec{\sigma},0)) = f(\vec{\sigma},0) = 0$. Therefore, $f(\vec{\sigma},0) = 1$, and so $f(\vec{\sigma},1) = f(\vec{\sigma},f(\vec{\sigma},0)) = 1$. Since both $f(\vec{\sigma},0) = 1$ and $f(\vec{\sigma},1) = 1$, then $\forall x'(f(\vec{\sigma},x')) = 1$.

If Q is an existential quantifier, then we will prove that for any assignment $\vec{\sigma}$, $\exists x'(f(\vec{\sigma},x')) = 0$ iff $f(\vec{\sigma},f(\vec{\sigma},1)) = 0$:

($\Rightarrow$) If $\exists x'(f(\vec{\sigma},x')) = 0$, this means that both $f(\vec{\sigma},0) = 0$ and $f(\vec{\sigma},1) = 0$. Therefore $f(\vec{\sigma},f(\vec{\sigma},1)) = f(\vec{\sigma},0) = 0$.

($\Leftarrow$) If $f(\vec{\sigma},f(\vec{\sigma},1)) = 0$, then it cannot be the case that $f(\vec{\sigma},1) = 1$, otherwise $f(\vec{\sigma},f(\vec{\sigma},1)) = f(\vec{\sigma},1) = 1$. Therefore, $f(\vec{\sigma},1) = 0$, and so $f(\vec{\sigma},0) = f(\vec{\sigma},f(\vec{\sigma},1)) = 0$. Since both $f(\vec{\sigma},1) = 0$ and $f(\vec{\sigma},0) = 0$, then $\exists x'(f(\vec{\sigma},x')) = 0$. $\square$

Following this lemma, quantifier elimination can be performed by replacing quantified formulas by their quantifier-free equivalents. Table 4.1 compares the formulas produced by quantifier elimination using Shannon expansion and self-substitution.

Note that Lemma 1 has an additional theoretical significance. Unlike Shannon expansion, self-substitution does not only provide an alternative method of quantifier elimination, but also directly defines a witness for an existentially quantified formula. The following corollary, derived directly from Lemma 1, expresses this result:

Corollary 1. *Given a formula $\varphi = \exists x'(f(\vec{x},x'))$, $f(\vec{x},1)$ is a witness to φ.*

4.3.2 Quantifier elimination in BDDs

Normally, one would expect that for k quantifiers it would be necessary to apply quantifier elimination k times. However, the case with only one type of quantifier can be decided in constant time given the input in the format of a BDD:

Lemma 2. *Let $\varphi = Q\vec{x}(f(\vec{x}))$ be a QBF with only one type of quantifier Q, which is either universal or existential. Let B be the BDD representation of the Boolean function f. Then,*

1. *If Q is universal, φ evaluates to 1 iff B is the terminal node 1.*

2. *If Q is existential, φ evaluates to 0 iff B is the terminal node 0.*

Proof. Let $b_\forall = 1$ and $b_\exists = 0$. First note that, according the semantics of the quantifiers, $Q\vec{x}(f(\vec{x}))$ evaluates to b_Q iff for any assignment $\vec{\sigma}$ for $\vec{x}$, $f(\vec{\sigma})$ evaluates to b_Q. This applies both to the case when Q is universal, in which case b_Q is 1, and to the case when Q is existential, in which case b_Q is 0.

Since a complete assignment defines a path to a terminal node, all assignments evaluating to the same value b_Q means that all paths in B lead to the terminal node labeled by b_Q. But according to the non-redundancy property of BDDs, this means that B is reduced to the terminal node labeled by b_Q. □

In practice, Lemma 2 means that, when a function $f(\vec{x})$ is represented by a BDD B, it is not necessary to use quantifier elimination to evaluate $\forall\vec{x}(f(\vec{x}))$ or $\exists\vec{x}(f(\vec{x}))$. Rather, this can be done simply by testing if B is different from the appropriate terminal node.

4.4 Realizability

As explained in Chapter 3, to be able to synthesize an implementation $g(\vec{y})$ for a specification $f(\vec{y}, \vec{x})$, f must be realizable, meaning for each assignment to $\vec{y}$ there must be a satisfying assignment to $\vec{x}$. This section will present two methods for testing if a specification is realizable. The first can be applied to BDDs using any ordering, and the second is exclusive to input-first BDDs.

4.4.1 Realizability in BDDs of general ordering

As defined in Chapter 3, $\varphi = f(\vec{y}, \vec{x})$ is realizable if $\forall\vec{y}\exists\vec{x}(f(\vec{y}, \vec{x}))$ evaluates to 1. Therefore, testing realizability can be approached simply as quantifier elimination of a QBF. For that, one can use either Shannon expansion or self-substitution.

Since φ has two types of quantifiers, Lemma 2 cannot be used directly to test realizability. However, if the existential quantifiers are eliminated first, we obtain a QBF $\varphi' = \forall\vec{y}(f'(\vec{y}))$ that is equivalent to the original formula φ. At this point, Lemma 2 can be applied to test realizability by checking if the BDD representation of $f'(\vec{y})$ is the terminal node 1. This procedure is described in Algorithm 5, which can be implemented using either of the methods of quantifier elimination.

```
Data: A BDD $B$ representing a Boolean function $f(\vec{y}, \vec{x})$, where $\vec{x} = \langle x_1, \ldots, x_m \rangle$
Result: A Boolean value indicating if $f$ is realizable
1 $B_0 \leftarrow B$
2 for $i \leftarrow 1$ to $m$ do
3     $B_i \leftarrow$ EliminateExistentialQuantifier($i$, $B_{i-1}$)
4 end
5 return IsOne($B_m$)
```

Algorithm 5: Algorithm for checking realizability from a BDD.

4.4.2 Realizability in input-first BDDs

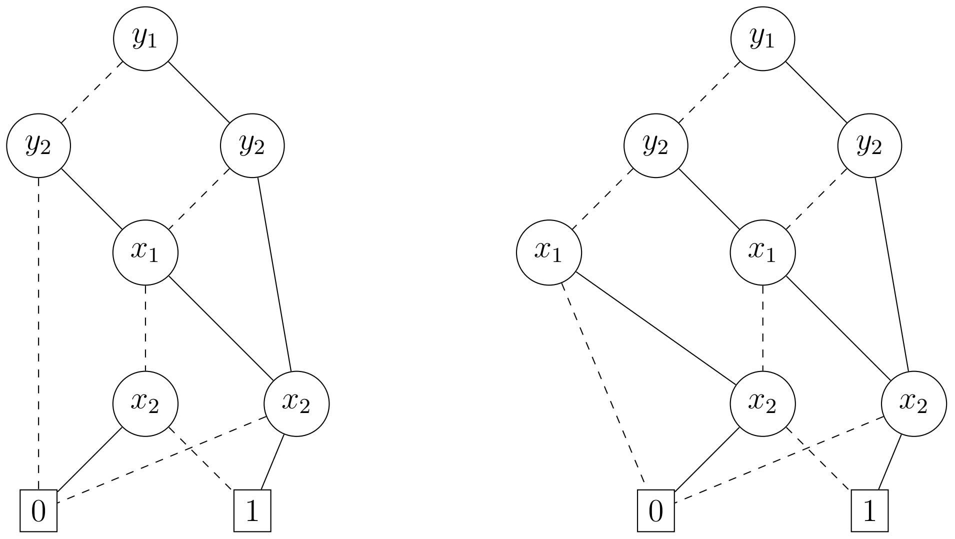
If we have an input-first BDD representation of f, it is possible to check realizability by analysing the structure of the BDD, as stated by the following lemma:

Lemma 3. *Given an input-first BDD B representing a function $f(\vec{y}, \vec{x})$, f is realizable iff no node for an input variable has a child pointing to the terminal node 0 in B.*

Proof. Suppose f is not realizable. This is equivalent to saying that there exists an assignment $\vec{\nu}$ for $\vec{y}$ such that for any assignment $\vec{\sigma}$ to $\vec{x}$, $f(\vec{\nu}, \vec{\sigma}) = 0$. Since B uses an input-first ordering, all input variables come before all output variables. Therefore, $\vec{\nu}$ defines a path from the root of B through the input variables, reaching a node v. $f(\vec{\nu}, \vec{\sigma}) = 0$ for every $\vec{\sigma}$ iff every complete path from v leads to the terminal node 0. But from the non-redundancy of BDDs, this happens iff v is the terminal node 0. Therefore, f is not realizable iff there is a path from the root of B going only through input variables that ends in the terminal node 0. □

Fig. 4.1 illustrates Lemma 3. In (a) the left y_2 input variable has its negative child pointing to the terminal node 0. This means that for the input $\langle y_1 \mapsto 0, y_2 \mapsto 0 \rangle$ there is no possible output that satisfies the formula represented by this BDD, and therefore it is not realizable. Compare with the realizable BDD in (b), which has $\langle x_1 \mapsto 1, x_2 \mapsto 0 \rangle$ as a satisfying output in this case.

Therefore, in an input-first BDD realizability can be verified by navigating the BDD until reaching a node labeled by an output variable. If in some path the terminal node 0 is reached before finding an output variable, the formula is not realizable. Algorithm 6 describes this process, using operations in the model of Algorithm 4 for traversing the BDD. The algorithm starts at the root of the BDD and follows all paths through the input variables until either a) reaching an output variable or the terminal node 1, in which case the path is satisfiable, or b) reaching the terminal node 0, in which case the path is



(a) not realizable (b) realizable

Figure 4.1: Example of not-realizable and realizable input-first BDDs.

unsatisfiable and so the BDD is unrealizable.

```
Data: An input-first BDD $B$ representing a Boolean function $f(\vec{y}, \vec{x})$
Result: A Boolean value indicating if $f$ is realizable
Procedure IsRealizable($B$)
    if $B$ is a terminal node then
        return Value($B$)
    else if $B$ is an input variable node $(y_j, B_0, B_1)$ then
        return IsRealizable_Rec($B_0$) $\wedge$ IsRealizable_Rec($B_1$)
    else
        return 1
```

Algorithm 6: BDD traversal algorithm for checking realizability from an input-first BDD.

4.5 Synthesis

Having established methods to verify realizability of a specification f, we can address our main objective, which is to synthesize an implementation g from f. In this section we present two methods for doing so, starting from a BDD representation of f. The first is an extension of the quantifier elimination-based approach for testing realizability, and abstracts the representation of the formula as a BDD. On the other hand, the second is designed with the BDD representation in mind, being based on traversing the structure of the BDD, and requires the BDD to use an input-first ordering.

As described in the beginning of the chapter, our methods return g in the form of a sequence of witness BDDs $\vec{W} = \langle W_1, \ldots, W_m \rangle$. Although each W_i represents a different function, individual nodes can be shared between different BDDs. Therefore, the sequence can be visualized as a single multi-rooted BDD. This visualization will be useful when the BDD is mapped to a hardware or software implementation, because it avoids duplicated computations.

4.5.1 Synthesis via quantifier elimination

Our first method for synthesis is an extension of Algorithm 5 for checking realizability. Recall that in Algorithm 5 existentially quantified variables were progressively eliminated from the formula using either Shannon expansion or self-substitution, and if by the end the formula had been reduced to the terminal node 1 then it was realizable. We extend this algorithm to synthesize at each step a partial witness for the variable being eliminated, and afterwards turn them into full witnesses by removing the extra variables through BDD composition. The method is described in more details next.

