Masaryk University
Faculty of Informatics

Satisfiability of DQBF Using Binary Decision Diagrams

Master's Thesis

Juraj Síč

Brno, Spring 2020

Masaryk University
Faculty of Informatics

Satisfiability of DQBF Using Binary Decision Diagrams

Master's Thesis

Juraj Síč

Brno, Spring 2020

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Juraj Síč

Advisor: doc. RNDr. Jan Strejček, Ph.D.

Acknowledgements

I would like to thank my advisor doc. RNDr. Jan Strejček, Ph.D. for all his help with writing this thesis, especially for the fruitful discussions we held and for finding the time to thoroughly read the drafts and offer very helpful and extensive improvement suggestions.

Abstract

In this thesis, we devise and implement a satisfiability solver DQBDD for *dependency quantified Boolean formulas* (DQBFs), which are an extension of *quantified Boolean formulas* (QBFs) where the dependencies between quantifiers are explicitly given. It uses *binary decision diagrams* (BDDs) as an underlying representation of Boolean formulas with quantifier elimination approach for solving. We show that an existing solution based on quantifier elimination, which uses quantifier localisation to push quantifiers inside the formula, is erroneous and propose a fix for it with an enhancement which allows universal quantifier elimination inside subformulas. Finally, we compare the performance of DQBDD with existing state-of-the-art tools (dCAQE, HQS, iDQ and iProver) and show that it results in a very competitive solver.

Keywords

dependency quantified Boolean formula, DQBF, binary decision diagram, BDD, satisfiability, quantifier elimination, quantifier localisation

Contents

1 Introduction

The well known NP-complete SAT problem that determines whether a Boolean formula is satisfiable or not is one of the most fundamental problems of computer science. In addition to its theoretical importance, it has many practical applications. However, sometimes SAT is not enough, which led to a generalisation of SAT problem called the quantified Boolean formula (QBF) problem.

The QBF problem is a PSPACE-complete problem of determining whether a Boolean formula with universal and existential quantifiers is satisfiable. This allows for a more succinct representation of problem descriptions and wider possibilities of applications that resulted in several different QBF solvers. However, for some applications, for example *partial equivalence checking* (PEC) [1], this is still not enough. We can extend QBF by Henkin quantifiers [2], which allow to explicitly give the dependencies between quantified variables, resulting in *dependency quantified Boolean formula* (DQBF) [3]. The problem of deciding satisfiability of DQBF is NEXPTIME-complete [4], which means that it allows even more succinct problem descriptions.

In recent years, DQBF solving is on a rise (driven mostly by PEC) which resulted in the first known solution for DQBF problem using DPLL algorithm [5]. This was followed by multiple DQBF solvers, namely iDQ [6], iProver [7], HQS [8, 9, 10], and dCAQE [11], using different solving techniques. HQS, the most successful of these (winner of both 2018 and 2019 DQBF track of QBF Evaluation competition [12, 13]) uses *quantifier elimination* [8]. The quantifiers are eliminated one by one using universal expansion (which increases the size of the formula) until we end up with a QBF on which an existing QBF solver is run. It also uses a succinct representation of Boolean formulas called *and-inverter graphs* (AIGs) [14] that made this solving method feasible.

Binary decision diagrams (BDDs) [15] are another data structure that can represent Boolean formulas. Since its introduction, BDDs have been used in many different applications. Compared to AIGs, they are a better representation in the way that, for a given variable ordering, there is only one BDD for each class of equivalent Boolean formulas.

In this work, we are interested in creating a DQBF solver that uses BDDs as an underlying representation of formulas combined with quantifier elimination. We use *quantifier localisation* [10] (also used in HQS) which allows us to push quantifiers inside the formula and eliminate them only "locally" in subformulas. We show that the localisation used in HQS is erroneous and we give a corrected version of the localisation algorithm. We also give an important missing proof that shows the localisation can also be used in subformulas and enhance it with "local" universal expansion.

We present all the needed theory in Chapter 2. In Chapter 3 we give the aforementioned proofs. Chapter 4 gives an overview of existing DQBF solving techniques. In Chapter 5 we show the new algorithm and describe its implementation in Chapter 6 resulting in tool DQBDD. Finally, we perform an experimental evaluation where we compare DQBDD with other solvers in Chapter 7.

2 Theory

In this chapter we give the definitions for *dependency quantified Boolean formulas* (DQBFs), *binary decision diagrams* (BDDs) and *quantifier trees*.

For DQBFs, we build up from *Boolean formulas* (BFs) to which we add quantifiers resulting in *quantified Boolean formulas* (QBFs). After this, we add explicit dependencies to the definition of QBFs, which results in DQBFs. For these, we give two forms, one that can only have quantifiers at the beginning of the formula and one that allows them deeper in the formula.

After this, we define BDDs and recall some of their properties. Finally, we describe quantifier trees which we can use as a representation of DQBFs.