Recall from Corollary 1 that for an open QBF of the form $\exists x(f(\vec{y}, x))$, $f(\vec{y}, 1)$ is a witness for x. Note that in Algorithm 5 the BDD B_i represents a function $f_i(\vec{y}, x_i, \dots, x_m)$ that is equivalent to the open QBF $\exists x_1, \dots, x_{i-1}(f(\vec{y}, x_1, \dots, x_{i-1}, x_i, \dots, x_m))$. For $i = m$, we have B_m representing $f_m(\vec{y}, x_m)$. We can therefore apply Corollary 1 to obtain $g_m(\vec{y}) = f_m(\vec{y}, 1)$ as a witness for x_m. If we apply the same reasoning for x_{m-1}, we obtain from function $f_{m-1}(\vec{y}, x_{m-1}, x_m)$ the witness $g'_{m-1}(\vec{y}, x_m) = f_{m-1}(\vec{y}, 1, x_m)$. Note that in this case g'_{m-1} is a partial witness, being dependent not only on the input variables but also on x_m. However, since we have the witness $g_m(\vec{y})$ for x_m, we can compose it in g_{m-1}, obtaining the witness $g_{m-1}(\vec{y}) = f_{m-1}(\vec{y}, 1, g_m(\vec{y}))$. Repeating this process for the remaining output variables, replacing in the partial witnesses the variables whose witnesses we have already computed, we obtain the sequence of witnesses $\langle g_1(\vec{y}), \dots, g_m(\vec{y})\rangle$, where $g_i(\vec{y}) = f_i(\vec{y}, 1, g_{i+1}(\vec{y}), \dots, g_m(\vec{y}))$.

Algorithm 7 extends Algorithm 5 to return this sequence of witnesses. Note that the BDD $W'_i = B_{i-1}|_{x_i=1}$ represents the function $g'_i(\vec{y}, x_{i+1}, \dots, x_m) = f_i(\vec{y}, 1, x_{i+1}, \dots, x_m)$. Furthermore, note that the result of eliminating x_i from $f_i(\vec{y}, x_i, x_{i+1}, \dots, x_m)$ in Algorithm 5 is either $f_i(\vec{y}, 0, x_{i+1}, \dots, x_m) \vee f_i(\vec{y}, 1, x_{i+1}, \dots, x_m)$, if using Shannon expansion, or $f_i(\vec{y}, f_i(\vec{y}, 1, x_{i+1}, \dots, x_m), x_{i+1}, \dots, x_m)$, if using self-substitution. In both cases $f_i(\vec{y}, 1, x_{i+1}, \dots, x_m)$ is a sub-expression, so the BDD $B_{i-1}|_{x_i=1}$ representing it was already being computed in Algorithm 5. Therefore, all we are doing is storing the intermediate result as a partial witness.

The notation $W'_i|_{x_{i+1}=W_{i+1}, \dots, x_m=W_m}$ in line 10 represents the iterative composition of all the witnesses from W_{i+1} to W_m. That is, first we compose W'_i and W_{i+1} on variable x_{i+1}, then compose the result with W_{i+1} on x_{i+2}, and so on until W_m on x_m. Since x_j itself does not appear in witness W_j, this removes the extra variables from the partial witness W'_i, obtaining the witness W_i.

```
**Data**: A BDD $B$ representing a Boolean function $f(\vec{y}, \vec{x})$, where $\vec{x} = \langle x_1, \ldots, x_m \rangle$
**Result**: A sequence of BDDs representing witnesses for $\vec{x}$, if $f$ is realizable
1  $B_0 \leftarrow B$
2  for $i \leftarrow 1$ to $m$ do
3      $W'_i \leftarrow B_{i-1}|_{x_i=1}$
4      $B_i \leftarrow$ EliminateExistentialQuantifier($i$, $W'_i$, $B_{i-1}$)
5  end
6  if $B_m$ is not the terminal node $1$ then
7      Signal error: formula not realizable
8  end
9  for $i \leftarrow m$ to $1$ do
10     $W_i \leftarrow W'_i|_{x_{i+1}=W_{i+1},\ldots,x_m=W_m}$
11 end
12 return $\langle W_1, \ldots, W_m \rangle$
```

Algorithm 7: Algorithm for synthesis based on quantifier elimination.

4.5.2 Input-first synthesis

Our second method for synthesis is designed to be applied over the input-first BDD representation of the specification. In the quantifier elimination method, BDDs were merely the chosen representation for Boolean formulas, and the algorithm was based on properties of QBFs. This method, on the other hand, is designed specifically to be used with a BDD representation, as it traverses recursively the structure of the BDD to construct witnesses to the output variables.

Recall that in an input-first BDD all input variables appear before the output variables. Also note the following lemma, following from the non-redundancy property of BDDs:

Lemma 4. *If all paths in a BDD B lead to the terminal node 0, B is the terminal node 0. Equivalently, if a BDD B is different from the terminal node 0, then it has a path leading to the terminal node 1.*

Consider next the following definitions:

Definition 4. *Let z be an arbitrary variable from a BDD B. A node v in B is said to be z-*trimmed *if v is a terminal node or it is labeled by a variable z' such that $z \leq z'$ in the ordering used by B. A node is in the* fringe of z *if it is z-trimmed and has at least one parent that is not z-trimmed.*

Fig. 4.2 illustrates these definitions. Nodes that are x_1-trimmed are shaded, and nodes in the fringe of x_1 are highlighted.

We now describe the synthesis method. This method is performed in m steps, where m is the number of output variables. We start with the input-first BDD B_1 encoding a specification $f_1(\vec{y}, \vec{x})$. Assume, without loss of generality, that the ordering of the output variables in B_1 is $x_1 < x_2 < \ldots < x_{m-1} < x_m$. On the i-th step we:

1. synthesize a witness BDD W_i for output variable x_i from B_i;

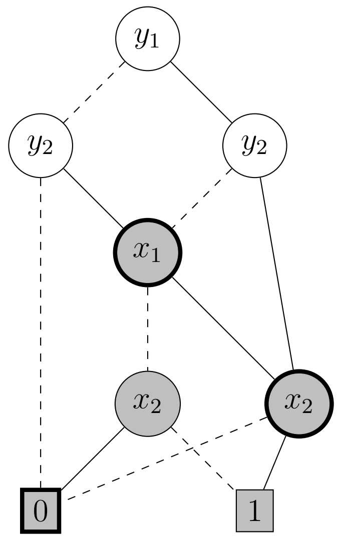


Figure 4.2: BDD showing the x_1-trimmed nodes and the fringe of x_1.

2. compose B_i with W_i on x_i to obtain B_{i+1}.

We will describe later how to specifically obtain W_i from B_i. For now, we can just assume it encodes a witness function $g_i(\vec{y})$ for x_i. Note that B_i is the result of composing the original BDD B_1 with the witnesses of all output variables until x_i. Therefore, B_i encodes a function $f_i(\vec{y}, x_i, \ldots, x_m) = f_1(\vec{y}, g_1(\vec{y}), g_2(\vec{y}), \ldots, g_{i-1}(\vec{y}), x_i, \ldots, x_m)$. Note that since all output variables $x_j < x_i$ have been removed, if there are any x_i nodes x_i is the top-most output variable in B_i. This will be important for the process of synthesizing W_i.

Now, we can state and prove the following lemma:

Lemma 5. *If for all i, $1 \leq i \leq m+1$, B_i is realizable, then $\langle g_1(\vec{y}), \ldots, g_m(\vec{y})\rangle$ are witnesses for B_1.*

Proof. Assume that all B_i are realizable. Then, $g_i(\vec{y})$ can be synthesized for all i. Recall that B_i encodes a function $f_i(\vec{y}, x_i, \ldots, x_m) = f_1(\vec{y}, g_1(\vec{y}), g_2(\vec{y}), \ldots, g_{i-1}(\vec{y}), x_i, \ldots, x_m)$. Therefore, B_{m+1}, the BDD produced when the last witness W_m is composed, encodes a function $f_{m+1}(\vec{y}) = f_1(\vec{y}, g_1(\vec{y}), \ldots, g_m(\vec{y}))$. Since B_{m+1} is realizable, $\forall \vec{y}(f_{m+1}(\vec{y}))$ is true. This means that $f_{m+1}(\vec{\nu}) = 1$ for any assignment $\vec{\nu}$. Replacing f_{m+1} by its definition, we have that $f_1(\vec{\nu}, g_1(\vec{\nu}), \ldots, g_m(\vec{\nu})) = 1$ for any $\vec{\nu}$, and so $\langle g_1(\vec{y}), \ldots, g_m(\vec{y})\rangle$ are witnesses to B_1. $\square$

Lemma 5 means that, as long as B_1 is realizable and we construct W_i from B_i such that $B_{i+1} = B_i|_{x_i = W_i}$ is realizable, $\langle W_1, \ldots, W_m\rangle$ will be correct witness BDDs. Keeping this in mind, we next describe how to obtain W_i from B_i.

Recall that x_i is the top-most output variable in B_i. This means that all nodes above the fringe of x_i are input nodes. To synthesize W_i, we will replace every node v in the fringe of x_i with a terminal node representing an assignment to x_i, as follows:

1. If v is the terminal node 0, B_i is not realizable, and so the process terminates with an error.

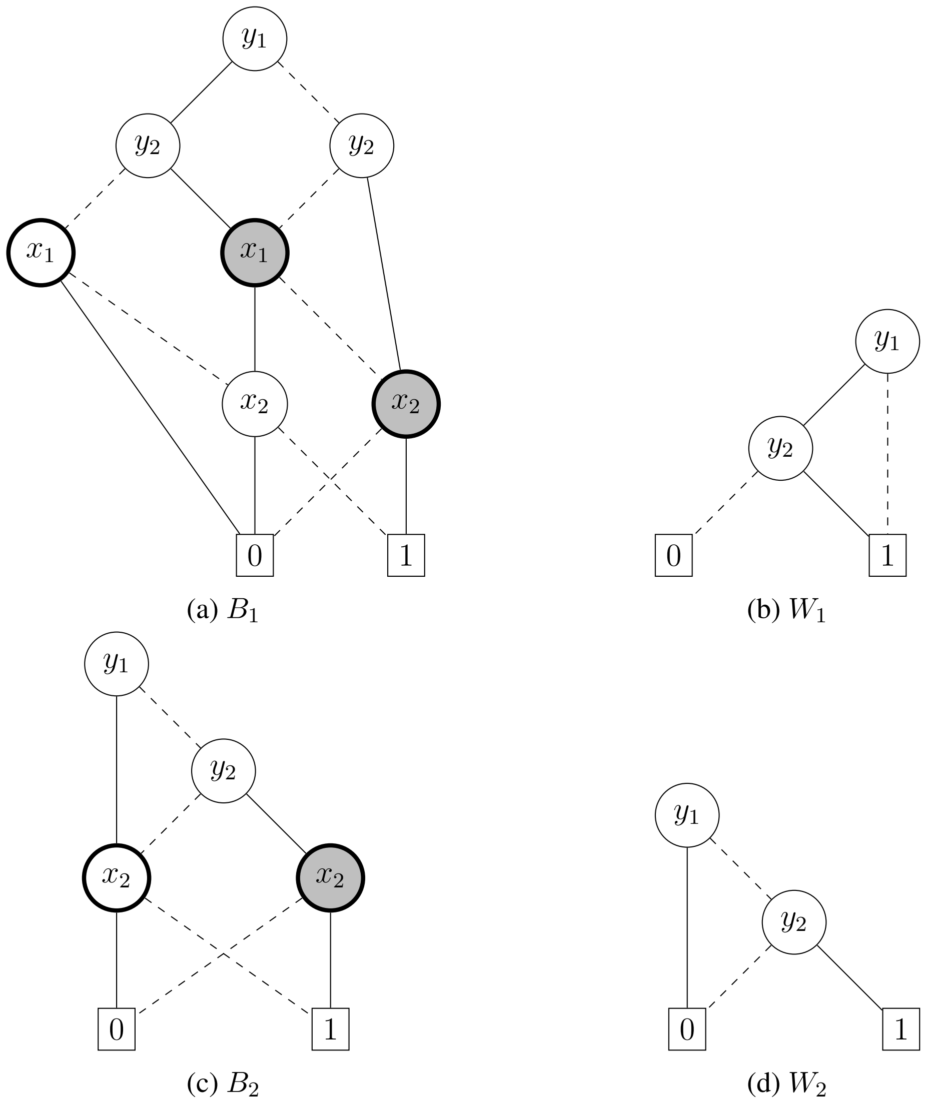


Figure 4.3: Example of the structural synthesis algorithm.