2.1 Boolean Formulas

We first start with defining Boolean formulas. For these, we need a set of variables $V = \{x_1, \ldots, x_n\}$. We then define Boolean formulas as these variables connected by logical operatives *and* ($\wedge$), *or* ($\vee$) and *negation* ($\neg$).

Definition 1. Let $V = \{x_1, \ldots, x_n\}$ be a set of variables. The set of *Boolean formulas* (BFs) *over* V, denoted by $\mathbf{\Phi}_V^{\mathrm{BF}}$, is defined as the smallest set fulfilling these conditions:

- $0, 1 \in \mathbf{\Phi}_V^{\mathrm{BF}}$,
- $x \in \mathbf{\Phi}_V^{\mathrm{BF}}$ if $x \in V$,
- $(\phi_1 \wedge \phi_2) \in \mathbf{\Phi}_V^{\mathrm{BF}}$ if $\phi_1, \phi_2 \in \mathbf{\Phi}_V^{\mathrm{BF}}$,
- $(\phi_1 \vee \phi_2) \in \mathbf{\Phi}_V^{\mathrm{BF}}$ if $\phi_1, \phi_2 \in \mathbf{\Phi}_V^{\mathrm{BF}}$,
- $\neg\phi \in \mathbf{\Phi}_V^{\mathrm{BF}}$ if $\phi \in \mathbf{\Phi}_V^{\mathrm{BF}}$.

By $\phi\,[\phi_2/\phi_1]$, where $\phi, \phi_1, \phi_2 \in \mathbf{\Phi}_V^{\mathrm{BF}}$, we denote the BF ϕ where each occurrence of ϕ_1 is replaced by ϕ_2. We also use $(\phi_1 \Leftrightarrow \phi_2)$ as a shorthand for the formula $((\phi_1 \wedge \phi_2) \vee (\neg\phi_1 \wedge \neg\phi_2))$.

A function $v\colon V \to \{0,1\}$ is called a *valuation* over the set of variables V and the set of all valuations over V is denoted by $\mathbf{A}_V$. A valuation says whether each variable is true or false which we can extend to BFs: an *evaluation* of $\phi \in \mathbf{\Phi}_V^{\mathrm{BF}}$ in a valuation $v \in \mathbf{A}_V$, denoted $v(\phi)$, is defined as

- if $\phi = 0$ then $v(\phi) = 0$,
- if $\phi = 1$ then $v(\phi) = 1$,
- if $\phi = x_i$ where $x_i \in V$, then $v(\phi) = v(x_i)$,
- if $\phi = \neg\phi'$, then $v(\phi) = 1$ if $v(\phi') = 0$, otherwise $v(\phi) = 0$,
- if $\phi = (\phi_1 \wedge \phi_2)$, then $v(\phi) = 1$ if both $v(\phi_1) = 1$ and $v(\phi_2) = 1$, otherwise $v(\phi) = 0$, and
- if $\phi = (\phi_1 \vee \phi_2)$, then $v(\phi) = 1$ if $v(\phi_1) = 1$ or $v(\phi_2) = 1$, otherwise $v(\phi) = 0$.

Each $\phi \in \mathbf{\Phi}_V^{\mathrm{BF}}$ represents a *Boolean function* $f_\phi\colon \mathbf{A}_V \to \{0,1\}$ *over* V such that $f_\phi(v) = v(\phi)$. Conversely, for each Boolean function we can find a BF representing it. The set of all Boolean functions over V is denoted by $\mathbb{F}_V$. The *support set* $\mathrm{supp}(f)$ of Boolean function f is defined as the set of variables occurring in a BF ϕ where ϕ represents f and from all the BFs representing f, ϕ has the smallest number of variables occurring in it. That is, changing the values of variables that are not in the support set does not change the output value of f.

An important notion for BFs is *satisfiability*. We say that $\phi \in \mathbf{\Phi}_V^{\mathrm{BF}}$ is *satisfiable* if there exists a valuation $v \in \mathbf{A}_V$ in which ϕ is true, that is $v(\phi) = 1$. Let $\phi_1, \phi_2 \in \mathbf{\Phi}_V^{\mathrm{BF}}$. We say that they are *equivalent*, denoted $\phi_1 \equiv \phi_2$, if for all valuations v it holds that $v(\phi_1) = v(\phi_2)$. If they are either both satisfiable or they are both unsatisfiable, we call them *equisatisfiable*, denoted $\phi_1 \approx \phi_2$. Notice, that two BFs are equivalent iff they represent the same Boolean function.

Example 1. Let

$$\phi = ((x_1 \wedge x_2) \Leftrightarrow (y_1 \Leftrightarrow y_2)).$$

This formula says that if both x_1 and x_2 are true, then y_1 and y_2 must be the same, otherwise they must be different. It is satisfiable, because in

valuation v such that $v(x_1) = v(x_2) = v(y_1) = v(y_2) = 1$ it holds that ϕ is evaluated to $v(\phi) = 1$. The Boolean function f_ϕ that ϕ represents has the support set $\text{supp}(f_\phi) = \{x_1, x_2, y_1, y_2\}$.

2.1.1 Conjunctive Normal Form

To work with BFs, it is usually easier to have them in some special form. The first one we define is called conjunctive normal form.

Definition 2. We say that Boolean formula $\phi \in \mathbf{\Phi}_V^{\text{BF}}$ is in *conjuctive normal form* (CNF) if there exist $\phi_1, \ldots, \phi_n \in \mathbf{\Phi}_V^{\text{BF}}$ where

$$\phi = \phi_1 \wedge \cdots \wedge \phi_n$$

and

$$\phi_i = l_{i1} \vee \cdots \vee l_{im_i}$$

for each $i = 1, \ldots, n$, and for each l_{ij} where $j \in \{1, \ldots, m_i\}$ either $l_{ij} = x$ or $l_{ij} = \neg x$ for some $x \in V$.

Formulas $\phi_1, \ldots, \phi_n$ are called clauses and l_{ij} are called literals. In other words, a Boolean formula is in CNF if it is a conjunction of clauses, which are disjunctions of literals, where a literal is a variable or its negation.

Example 2. The formula

$$\phi_{CNF} = (x \vee \neg y) \wedge (\neg x \vee y)$$

is in CNF with clauses $(x \vee \neg y)$ and $(\neg x \vee y)$ and literals $x, \neg y, \neg x$ and y.

For every BF, ϕ there exists an equivalent formula in CNF which can be exponentially larger than ϕ. However, by using Tseitin transformation [16] we can create an equisatisfiable BF ϕ_{CNF} which is only polynomially larger than the original BF ϕ.

2.1.2 Negation Normal Form

Definition 3. We say that BF ϕ is in *negation normal form* (NNF) if all negations occur only in front of variables.

Each BF ϕ can be easily transformed into an equivalent BF ϕ' in NNF by applying these three equivalences:

$$\neg(\phi_1 \wedge \phi_2) \equiv (\neg\phi_1 \vee \neg\phi_2) \tag{2.1}$$
$$\neg(\phi_1 \vee \phi_2) \equiv (\neg\phi_1 \wedge \neg\phi_2) \tag{2.2}$$
$$\neg\neg x \equiv x \tag{2.3}$$

Example 3. Formula $\neg(x_1 \wedge (x_2 \vee \neg x_3))$ that is not in NNF is by (2.1) equal to $(\neg x_1 \vee \neg(x_2 \vee \neg x_3))$ which is by (2.2) equal to $(\neg x_1 \vee (\neg x_2 \wedge \neg\neg x_3))$. Finally, by (2.3) we get $(\neg x_1 \vee (\neg x_2 \wedge x_3))$ which is in NNF.

2.2 Quantified Boolean Formulas

Having defined BFs, we can move to the next step on the way to defining DQBF by adding quantifiers. We add existential ($\exists$) and universal ($\forall$) quantifiers bounded to variables which result in the definition of *quantified Boolean formulas* (QBFs). Usually, definitions of QBFs allow quantifiers everywhere inside formula but here we only define of one special form of QBFs called prenex normal form. In this form, QBFs allow quantifiers only at the beginning of the formula. However, this does not change the expressibility as it is possible to transform every QBF to an equivalent QBF in prenex normal form [17].

Definition 4. Let $V = \{x_1, \ldots, x_n\}$ be a set of variables and $\phi \in \mathbf{\Phi}_V^{\mathrm{BF}}$. A *quantified Boolean formula* (QBF) ψ *over* V in *prenex normal form* is given by

$$\psi = Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \, \phi$$

where $Q_i \in \{\exists, \forall\}$ for all $i \in \{1, 2, \ldots, n\}$.

The BF ϕ is called the *matrix* of ψ and $Q_1 x_1 Q_2 x_2 \ldots Q_n x_n$ is called the *quantifier prefix* of ψ. A variable x_i is called *existential* if $Q_i = \exists$ and *universal* if $Q_i = \forall$. The set of all existential variables of QBF ψ is denoted by $V_\psi^\exists$ and the set of all universal variables of ψ is denoted by $V_\psi^\forall$. We define a *level function* $l\colon V \to \mathbb{N}$ which maps to each variable the number of quantifiers occurring before the variable in the formula. That is the variable $x_i \in V$ from QBF ψ of Definition 4 has level $l(x_i) = i$. We denote the set of variables with level lower than some $n \in \mathbb{N}$ as $V_\psi^{<n} = \{\, x \in V \mid l(x) < n \,\}$.

Again, we define satisfiability. Usually, satisfiability for QBFs is defined trough valuation function as in BFs case where evaluation of QBFs follows the same rules as BFs with two newly added rules for quantifiers:

- $v(\forall x Q_i x_i \dots Q_n x_n \, \phi) = 1$ if both $v(Q_i x_i \dots Q_n x_n \, \phi \, [0/x]) = 1$ and $v(Q_i x_i \dots Q_n x_n \, \phi \, [1/x]) = 1$, otherwise it equals 0,
- $v(\exists x Q_i x_i \dots Q_n x_n \, \phi) = 1$ if $v(Q_i x_i \dots Q_n x_n \, \phi \, [0/x]) = 1$ or $v(Q_i x_i \dots Q_n x_n \, \phi \, [1/x]) = 1$, otherwise it equals 0.