2. If v is a variable node labeled by x_i and its positive child is the terminal node 0, replace it by the terminal node 0.
3. Otherwise, replace v by the terminal node 1.

We call this operation *trimming* of x_i. Note that this removes all x_i-trimmed variable nodes. Since x_i is the top-most output variable, the resulting BDD W_i contains only input variables, and so it encodes a function $g_i(\vec{y})$. Fig. 4.3 shows a complete example of the algorithm applied to a BDD with two output variables. In the BDD B_1, highlighted nodes are to be replaced by terminal nodes: gray nodes by the terminal node 1 and white nodes by the terminal node 0. After these nodes are replaced and the BDD reductions are applied, we obtain the BDD W_1. Composing B_1 and W_1 on variable x_1 produces the BDD B_2. Note that this is equivalent to replacing each x_1 node in B_1 by its positive child if it was previously replaced by the terminal node 1, or its negative child if it was previously replaced by the terminal node 0. Finally, W_2 is obtained from B_2 by again replacing the highlighted nodes by the appropriate terminal nodes.

We will now prove that this method produces correct witnesses. Note that what we are doing is assigning x_i to 1 except when this would lead to the terminal node 0. Let v' be the child of v that this assignment leads to. Since it is different from the terminal node 0, by Lemma 4 a path from v' to the terminal node 1 always exists. This means that for any assignment to $\vec{y}$, given the assignment produced by the witness $g_i(\vec{y})$, there exists an assignment to $x_{i+1}, \dots, x_m$ that satisfies $f_i(\vec{y}, g_i(\vec{y}), x_{i+1}, \dots, x_m)$. This means that $\forall\vec{y}\exists x_{i+1} \dots x_m(f_i(\vec{y}, g_i(\vec{y}), x_{i+1}, \dots, x_m))$ is true. Since $f_{i+1}(\vec{y}, x_{i+1}, \dots, x_m) = f_i(\vec{y}, g_i(\vec{y}), x_{i+1}, \dots, x_m)$, we can rewrite this as $\forall\vec{y}\exists x_{i+1} \dots x_m(f_{i+1}(\vec{y}, x_{i+1}, \dots, x_m))$, which means f_{i+1} is realizable. This allows us to make the following claim:

Lemma 6. *If W_i is synthesized from B_i by trimming x_i, and B_i is realizable, then B_{i+1} is realizable as well.*

Note that, since if B_i is realizable B_{i+1} will be as well, the case in which v is the terminal node 0 can only happen in the first step, if B_1 is not realizable. This means that this method will detect if the specification is not realizable in the very beginning, before any witness is synthesized. This is in practice the same method to check realizability from Algorithm 6.

From Lemmas 5 and 6, we obtain the following corollary:

Corollary 2. *If B_1 is realizable, $\langle g_1(\vec{y}), \dots, g_m(\vec{y})\rangle$ are witnesses for B_1.*

This proves that, given a realizable implementation, this method will synthesize a correct implementation. The method is presented in full in Algorithm 8, using supporting operations in the models presented in Algorithm 4 to perform BDD traversal.

```
Data: An input-first BDD $B$ representing a Boolean function $f(\vec{y}, \vec{x})$
Result: A sequence of BDDs representing the witnesses for $\vec{x}$
1  Function Synthesis($B$)
2      $B_1 \leftarrow B$
3      for $i \leftarrow 1$ to $m$ do
4          $W_i \leftarrow$ Trim_Init($x_i$, $B_i$)
5          $B_{i+1} \leftarrow B_i|_{x_i=W_i}$
6      end
7      return $\langle W_1, \dots, W_m\rangle$
8  Function Trim($x_i$, $B$)
9      if $B$ is an input variable node $(y_j, B_0, B_1)$ then
10         return MkBDD($y_j$, Trim_Rec($x_i$, $B_0$), Trim_Rec($x_i$, $B_1$))
11     else if $B$ is the terminal node $0$ then
12         Signal error: formula not realizable
13     else if $B$ is an $x_i$ variable node and its positive child is the terminal node $0$ then
14         return BDDZero()
15     else
16         return BDDOne()
```
Algorithm 8: Algorithm for synthesis using input-first BDDs.

Default 1 Note that, while in this algorithm we replace a node v in the fringe of x_i for the terminal node 1 whenever this would not lead to an unsatisfying assignment, there are cases when it would be equally acceptable to replace it by the terminal node 0. This is the case when v is already the terminal node 1, or when it is a variable node x_j with $j \neq i$.

Since in this case the variable x_i was skipped, its assignment is irrelevant. This is also the case when v is a x_i node but neither of its children is the terminal node 0. In this case there would be a satisfying assignment to the other variables regardless of the assignment to x_i, but this assignment would be different in each case. The choice of using the terminal node 1 instead of 0 was mostly arbitrary, but there is an advantage to always defaulting to the same terminal node in that if there is a greater number of paths leading to the same terminal node, the chance of reductions by non-redundancy increases, producing smaller BDDs. Finding more efficient heuristics for choosing which terminal node to replace a node with is a matter of future work.

5 EXPERIMENTAL EVALUATION AND RESULTS

When designing our experiments, we had several goals in mind. One of the most important questions was how our methods for synthesis would scale with the size of the input and output, since it is often hard to predict how much the sizes of BDDs will increase with the number of variables. Another question was how much effect the ordering of the BDD would have on its size and, consequently, on the performance of the algorithms. Finally, we wished to see how our different methods compared with each other.

The results presented here will serve to paint a general picture of the synthesis process using BDDs, providing a guidance for the future direction of this work. Because it is difficult to perform a theoretical analysis of algorithms based on BDD manipulation, the practical results obtained through these experiments are critical in deciding future paths of research.

5.1 Test cases

Since we wished to know how the performance of our methods varied with the number of variables, we had to employ test cases that could be implemented with an arbitrary size. The natural choices for functions of this type were operations over integers in binary. Besides being possible to implement with integers in any size, these operations are widely used and are often the subject of synthesis research.

In order to see the synthesis process at work, we also needed to select functions that had an intuitive declarative definition, that is, a definition that does not directly describe how the function would be computed. This definition can then be used as the specification f to serve as the basis for the synthesis of the actual implementation of the function. Considering these requirements, we selected five problem instances to use for our experiments, as shown in Table 5.1. In it, x, y and y' are n-bit integers, represented each in the BDD by groups of n Boolean variables. x consists of output variables, while y and y' are composed of input variables split into two different integers.

The relational operators over integers $\leq$, $\geq$, $=$ were encoded as Boolean expressions over the variables representing them. Assuming an integer z is represented by the vector of variables $\langle z_n, z_{n-1}, \ldots, z_2, z_1 \rangle$, where z_n represents the most significant and z_1 the least significant bit, the relational operators are encoded as follows:

- $z = z'$: $\varphi = \bigwedge_{i=1}^{n}(z_i \leftrightarrow z'_i)$
- $z \leq z'$: φ_n, where $\varphi_i = (\neg z_i \wedge z'_i) \vee ((z_i \leftrightarrow z'_i) \wedge \varphi_{i-1})$ and $\varphi_0 = 1$

	Desired implementation	Specification
Subtraction	$x = y' - y$	$x + y = y'$
Min	$x = min(y, y')$	$(x \leq y) \wedge (x \leq y') \wedge ((x = y) \vee (x = y'))$
Max	$x = max(y, y')$	$(x \geq y) \wedge (x \geq y') \wedge ((x = y) \vee (x = y'))$
Average (floor)	$x = \lfloor \frac{y + y'}{2} \rfloor$	$(x + x = y + y') \vee (x + x + 1 = y + y')$
Average (ceiling)	$x = \lceil \frac{y + y'}{2} \rceil$	$(x + x = y + y') \vee (x + x = y + y' + 1)$

Table 5.1: Instances used in the experiments with their definitions.

- $z \geq z'$: φ_n, where $\varphi_i = (z_i \wedge \neg z'_i) \vee ((z_i \leftrightarrow z'_i) \wedge \varphi_{i-1})$ and $\varphi_0 = 1$

The $+$ operator is more complex, and since it is an operation that returns an integer rather than a Boolean it cannot be implemented as a single Boolean formula. Rather, it produces n formulas $\varphi_n, \ldots, \varphi_1$ representing a new integer, which can be later combined into a single formula through one of the relational operators above. The encoding for addition should be easily recognizable by those familiar with addition in binary:

$$\varphi_i = z_i \oplus z'_i \oplus c_{i-1}$$
$$c_i = (z_i \wedge z'_i) \vee (z_i \wedge c_{i-1}) \vee (z'_i \wedge c_{i-1})$$

In this encoding, c_i represents the carry-out from the addition in the i-th position. c_0, the carry-in for the first position, is normally 0, but can be set to 1 to add an extra term of 1 to the sum, which is useful in the formulas for average.

Note that in order for the formula for subtraction to be realizable for n-bit integers, $+$ is interpreted as addition modulo n, or alternatively addition with the possibility of overflow. On the other hand, in the formulas for average we need the result of the addition with an extra bit added if necessary. This extra bit can be obtained by simply taking c_n. Therefore the comparisons in these formulas are actually performed over $(n + 1)$-bit integers.

5.2 Orderings

In order to analyze the impact of the ordering on the size of the BDDs and the performance of the algorithms, we executed the experiments using different orderings and compared their results.

The first ordering employed was the input-first ordering already described in Chapter 4, which specifies that all input variables must come before the output variables.

To compare to the input-first ordering, we defined an alternative ordering which we call *fully-interleaved*. In this ordering, output variables are interspersed with input variables, such that variables in the same position are close together in the ordering. This seems like a good ordering to use for operations over integers in binary, because the value of a bit in a given position of the output often depends more strongly on the values in the same position in the input. This keeps dependencies localized inside the BDDs.

	Input-first	Fully-interleaved
Subtraction/Average	$(y_1 < \ldots < y_n < y'_1 < \ldots < y'_n < x_1 < \ldots < x_n)$	$(y_1 < y'_1 < x_1 < \ldots < y_n < y'_n < x_n)$
Min/Max	$(y_n < \ldots < y_1 < y'_n < \ldots < y'_1 < x_n < \ldots < x_1)$	$(y_n < y'_n < x_n < \ldots < y_1 < y'_1 < x_1)$

Table 5.2: Ordering of the variables for each instance.

When choosing one of these two orderings we define how input and output variables are positioned in relation to each other. However, there is still a choice to be made in how input and output variables are positioned internally among themselves. For example, the ordering can follow an increasing or decreasing order of indices. Examining the individual test cases, we see that the best choice depends on the instance. First, note that placing variables that have wider influence on the formula closer to the top tends to produce smaller BDDs, since assigning these variables early will simplify several sub-expressions at the same time. Note that for *subtraction* and *average*, the variables with lower index have wider influence, because in the addition operation the value of the most significant bits depends on the carry of the least significant bits. For *max* and *min*, on the other hand, the variables with higher index have wider influence, because the greater of two integers in binary is decided by the most significant position on which they differ. Therefore, we chose to use a decreasing order for *max* and *min*, and and increasing order for the rest of the instances. The resulting orderings are shown on Table 5.2.

Apart from these specific orderings we also generated 100 randomly-ordered BDDs for each value of n we employed, so that in each experiment we could compare the average results for the random orderings with the input-first and fully-interleaved results.