We follow different but equal definition based on Skolemisation, which replaces each existential variable x by some Boolean function over universal variables with lower level than x. This is better suited to show a correspondence of QBFs and DQBFs when we add dependencies in the next section. A QBF ψ of Definition 4 is satisfiable if for each $x \in V_\psi^\exists$ there exists a function $s_x \in \mathbb{F}_{V_\psi^\forall \cap V_\psi^{<l(x)}}$ (called *Skolem function*) such that the matrix ϕ of ψ, where every $x \in V_\psi^\exists$ is replaced by some BF that represents s_x, is evaluated to 1 in every valuation $v \colon V_\psi^\forall \to \{0,1\}$.

Example 4. Let

$$\psi = \forall x_1 \forall x_2 \exists y_1 \exists y_2 \, ((x_1 \wedge x_2) \Leftrightarrow (y_1 \Leftrightarrow y_2))$$

be a QBF where the matrix is formula ϕ from Example 1. This QBF is satisfiable because for all values of x_1, x_2 we can find values of y_1 and y_2 such that they are same if $x_1 = x_2 = 1$ and different otherwise. The Skolem functions showing satisfiability of this formula are for example s_{y_1} which is represented by BF $x_1 \wedge x_2$, and s_{y_2} which is represented by BF 1. If we replace existential variables with their Skolem functions, we get BF

$$((x_1 \wedge x_2) \Leftrightarrow ((x_1 \wedge x_2) \Leftrightarrow 1))$$

which is true in all valuations.

2.3 Dependency Quantified Boolean Formulas

Quantifiers have expanded the succinctness of BFs quite considerably. However, there is still one drawback of QBFs — quantified variable x

depends on all variables that are quantified before x in the formula. The question arises whether it is possible define a formula where the dependency relation is somehow explicitly given. The answer is yes, we can use *dependency quantified Boolean formulas* (DQBFs).

DQBFs allow existential variables to be non-linearly dependent on the set of universal variables by explicitly writing out the set of universal variables on which each existential variable is dependent. The universal variables from this set are then used as the support set of the Skolem function of the existential variable which allows for even greater succinctness of DQBFs over QBFs.

We start with the definition of DQBFs in *prenex normal form* on which we explain the basic notions and then we give the definition of DQBFs in non-prenex *negation normal form* which allows quantifiers inside formulas.

2.3.1 Prenex Normal Form

The first definition of DQBF form called prenex normal form is an analogy of the prenex normal form of QBFs.

Definition 5. Let $V = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ be a set of variables and $\phi \in \mathbf{\Phi}_V^{\text{BF}}$ a BF over V. A *dependency quantified Boolean formula* (DQBF) ψ in *prenex normal form* (PNF) is given by

$$\psi = \forall x_1 \ldots \forall x_n \exists y_1(D_1) \ldots \exists y_m(D_m)\, \phi$$

where $D_i \subseteq \{x_1, \ldots, x_n\}$ for each $i \in \{1, \ldots, m\}$ is a *dependency set* of variable y_i.

BF ϕ is the *matrix* of ψ and $\forall x_1 \ldots \forall x_n \exists y_1(D_1) \ldots \exists y_m(D_m)$ the *quantifier prefix* of ψ. We call the variables from $V_\psi^\exists = \{y_1, \ldots, y_m\}$ *existential* and variables from $V_\psi^\forall = \{x_1, \ldots, x_n\}$ *universal*. We also say that ψ is in *prenex conjunctive normal form* (PCNF) if it is in PNF where the matrix is in CNF.

Notation. When we enumerate the elements of dependency sets in formula, we do not write curly braces around it, so for example instead of $\forall x_1 \forall x_2 \exists y(\{x_1, x_2\})\, \phi$, we write $\forall x_1 \forall x_2 \exists y(x_1, x_2)\, \phi$.

Example 5. Let us use an example by Rabe [18] to explain the meaning behind dependency sets. Let

$$\psi = \forall x_1 \forall x_2 \exists y_1(x_1) \exists y_2(x_2)\, ((x_1 \wedge x_2) \Leftrightarrow (y_1 \Leftrightarrow y_2))$$

be a DQBF. The matrix is the BF ϕ from Example 1 and the quantifier prefix is very similar to the QBF from Example 4 but now both y_1 and y_2 depend only on one variable, x_1 and x_2 respectively. The QBF was satisfiable but this DQBF is not. To show why, we can look at it as a game where y_1 and y_2 are trying to satisfy the formula and x_1, x_2 are trying to make it false. However, y_1 knows only how x_1 is behaving and similarly y_2 knows only what x_2 is doing. As explained in Example 1 the matrix says that if both x_1 and x_2 are true, then y_1 and y_2 should be same, otherwise they need to be different. However, this is not possible, because if for example both y_1, y_2 decide to be true (similarly for false) when their respective universal variables are true then if one of the universal variable change, for example x_1 turns to false, then y_2 does not know about this and stays true. In that case y_1 has to change the value to false so they become different. The same thing happens to y_2 if x_2 changes its value to false. So we get behaviour where both existential variables just copy the behaviour of their universal variable. But at the end, if both x_1 and x_2 change their values to false, then the values of y_1 and y_2 are the same (false) which makes the formula ψ false.

With the example in mind, we define satisfiability for DQBF in PNF similarly to how satisfiability is defined for QBFs with Skolem functions. DQBF ψ of Definition 5 is *satisfiable* if for each $y \in V_\psi^\exists$ there exists a Skolem function $s_y \in \mathbb{F}_{D_y}$ such that ϕ, where each $y \in V_\psi^\exists$ is replaced by a BF representing Boolean function s_y, evaluates to 1 in every valuation $v \in \mathbf{A}_{V_\psi^\forall}$.

Example 6. The possible Skolem functions s_{y_1} for existential variable y_1 from the previous example are those that are represented by BFs 0, 1, x_1 or $\neg x_1$. For y_2 the Skolem functions s_{y_2} can be those that are represented by BFs 0, 1, x_2 or $\neg x_2$. However, we cannot choose a pair s_{y_1} and s_{y_2} such that by replacing y_1 and y_2 with them, we get a BF that is true in all valuations. In the previous example we actually used

s_{y_1} represented by x_1 and s_{y_2} represented by x_2, and we got BF

$$((x_1 \wedge x_2) \Leftrightarrow (x_1 \Leftrightarrow x_2))$$

which is obviously false in some valuations.

Remark. QBFs in PNF can be seen as a special case of DQBFs in PNF where the dependency sets are linearly ordered by the subset relation ordering. This means, that a DQBF ψ, where for each two dependency sets D_{y_1} and D_{y_2} it holds that $D_{y_1} \subseteq D_{y_2}$ or $D_{y_2} \subseteq D_{y_1}$, can be easily transformed to QBF by reordering the quantifiers.

2.3.2 Negation Normal Form

In this section we define *negation normal form* of DQBF which is non-prenex thus allowing quantifiers inside the formula. However, negation is only allowed in front of the variables. This form is also not closed, meaning that there can be so-called free variables that are not bounded by any quantifier. The definitions in this section are taken from Ge-Ernst et al. [10].

We first give the definition of the set $\mathbf{\Phi}_V^{\mathrm{DQBF}}$ of DQBFs in *negation normal form* where we use the rules of Figure 2.1. Each rule has the list of conditions (above the line) that has to hold so that the resulting formula ψ under the line is in $\mathbf{\Phi}_V^{\mathrm{DQBF}}$. We also define the set of existential $V_\psi^\exists$, universal $V_\psi^\forall$, and free V_ψ^{free} variables occurring in ψ (the columns $V_\cdot^\exists$, $V_\cdot^\forall$ and $V_\cdot^{\mathrm{free}}$ respectively). We also use $V_\psi^Q = V_\psi^\exists \cup V_\psi^\forall$ and $V_\psi = V_\psi^Q \cup V_\psi^{\mathrm{free}}$. Furthermore, ψ^{-y} results from ψ by removing y from every dependency set in ψ.

Definition 6. Let V be a set of variables. The set $\mathbf{\Phi}_V^{\mathrm{DQBF}}$ of *DQBFs in negation normal form* (NNF) *over* V is defined to be the smallest set satisfying the rules from Figure 2.1.

Remark. In the definition by Ge-Ernst et al. [10] the last two rules in Figure 2.1 have condition $x \in V_\psi^{\mathrm{free}}$ instead of $x \in V \setminus V_V^Q$. This condition is too strong and does not allow some formulas which are valid. For example, as we show in Section 3.1, we can transform DQBFs by pushing quantifiers inside the formula and this way we can end up with a formula $\forall x\, \psi$ where $x \notin V_\psi$. However, according to the aforementioned definition, this would not be a valid formula.