5.3 Experimental setup

The synthesis algorithms were implemented in C++11 using the CUDD library for working with BDDs (SOMENZI, 2012). The experiments were executed remotely in the DAVinCI cluster at Rice University, which allowed them to be run in parallel. The cluster consists of 192 Westmere nodes of 12 processor cores each, running at 2.83 GHz with 4 GB of RAM per core, and 6 Sandy Bridge nodes of 16 processor cores each, running at 2.2GHz with 8 GB of RAM per core.

For each of the five test cases we constructed BDDs encoding their specification, varying the number of bits n of the arguments. Next, we ran the methods for synthesis presented in Chapter 4 over these BDDs and collected data on their performance and output. This was done using the input-first, fully-interleaved and random orderings. The method for input-first synthesis was naturally applied only to the BDDs following this ordering, while the method based on quantifier elimination was applied to all orderings. In order to observe how self-substitution compares to Shannon expansion, the synthesis through quantifier elimination was performed twice over each BDD, once using each technique.

As could be expected, the *min* and *max* instances produce very similar results, as do *average (floor)* and *average (ceiling)*. Therefore in the results presented below we will show only the results for *subtraction*, *max* and *average (floor)*, and *min* and *average (ceiling)* can be assumed to be similar.

5.4 BDD construction

In order to run our synthesis methods, we first need to have the specification represented in the format of a BDD. Therefore, our first concern is with the construction of this initial BDD. There are two factors we must be aware of, the size of the BDD and the time taken to construct it. Recall that BDD size is often hard to predict, and can easily lead to a combinatorial explosion. Therefore, if the BDD representation of the specification is too large, we might run out of memory before even starting the synthesis. At the same time, we also must consider the time required to construct the initial BDD, because even if the synthesis process is efficient this will be of little benefit if preparing the specification BDD is too time-consuming.

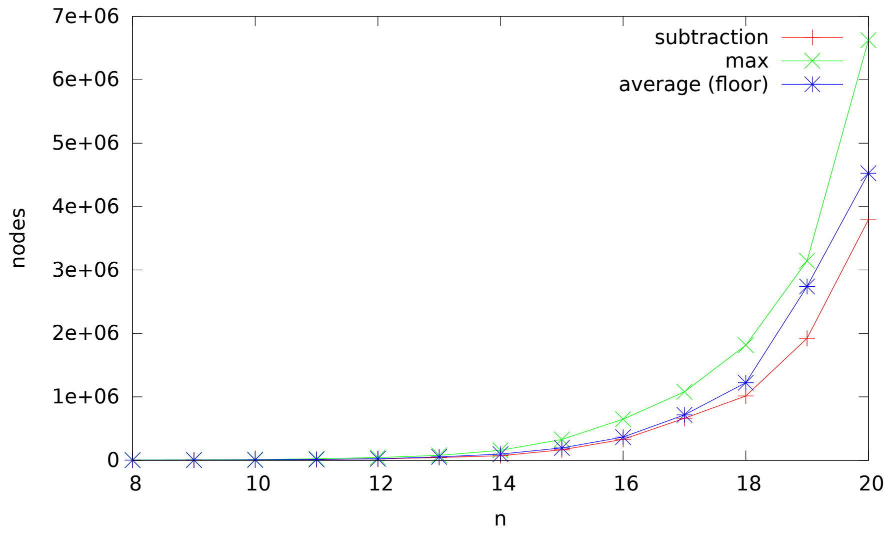
We measured the number of nodes and construction time of the initial BDD for each instance and ordering, seeing how these scale as n increased. We provide detailed analyses below.

5.4.1 Random orderings

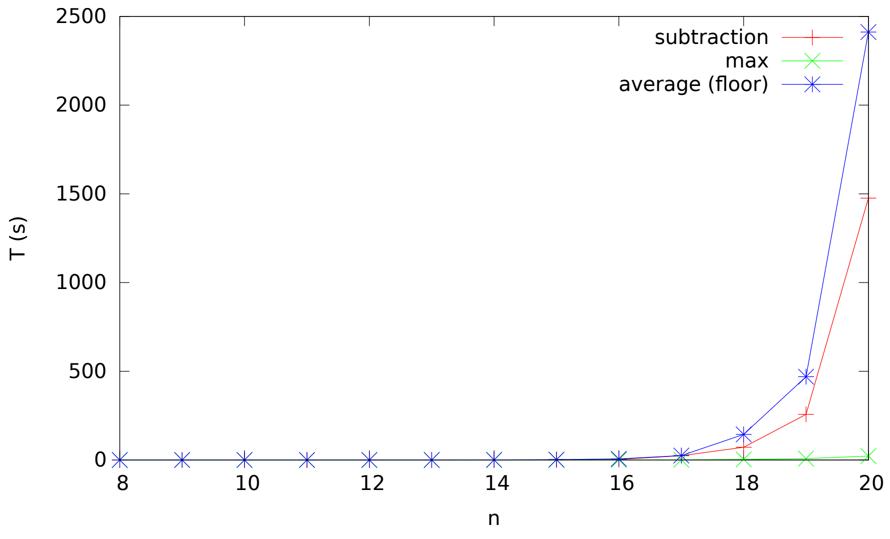
Fig. 5.1 shows, for the *subtraction*, *max* and *average (floor)* instances, the median results of the 100 random BDDs for each n from 8 to 20. We chose the median instead of the mean because the distribution of the results of the randomly-ordered BDDs was seen to have a long tail, with a small number of cases having far larger sizes and construction times than the majority. This results in the value for the mean being significantly larger than most of the individual values, but has little influence in the median, therefore we considered the latter to be more representative.

Both size and construction time show an exponential behavior as n increases, producing for higher n BDDs with millions of nodes and taking up to 40 minutes to be constructed. This demonstrates how quickly BDDs can grow out of control, and that it is not easy to pick a good ordering for a BDD at random.

The case for *max* calls for special consideration. Note that the construction time takes longer to increase than for the other instances, even though the BDD produced is on average larger. For larger n *max* would probably exhibit a more pronounced growth like the other instances, but experiments verifying this could not be performed because the size of the BDD became too large. This slower growth can be explained by examining the encoding of the specifications as Boolean functions. Note that the sub-expressions $(x = y)$, $(x = y')$, $(x \geq y)$ and $(x \geq y')$ that form the specifications for *max* only compare two n-bit integers at a time. The encoding of the relational operators as Boolean formulas is iterative, so that comparing i bits takes i steps. Therefore, each of these sub-expressions will be constructed in n steps, with step i producing a BDD with $2i$ variables. Only in the end there will be a single step that combines the BDDs for each sub-expression into a final BDD with $3n$ variables. On the other hand, for the other instances each sub-expression uses all three n-bit integers. This means that the i-th step of a sub-expression produces a BDD with $3i$ variables. Therefore the intermediary operations have to deal with much larger BDDs, causing them to take longer.



(a)



(b)

Figure 5.1: Size and construction time of the initial BDD according to the instance.

5.4.2 Input-first

Next, we compare the average results using random orderings with the results for the construction of the input-first BDDs. Fig. 5.2 shows for the three instances the construction time of the input-first BDD in comparison to the median construction time of the random instances.

In all three cases it can be seen that the time needed to construct the BDD increases much faster when using the input-first ordering. In fact, because of this we were unable to run the experiments for this ordering up to $n = 20$ due to time and memory constraints. This confirms that, although this is an intuitive ordering to use, making reasoning about synthesis easier, it can be very inefficient in practice.

Notice that differently from the random orderings, the construction time of the input-first BDD for *max* surpasses the other instances as n grows. This happens because, although the same reasoning of the BDDs for the sub-expressions only having $2n$ variables applies, the bottleneck when using this ordering lies on the BDDs of $3n$ variables. Consider the BDD for the expression $((x = y) \vee (x = y'))$. This BDD has $3n$ variables, and tests if the output variables have the same values as either group of input variables. However, due to the input-first ordering, all input variables are on the top of the BDD, with the output variables on the bottom, and so in order to perform this comparison the BDD must memorize the value of the input before checking the output. To do this, the BDD needs one path through the input variables for each possible assignment of y and y', totaling $2^{2n} = 4^n$ different paths. This means that the section of the BDD for the input variables will be as large as possible. Afterwards, with the conjunction of the $(x \geq y)$ and $(x \geq y')$ sub-expressions, this BDD becomes smaller, but having to construct this massive BDD affects greatly the total construction time.

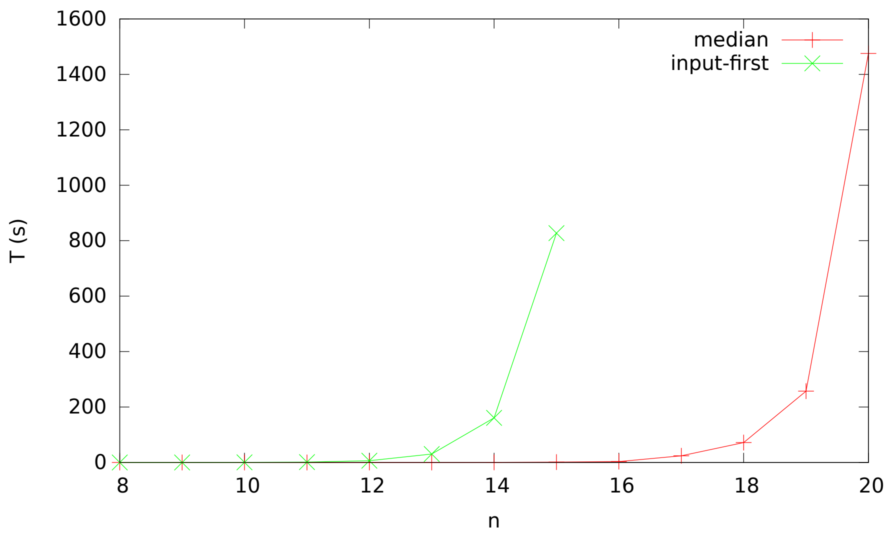
Fig. 5.3 shows the size of the initial input-first BDD for each instance. The graph also shows for comparison the median size of the BDD for *max* using the random orderings. As expected, the input-first BDDs show a faster growth as n increases, although not as pronounced as the growth in construction time.

5.4.3 Fully-interleaved

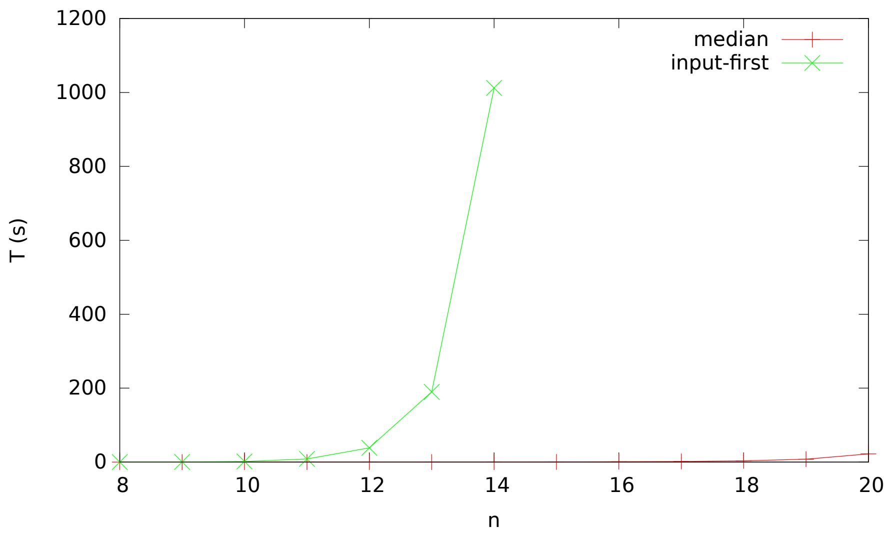
We expected the fully-interleaved ordering to be the one to perform best for the test cases used in these experiments. Not only that was the case but both the size of the fully-interleaved BDDs and the time taken to construct them were several orders of magnitude smaller than for the other orderings. In fact, instead of the exponential growth shown by the other orderings, the fully-interleaved BDDs showed linear growth, which allowed us to construct BDDs for much larger n when using this ordering.