rule	$V_\cdot^\exists$	$V_\cdot^\forall$	$V_\cdot^{\text{free}}$
$\dfrac{}{0 \in \mathbf{\Phi}_V^{\text{DQBF}}} \qquad \dfrac{}{1 \in \mathbf{\Phi}_V^{\text{DQBF}}}$	$\emptyset$	$\emptyset$	$\emptyset$
$\dfrac{x \in V}{x \in \mathbf{\Phi}_V^{\text{DQBF}}} \qquad \dfrac{x \in V}{\neg x \in \mathbf{\Phi}_V^{\text{DQBF}}}$	$\emptyset$	$\emptyset$	$\{x\}$
$\dfrac{\psi_1 \in \mathbf{\Phi}_V^{\text{DQBF}} \quad \psi_2 \in \mathbf{\Phi}_V^{\text{DQBF}} \quad (1)}{(\psi_1 \wedge \psi_2) \in \mathbf{\Phi}_V^{\text{DQBF}}}$	$V_{\psi_1}^\exists \cup V_{\psi_2}^\exists$	$V_{\psi_1}^\forall \cup V_{\psi_2}^\forall$	$V_{\psi_1}^{\text{free}} \cup V_{\psi_2}^{\text{free}}$
$\dfrac{\psi_1 \in \mathbf{\Phi}_V^{\text{DQBF}} \quad \psi_2 \in \mathbf{\Phi}_V^{\text{DQBF}} \quad (1)}{(\psi_1 \vee \psi_2) \in \mathbf{\Phi}_V^{\text{DQBF}}}$	$V_{\psi_1}^\exists \cup V_{\psi_2}^\exists$	$V_{\psi_2}^\forall \cup V_{\psi_2}^\forall$	$V_{\psi_1}^{\text{free}} \cup V_{\psi_2}^{\text{free}}$
$\dfrac{\psi \in \mathbf{\Phi}_V^{\text{DQBF}} \quad y \in V \setminus V_\psi^Q \quad (2)}{\exists y(D_y)\, \psi^{-y} \in \mathbf{\Phi}_V^{\text{DQBF}}}$	$V_\psi^\exists \cup \{y\}$	$V_\psi^\forall$	$V_\psi^{\text{free}} \setminus \{y\}$
$\dfrac{\psi \in \mathbf{\Phi}_V^{\text{DQBF}} \quad x \in V \setminus V_\psi^Q}{\forall x\, \psi \in \mathbf{\Phi}_V^{\text{DQBF}}}$	$V_\psi^\exists$	$V_\psi^\forall \cup \{x\}$	$V_\psi^{\text{free}} \setminus \{x\}$

Figure 2.1: The rules defining the syntax of DQBFs in NNF (based on Ge-Ernst et al. [10]) where (1) refers to

$$V_{\psi_1} \cap V_{\psi_2}^Q = \emptyset \quad \text{and} \quad V_{\psi_1}^Q \cap V_{\psi_2} = \emptyset \tag{1}$$

and (2) refers to

$$D_y \subseteq V \setminus (V_\psi^Q \cup \{y\}). \tag{2}$$

Remark. Notice that each DQBF ψ in PNF can be transformed into DQBF ψ' in NNF by transforming the matrix of ψ to NNF as explained in Section 2.1.2.

By $\psi\,[x_2/x_1]$ where $\psi \in \mathbf{\Phi}_V^{\text{DQBF}}$, $x_1, x_2 \in V$, $x_2 \notin V_\psi$ we denote the DQBF ψ in which every occurrence of x_1 (even in the dependency sets) is replaced by x_2. If $x_1 \notin V_\psi^Q$ and is not in any dependency set in ψ then by $\psi\,[\kappa/x_1]$, where $\kappa \in \{0, 1\}$, we denote the DQBF ψ in which every occurrence of x_1 is replaced by κ.

The definition of satisfiability of DQBF in NNF follows the same way as in the previous section but now we have to decide what to do with free variables. In non-prenex case of QBFs, free variables are usually assumed to be existential variables with quantifiers at the beginning of the formula. In DQBF case this means that free variables would have empty dependency sets so the definition needs to follow that. We first define a mapping, called *Skolem functions mapping* which maps to each existential and free variable some Skolem function fulfilling conditions given by its dependency set. Then we define *semantics* of DQBFs in NNF as the set of Skolem functions mappings which satisfy the formula.

Definition 7 (Skolem Functions Mapping)**.** Let $\psi \in \mathbf{\Phi}_V^{\text{DQBF}}$. We say that a mapping $s\colon (V_\psi^\exists \cup V_\psi^{\text{free}}) \to \mathbb{F}_{V_\psi^\forall}$ is a *Skolem functions mapping* of ψ if

- $\text{supp}(s(y)) \subseteq (D_y \cap V_\psi^\forall)$ for all $y \in V_\psi^\exists$ and
- $\text{supp}(s(y)) = \emptyset$ for all $y \in V_\psi^{\text{free}}$, i.e. $s(y)$ is either represented by 0 or 1.

The set of all Skolem functions mappings of ψ is denoted by $\mathbb{S}_\psi$. If $s \in \mathbb{S}_\psi$, we write $s(\psi)$ for the formula that results from ψ by replacing each existential and free variable y by a BF that represents $s(y)$ and omitting all quantifiers from ψ. This results in a BF containing only variables from $V_\psi^\forall$.

Definition 8 (Semantics)**.** Let $\psi \in \mathbf{\Phi}_V^{\text{DQBF}}$. We define the *semantics* $[\![\psi]\!]$ of ψ as follows:

$$[\![\psi]\!] = \{\, s \in \mathbb{S}_\psi \mid v(s(\psi)) = 1 \text{ for all } v \in \mathbf{A}_{V_\psi^\forall} \,\}.$$

Formula ψ is *satisfiable* iff $[\![\psi]\!] \neq \emptyset$ and the elements of $[\![\psi]\!]$ are *satisfying* Skolem functions mappings of ψ. We say that two formulas ψ_1 and ψ_2 are *equisatisfiable*, denoted $\psi_1 \approx \psi_2$, if both are either satisfiable or both are unsatisfiable.

Example 7. To explain the semantics of DQBF in NNF, we use the example by Ge-Ernst et al. [10]:

$$\psi = \forall x_1 \forall x_2 \, ((x_1 \Leftrightarrow x_2) \vee \exists y(x_2) \, \neg(x_1 \Leftrightarrow y)).$$

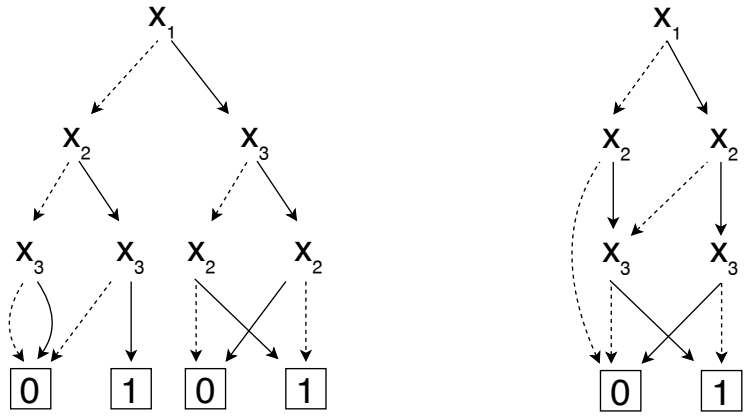
The meaning behind this formula is that either x_1 and x_2 are the same or there is some y which is dependent only on x_2 which is different from x_1. The Skolem functions mappings of ψ are then $\mathbb{S}_\psi = \{y \mapsto f_0, y \mapsto f_1, y \mapsto f_{x_2}, y \mapsto f_{\neg x_2}\}$ where f_0 is the Boolean function represented by Boolean formula 0, f_1 is represented by 1, f_{x_2} by x_2 and $f_{\neg x_2}$ by $\neg x_2$. It is obvious that only $s = y \mapsto f_{x_2}$ is a satisfying mapping of ψ, since only $s(\psi) = ((x_1 \Leftrightarrow x_2) \vee \neg(x_1 \Leftrightarrow x_2))$ is true in all valuations.

2.4 Binary Decision Diagrams

In this section we recall a definition of *binary decision diagrams* [15], which is a succinct representation of Boolean functions (and to an extent Boolean formulas) with an efficient polynomial time implementation of logical operations.

Definition 9. Let V be a set of variables. A *binary decision diagram* (BDD) *over* V is a rooted directed acyclic graph with terminal nodes labelled with 0 and 1, and a set N of non-terminal nodes, each labelled with a variable from V. Each $n \in N$ has two child nodes, low child and high child. For each $x \in V$ and every path σ from root to a terminal node there is at most one $n \in N$ in σ representing x.

Let $v \colon V \to \{0, 1\}$ be a valuation over V. BDD β in this valuation is evaluated to 0 or 1 by starting in the root and for every $n \in N$, labelled by some $x \in V$, either moving to the low child if $v(x) = 0$ or to the high child if $v(x) = 1$. The resulting terminal node is then the evaluation $v(\beta)$ of β in v. BDD β then represents a Boolean function $f \in \mathbb{F}_V$ if for each valuation v it holds that $f(v) = v(\beta)$. Alternatively,



(a) BDD (b) ROBDD

Figure 2.2: An example of a BDD and an ROBDD representing the same BF $((\neg x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge \neg(x_2 \Leftrightarrow x_3))$

as each BF represents some Boolean function too, we can look at BDDs as a representation of (un)satisfying valuations of some BF. That is, for each BF $\phi \in \mathbf{\Phi}_V^{\mathrm{BF}}$ we can find a BDD β where $v(\phi) = v(\beta)$ for each $v \in \mathbf{A}_V$.

In figures, non-terminal nodes are denoted by the labelled variables, terminal nodes are shown as boxes with the labelled constants, edges to low children are dashed and edges to high children are full.

Example 8. Figure 2.2(a) shows an example of a BDD representing the BF $(\neg x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge \neg(x_2 \Leftrightarrow x_3))$.

To have simpler and more effective algorithms for BDDs, we use *reduced ordered BDDs* (ROBDDs) which are BDDs such that

1. there is some linear order of variables where each path from the root of the BDD to a terminal node complies with it,
2. there are no isomorphic ("the same looking") induced subgraphs and
3. there are no nodes with the same high and low child.