Fig. 5.4 shows the size of the fully-interleaved BDDs for each instance from $n = 8$ to $n = 128$, with a very clear linear growth. Even for $n = 128$ the size of the BDD was under 2000 nodes for all three instances. Compare this to the random orderings, where the BDDs surpassed one million nodes as early as $n = 18$.

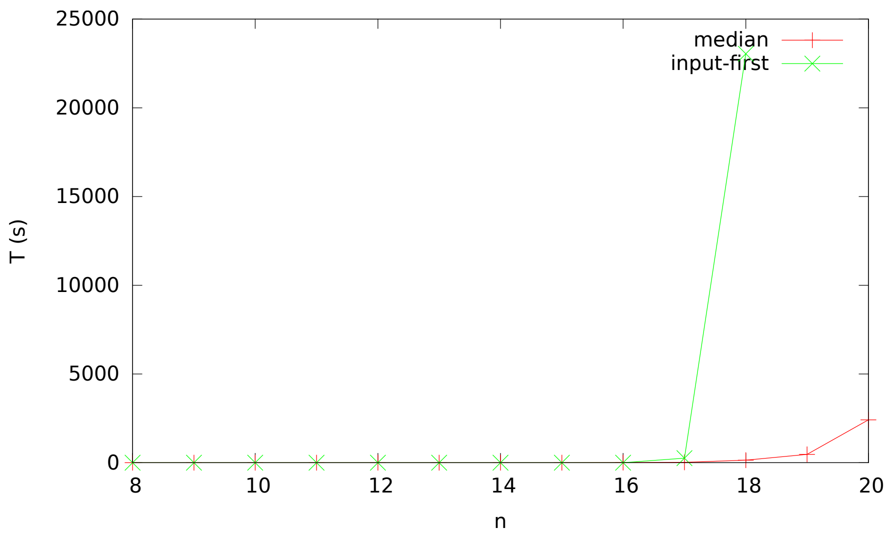
As a consequence of the smaller BDDs, construction time was also much faster. For all instances for n up to 128, all constructions took under 0.05s.



(a) Subtraction



(b) Max



(c) Average (floor)

Figure 5.2: Comparison of the construction time of the input-first BDDs with the average of the random orderings.

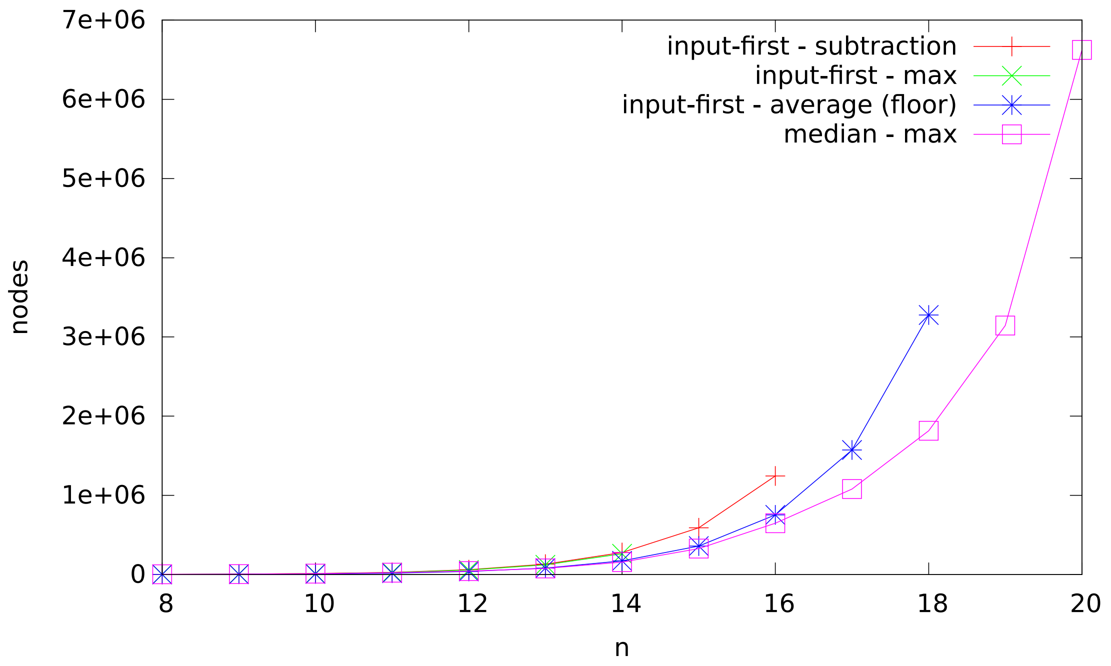


Figure 5.3: Size of the initial BDD using the input-first ordering.

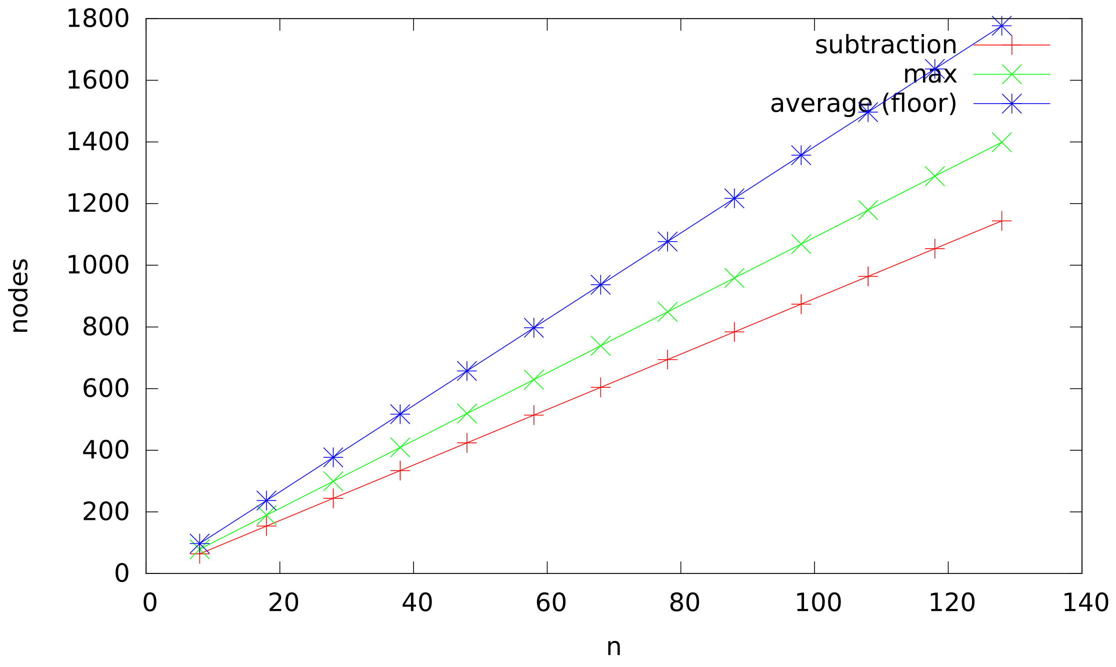


Figure 5.4: Size of the initial BDDs using the fully-interleaved ordering.

5.5 Synthesis

After having analysed the construction of the specification BDD, we move on to the synthesis process itself. In order to compare the different methods for synthesis, we measure both the running time of the method and the size of the multi-rooted BDD produced by it. For all orderings we ran the method based on quantifier elimination using both Shannon expansion and self-substitution, in order to compare the two. For the input-first ordering we also ran the method exclusive to this ordering, and compared its results with the quantifier elimination method. We also compared results between the different orderings.

5.5.1 Synthesis via quantifier elimination

Fig. 5.5 shows for each instance the running time of the quantifier elimination method using Shannon expansion and self-substitution. Each graph shows the results for the input-first BDD, as well as the median for the randomly-ordered BDDs.

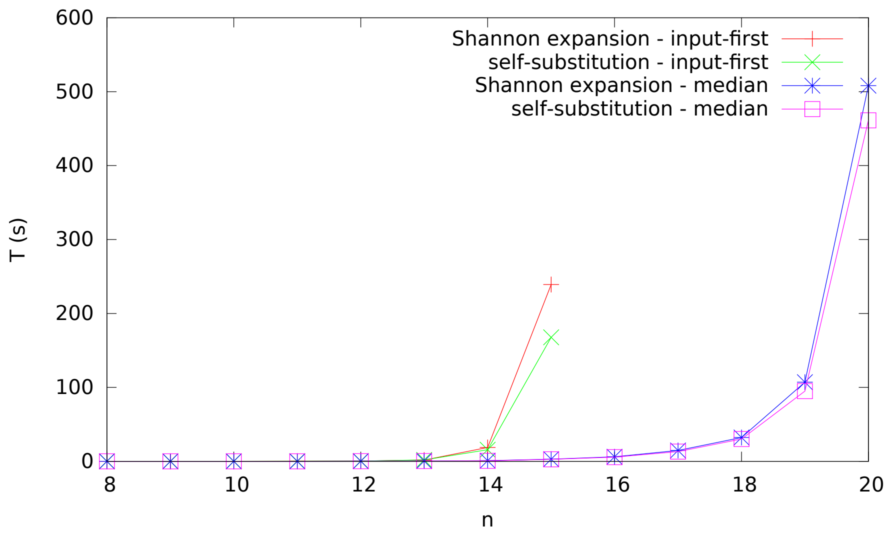
Unsurprisingly, just like in the construction of the initial BDDs, the running time of the synthesis algorithms increased much faster for the input-first BDDs. What is notable however is that when comparing for the same ordering the quantifier elimination method using Shannon expansion and using self-substitution we do not see a large difference. Self-substitution performed better for *subtraction* and *average (floor)*, and Shannon expansion for *max*. This suggests that self-substitution is a competitive technique for quantifier elimination, but for the purposes of synthesis the difference between them is mostly insignificant when compared to the difference between orderings. This suggests that any significant difference in performance between the two techniques is dominated by the total running time of the synthesis.

Fig. 5.6 shows the synthesis running time for each instance using the fully-interleaved ordering. Just as before, the results were far below those of the other orderings even for much larger n, and show a polynomial growth instead of exponential.

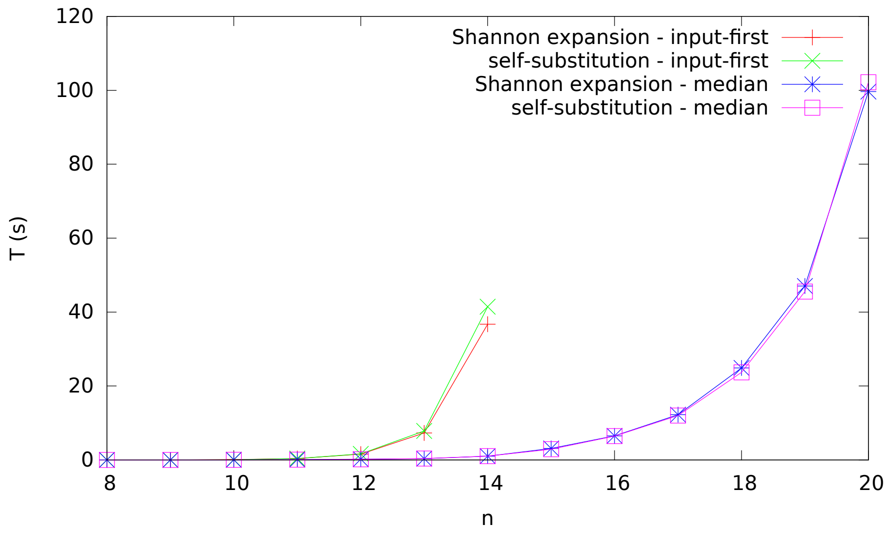
5.5.2 Input-first synthesis

Now we compare the running time between the general method based on quantifier elimination and the method exclusive to input-first BDDs. Fig. 5.7 shows the results for each instance. Since quantifier elimination performs similarly for self-substitution and Shannon expansion, we plotted only the results for self-substitution, which were in most cases faster. In all three cases, the method exclusive for input-first BDDs performed better than quantifier elimination. One reason for this might be that it synthesizes witnesses directly, while when using quantifier elimination the method first produces partial witnesses, from which the extra variables must later be eliminated through composition.

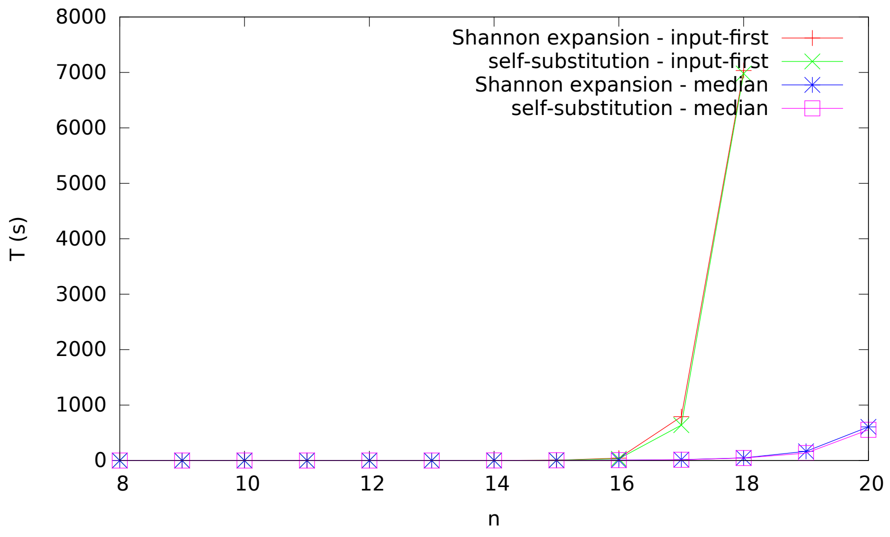
Although this seems to be a positive result for the input-first method, Fig. 5.8 shows that this is not an absolute victory. In it we can see that the specific method for input-first BDDs is outperformed by the general method using a random ordering. This suggests that using a more efficient synthesis method will not necessarily compensate for an inefficient ordering. This is an important result that shows that when synthesizing Boolean functions with BDDs it might be more important to find a good ordering than optimizing the synthesis itself. Nevertheless, for functions for which the input-first ordering works



(a) Subtraction

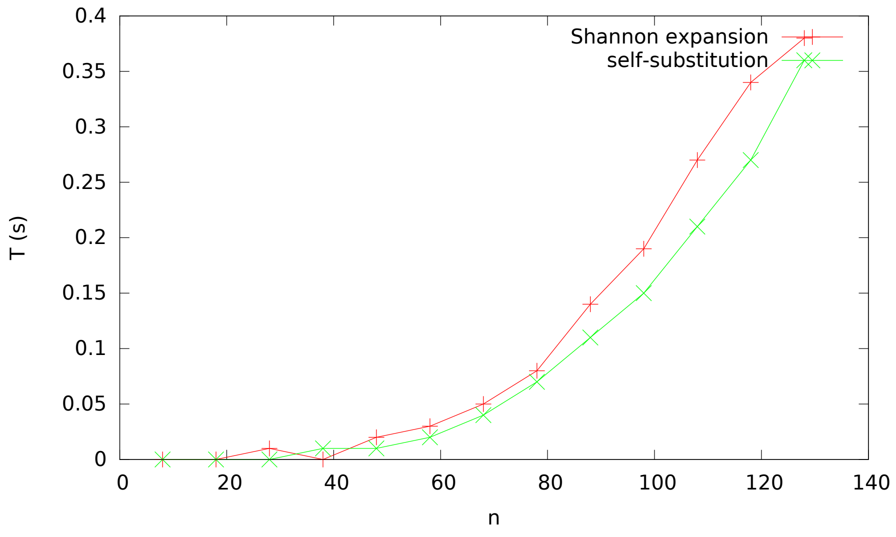


(b) Max

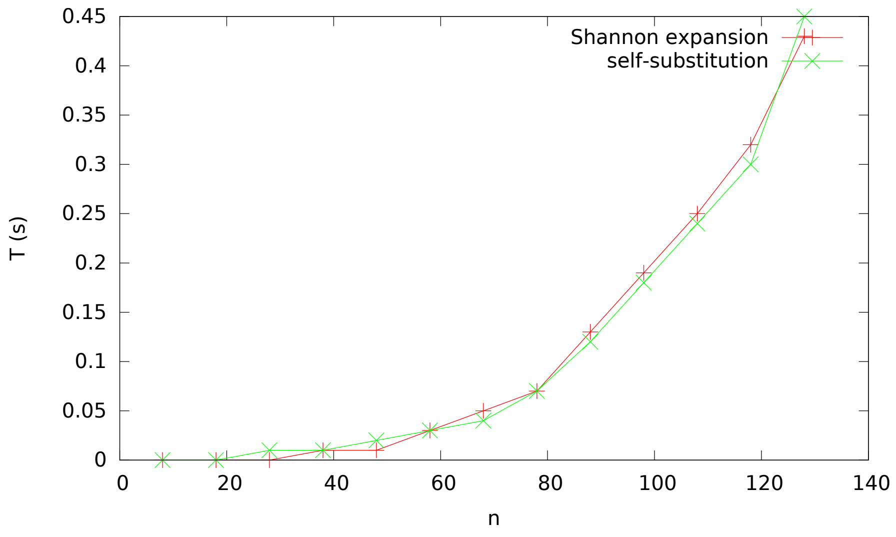


(c) Average (floor)

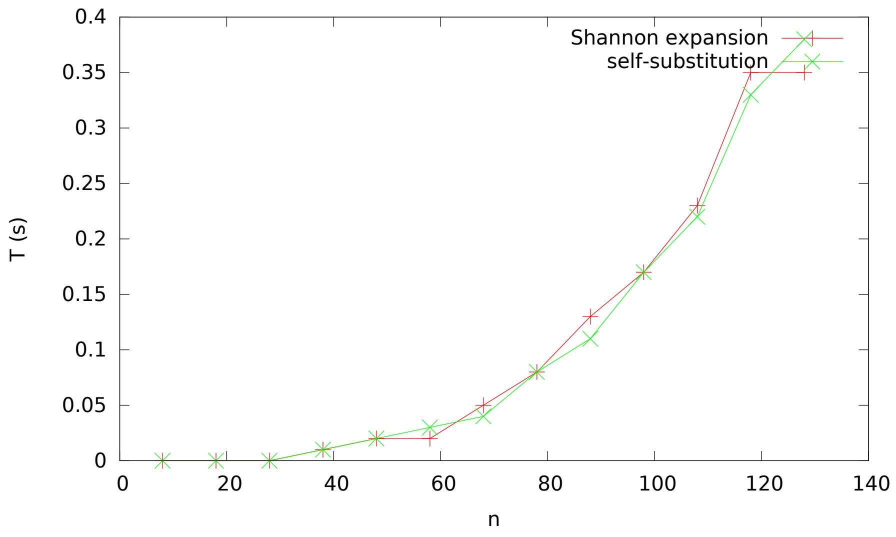
Figure 5.5: Running time of each synthesis algorithm for different instances.



(a) Subtraction

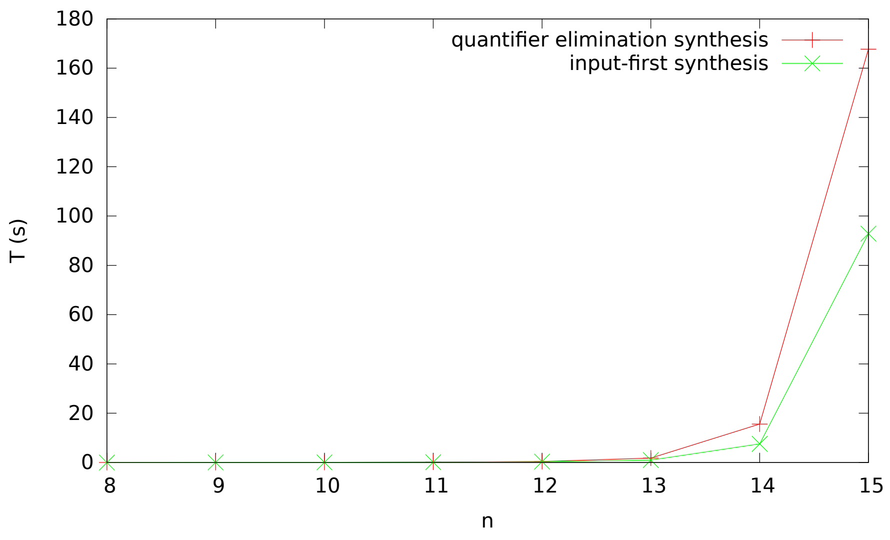


(b) Max

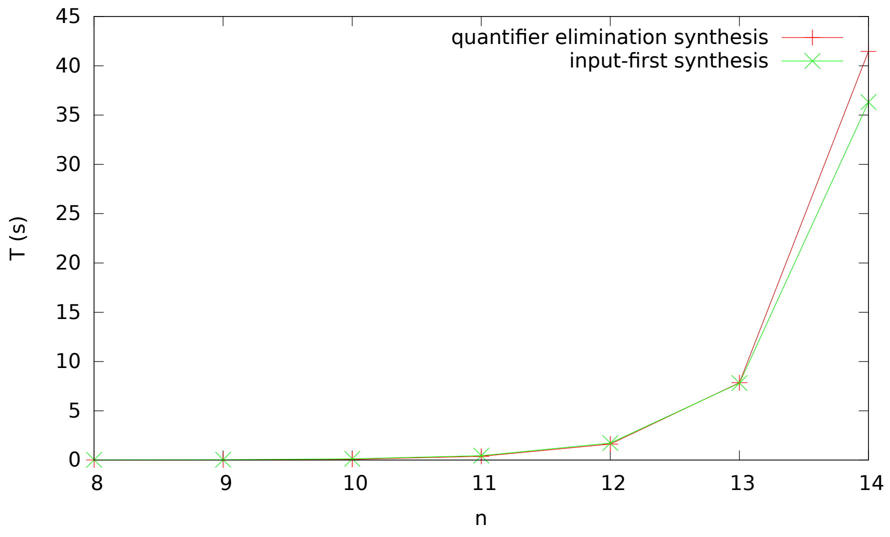


(c) Average (floor)

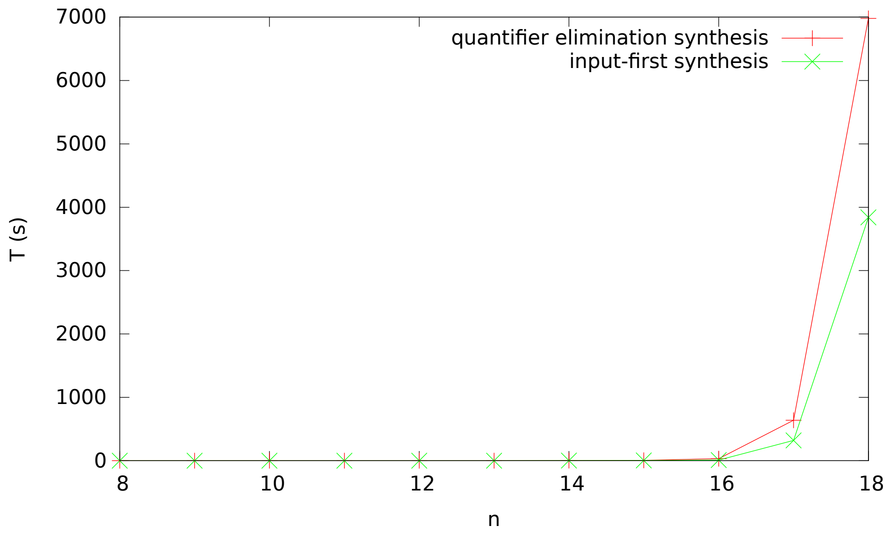
Figure 5.6: Running time of the synthesis for each instance, using the fully-interleaved ordering.



(a) Subtraction



(b) Max



(c) Average (floor)

Figure 5.7: Running time comparison of the different synthesis methods for input-first BDDs.

better, these results might justify the choice of using the method specific to this ordering rather than the general method using quantifier elimination.

Also note that, regardless of the ordering, for larger BDDs the running time for synthesis is significantly smaller than the construction time of the initial BDD. This suggests that the construction step is actually the bottleneck in the synthesis process. This places further importance in using efficient orderings to obtain smaller BDDs in place of trying to develop more efficient synthesis methods.

5.5.3 Implementation size

Running time is not the only important parameter in analysing the synthesis methods. It is also important to consider the size of the multi-rooted BDD produced, since this is a determinant of the size of the circuit or program that will implement the synthesized function. In fact, in certain cases having smaller synthesized functions is more important than having shorter synthesis time. For example, if the synthesized BDD is converted to a circuit, the synthesis process will run only once and on software, while the circuit will be implemented with physical components and be used possibly millions of times, making its size and efficiency a priority. However, one must keep in mind that although a BDD can be directly mapped to a circuit or program of equivalent size, techniques for Boolean simplification may be able to convert them to a much smaller implementation. Therefore, the size of the synthesized BDD should not be considered a final indication of the size of the implementation.

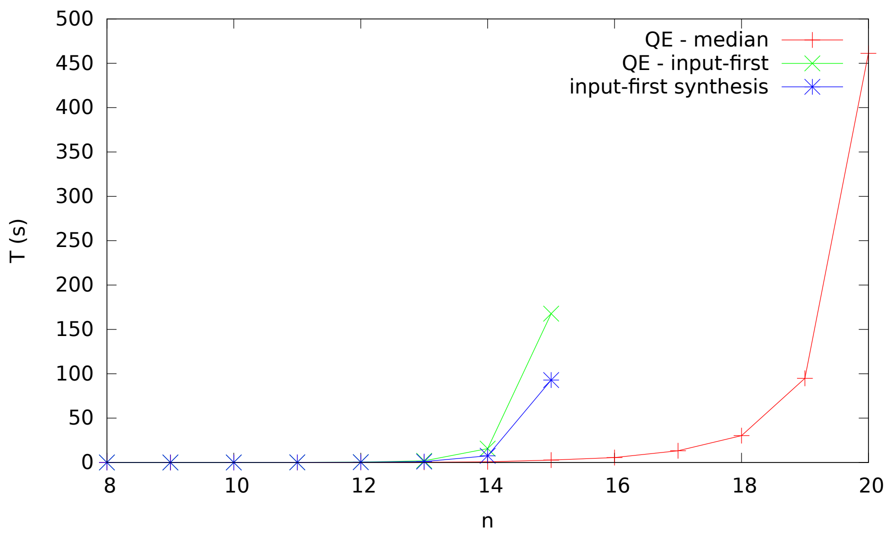
As it turns out, the BDDs synthesized by either method are the same size. This is logical because for the instances used in the experiments there is a single satisfying output for each possible input, therefore the synthesized functions must be equivalent between the methods. Since BDDs are a canonical representation of Boolean functions, equivalent BDDs are identical. On the other hand, the results between the methods may differ when synthesizing different functions with multiple possible outputs for the same input.

However, we can still compare between different orderings. Fig. 5.9 compares for each instance the size of the synthesized BDD using the input-first ordering with the median size when using the random orderings. Note that the difference in size of the synthesized BDDs is proportionally larger than the difference between the initial BDDs, giving further justification to use efficient orderings. Fig. 5.10 adds to this point by showing that the size of the synthesized BDDs using the fully-interleaved ordering is several times smaller even for much larger n.

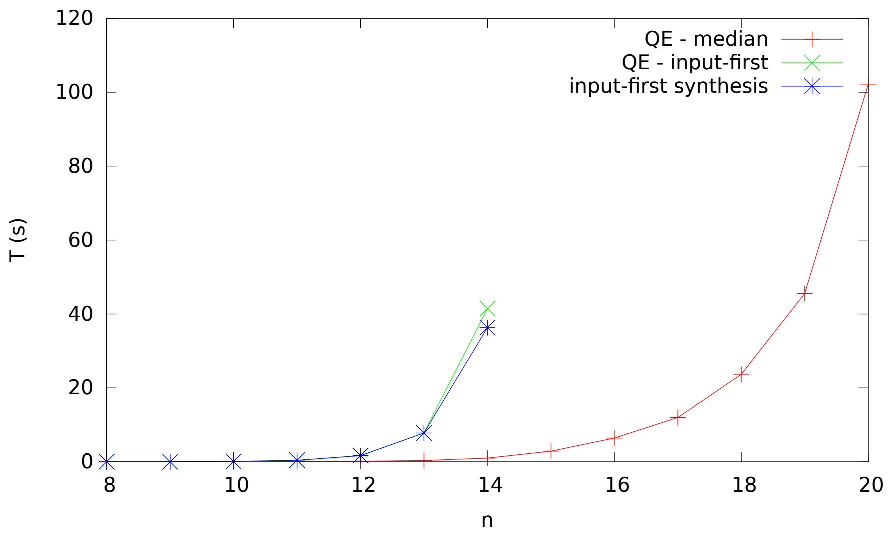
5.6 Final remarks

The experiments described in this chapter allowed us to obtain a general notion of the feasibility of the methods presented here for synthesis of Boolean functions, and granted us some insights on how to better make use of BDDs as an underlying data structure for synthesis.

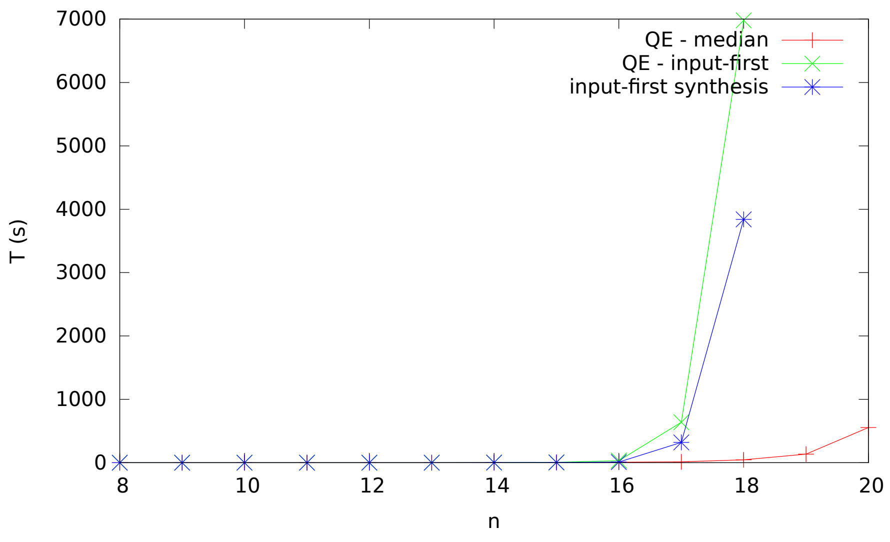
One of the most clear conclusions we can take from these results is that the ordering used for the BDDs has a massive impact on scalability. Both for construction of the initial BDD and the synthesis process itself, the ordering used can make the difference on whether the process will take milliseconds or hours. This can be easily seen when



(a) Subtraction

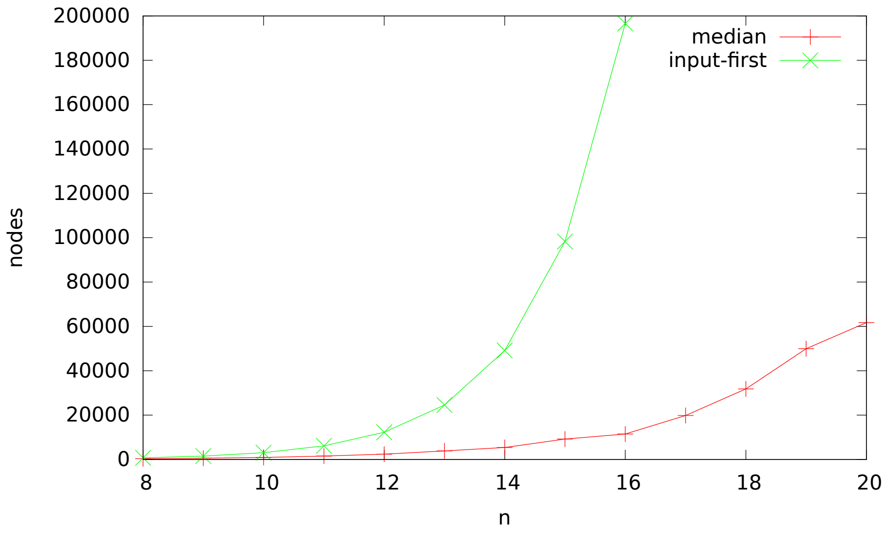


(b) Max

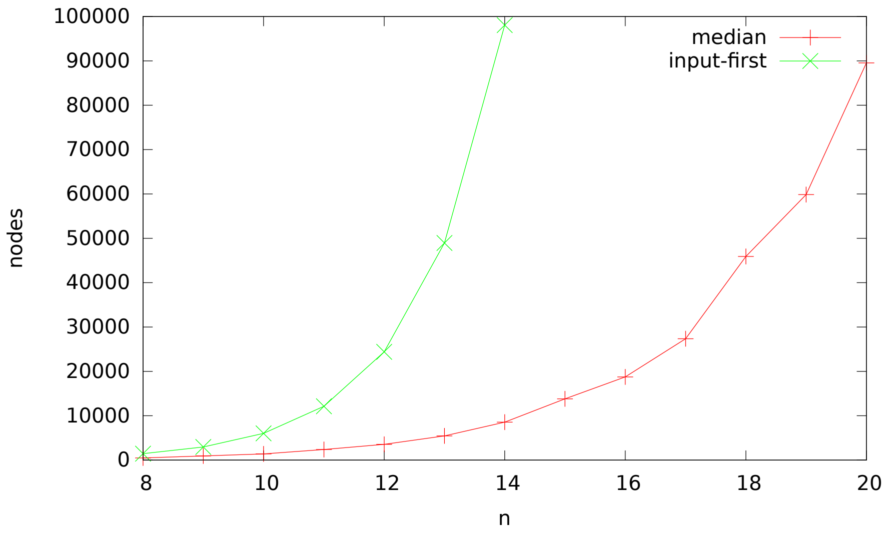


(c) Average (floor)

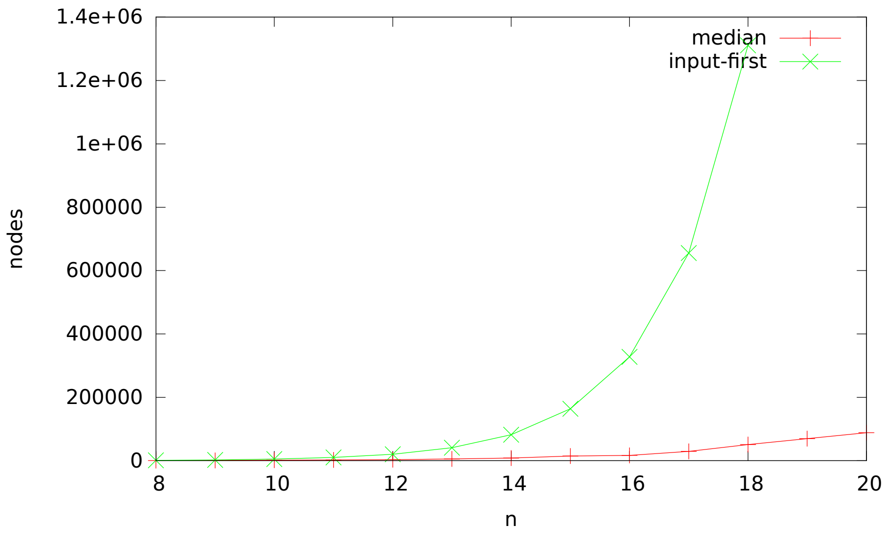
Figure 5.8: Comparison of synthesis running time between input-first and the random orderings.



(a) Subtraction



(b) Max



(c) Average (floor)

Figure 5.9: Comparison of the size of the synthesized BDD between input-first and the random orderings.

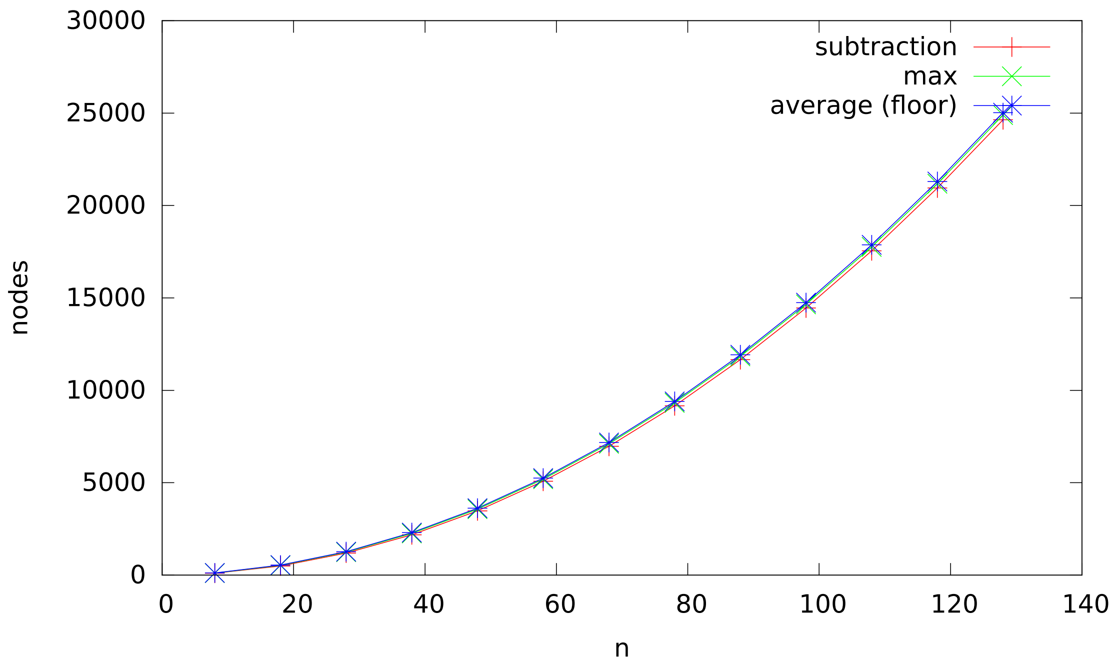


Figure 5.10: Size of the synthesized BDDs using the fully-interleaved ordering.

comparing the results between the input-first and fully-interleaved BDDs. While using the fully-interleaved ordering we were able to synthesize functions for n as large as 128 in under one second, it is safe to say that trying to synthesize such large functions using the input-first ordering, which had trouble reaching $n = 20$, would be impracticable. In fact, the difference between different synthesis methods turned out to be negligible compared to the difference between orderings. This suggests that, in using BDDs to synthesize Boolean functions, it might pay off to concentrate our efforts in identifying and taking advantage of efficient orderings before trying to develop faster methods.

It is also relevant that, as BDDs get bigger, their construction time seems to dominate the running time for synthesis. In light of this information, it might be worth researching ways to improve initial BDD construction performance. One possibility might be to study lazy construction schemes that would not construct the whole BDD at once, but rather hold off on constructing certain branches until, and only if, they are necessary.

6 CONCLUSIONS AND FUTURE WORK

In this work we introduced methods for synthesizing Boolean functions and testing the realizability of specifications using BDDs. We also proposed the self-substitution quantifier elimination technique as an alternative to Shannon expansion. Finally, we analyzed experimentally the synthesis process we developed using operations over integers in binary as test cases.

Our results grant an overview of the possibility of using BDDs for the synthesis of Boolean functions. In particular, they paint a clear picture of the effect of the BDD ordering used in the performance of the process, to the point that the actual synthesis method being employed has relatively little impact in comparison. Although the input-first ordering showed itself to lead to a large blowup in the size of the BDDs, for the case of integer arithmetic we have found that the fully-interleaved ordering produces very efficient results.

There are several different directions for future work. For starters, we limited the experiments presented here to operations over integers in binary. Although this is a very relevant class of functions, and some generalized conclusions can be derived from their results, these operations also share properties that cannot be generalized to other functions, for example the efficiency of the fully-interleaved ordering in representing them. It would be good to perform further experiments on a more varied set of instances to see how different classes of functions behave.

Another promising avenue is trying techniques to lessen the impact of the BDD size in the synthesis process. The early quantification method applied in (PAN; VARDI, 2005) to satisfiability solving, with good results, may be able to be adapted for our purposes. There may also be BDD variants that can bring benefits in this area. For example, often we are not concerned with every possible path in the BDD, but only on finding a single path that leads to the terminal node 1. Therefore large parts of the BDD may be ignored in the synthesis process, making it a waste to construct them in the first place. A BDD variant that delays the construction of branches until they are needed can avoid this problem. Another BDD variant that might be worth researching are Free Binary Decision Diagrams (FBDDs) (GERGOV; MEINEL, 1994), which relax the variable ordering requirements in BDDs by allowing separate branches to use different orderings. This might allow for more efficient representation of specifications in cases where an efficient global ordering is difficult to find.

There is also a lot of potential research to be done on the self-substitution technique for quantifier elimination, both in the context of synthesis and independently of it. Its properties are currently the subject of ongoing work.

Finally, one must recall that synthesizing the BDD is not the definite final step in the synthesis process. Rather, the function still has to be mapped to a physical implementation as a circuit or program. How to optimize this mapping in order to obtain an efficient implementation is a question that warrants study.

REFERENCES

BRAND, D. Verification of large synthesized designs. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, 1993, SANTA CLARA, CALIFORNIA, USA, NOVEMBER 7-11, 1993, 1993. **Proceedings...** IEEE Computer Society / ACM, 1993. p.534–537.

BRYANT, R. E. Graph-Based Algorithms for Boolean Function Manipulation. **IEEE Trans. Comput.**, Washington, DC, USA, v.35, n.8, p.677–691, Aug. 1986.

BURCH, J. R. et al. Symbolic Model Checking: 10^20 states and beyond. **Inf. Comput.**, [S.l.], v.98, n.2, p.142–170, 1992.

CHATALIC, P.; SIMON, L. Multi-resolution on compressed sets of clauses. In: IEEE INTERNATIONAL CONFERENCE ON TOOLS WITH ARTIFICIAL INTELLIGENCE (ICTAI 2000), 13-15 NOVEMBER 2000, VANCOUVER, BC, CANADA, 12. **Anais...** IEEE Computer Society, 2000. p.2–10.

GERGOV, J.; MEINEL, C. Boolean Manipulation with Free BDDs: an application in combinational logic verification. In: IFIP CONGRESS (1). **Anais...** [S.l.: s.n.], 1994. p.309–314.

HOFFEREK, G. et al. Synthesizing multiple boolean functions using interpolation on a single proof. In: FORMAL METHODS IN COMPUTER-AIDED DESIGN, FMCAD 2013, PORTLAND, OR, USA, OCTOBER 20-23, 2013. **Anais...** IEEE, 2013. p.77–84.

HU, A. J.; DILL, D. L. Reducing BDD Size by Exploiting Functional Dependencies. In: DAC. **Anais...** [S.l.: s.n.], 1993. p.266–271.

JIANG, J. R. Quantifier Elimination via Functional Composition. In: COMPUTER AIDED VERIFICATION, 21ST INTERNATIONAL CONFERENCE, CAV 2009, GRENOBLE, FRANCE, JUNE 26 - JULY 2, 2009. PROCEEDINGS. **Anais...** Springer, 2009. p.383–397. (Lecture Notes in Computer Science, v.5643).

JIANG, J. R.; LIN, H.; HUNG, W. Interpolating functions from large Boolean relations. In: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN (ICCAD'09), NOVEMBER 2-5, 2009, SAN JOSE, CA, USA, 2009. **Anais...** IEEE, 2009. p.779–784.

KLIEBER, W. et al. Solving QBF with Free Variables. In: PRINCIPLES AND PRACTICE OF CONSTRAINT PROGRAMMING - 19TH INTERNATIONAL CONFERENCE, CP 2013, UPPSALA, SWEDEN, SEPTEMBER 16-20, 2013. PROCEEDINGS. **Anais...** Springer, 2013. p.415–431. (Lecture Notes in Computer Science, v.8124).

KNUTH, D. E. **The Art of Computer Programming, Volume 4, Fascicle 1**: bitwise tricks & techniques; binary decision diagrams. 12th.ed. [S.l.]: Addison-Wesley Professional, 2009.

KUKULA, J. H.; SHIPLE, T. R. Building Circuits from Relations. In: COMPUTER AIDED VERIFICATION, 12TH INTERNATIONAL CONFERENCE, CAV 2000, CHICAGO, IL, USA, JULY 15-19, 2000, PROCEEDINGS. **Anais...** Springer, 2000. p.113–123. (Lecture Notes in Computer Science, v.1855).

KUNCAK, V. et al. Complete functional synthesis. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, PLDI 2010, TORONTO, ONTARIO, CANADA, JUNE 5-10, 2010, 2010. **Proceedings...** ACM, 2010. p.316–329.

MARI, F. et al. From Boolean Functional Equations to Control Software. **CoRR**, [S.l.], v.abs/1106.0468, 2011.

MINATO, S. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In: DAC. **Anais...** [S.l.: s.n.], 1993. p.272–277.

PAN, G.; VARDI, M. Y. Symbolic Techniques in Satisfiability Solving. **J. Autom. Reasoning**, [S.l.], v.35, n.1-3, p.25–50, 2005.

RICE, M.; KULHARI, S. **A survey of static variable ordering heuristics for efficient bdd/mdd construction**. [S.l.]: University of California, Riverside, 2008.

SOMENZI, F. Binary Decision Diagrams. **Calculational system design**, [S.l.], v.173, p.303, 1999.

SOMENZI, F. **CUDD**: CU Decision Diagram package release 2.5. 0. [S.l.]: University of Colorado at Boulder, 2012.

TRONCI, E. Automatic Synthesis of Controllers from Formal Specifications. In: ICFEM. **Anais...** [S.l.: s.n.], 1998. p.134–143.