ROBDDs have the nice property that for a given variable order, each BF ϕ is represented by exactly one ROBDD [15]. Furthermore, it represents all BFs equivalent to ϕ. However, the size of two ROBDDs in

different variable orders can for the same formula be very different. We can find a formula which in one ordering has an ROBDD representation whose number of nodes is linear to the size of the formula and in a different one it is exponential. The problem of finding the optimal variable ordering is NP-hard [19], therefore we usually use some heuristics to reduce the size of ROBDDs and we do not look for the perfect order.

Example 9. Figure 2.2(b) shows an ROBDD which represents the same BF as the BDD from Example 8. While the BDD from Figure 2.2(a) does not have ordered variables, the paths from root to terminals in ROBDD respect the order x_1, x_2, x_3. Also, the leftmost path does not contain x_3, because both its low and high child ended in 0. Furthermore, both x_2 nodes share an isomorphic subgraphs.

For each operation on BFs ($\wedge$, $\vee$, $\neg$), there is a polynomial-time algorithm for ROBDDs, which can get the resulting ROBDD representing the application of the operation. For example, if we have an ROBDD representing BF ψ_1, we can get an ROBDD representing $\neg\psi_1$ by swapping terminal nodes 0 and 1. See for example Andersen [20] for more details.

In the following chapters, when we use BDD we actually mean ROBDD.

2.5 Quantifier Trees

While BDDs are a good representation of BFs, we also need a representation of DQBFs. For this we define *quantifier trees* which can represent DQBFs in NNF.

Definition 10. Let V be a set of variables. A *quantifier tree over* V is a rooted tree with labelling l which assigns to each non-terminal node an operation $\diamond \in \{\wedge, \vee\}$ and to each terminal node a literal from V (a variable $x \in V$ or its negation), and a prefix mapping Q which maps to each node some DQBF prefix.

We denote the set of children of node n as $children(n)$. A quantifier tree with root r can possibly represent a DQBF ψ_r where:

- if $l(r) = l_r$ is a literal, then $\psi_r = Q(r)\, l_r$,

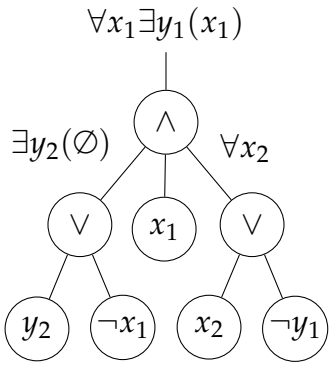


Figure 2.3: An example of quantifier tree

- if $l(r) = \diamond$ with $\diamond \in \{\wedge, \vee\}$ and $children(r) = \{n_1, \ldots, n_m\}$, then $\psi_r = Q(r)\, \psi_{n_1} \diamond \cdots \diamond \psi_{n_m}$ where $\psi_{n_1} \ldots \psi_{n_m}$ are DQBFs represented by subtrees rooted in children $n_1, \ldots, n_m$.

Not all trees represent some DQBF, we restrict ourselves only to the set of quantifier trees which represent some formula. However, for each DQBF there exists a quantifier tree representing it.

For a node n, we use $Q_\exists(n)$ and $Q_\forall(n)$ to denote the set of existentially and universally quantified variables in $Q(n)$. The sets V_n, $V_n^\exists$ and $V_n^\forall$ denote V_{ψ_n}, $V_{\psi_n}^\exists$ and $V_{\psi_n}^\forall$ respectively, where ψ_n is DQBF represented by a subtree rooted in n.

In figures, each node n is denoted by its assigned operation or literal $l(n)$, while the assigned DQBF prefix $Q(n)$ is always shown on the edge coming to it (see Figure 2.3). For the root, we add an extra "edge" on which we show this quantifier prefix.

Example 10. Figure 2.3 shows an example of quantifier tree representing DQBF

$$\forall x_1 \exists y_1(x_1)(\exists y_2(\emptyset)\, (y_2 \vee \neg x_1) \wedge x_1 \wedge \forall x_2\, (x_2 \vee \neg y_1)).$$

3 Transformations of DQBFs

In this chapter, we give theorems showing equisatisfiabilities of some DQBFs in NNF. Such formulas can be replaced with each other, allowing us to push (pull) quantifiers inside (from) the formula (Section 3.1) or eliminate them (Section 3.2). We use these results in Chapter 5 to develop an algorithm solving DQBFs.

3.1 Shifting Quantifiers

We start with a theorem that shows some rules by which we can push (extract) quantifiers in (from) the formula.

Theorem 3.1 ([10, Theorems 3, 4])**.** *Let* $\diamond \in \{\wedge, \vee\}$ *and* $\psi, \psi_1, \psi_2 \in \mathbf{\Phi}_V^{\mathrm{DQBF}}$*. We assume that variables* x*,* x_1*,* x_2*,* y*,* y_1*, and* y_2 *are not quantified in* ψ, ψ_1*, and* ψ_2*. We also assume that* x' *and* y' *are fresh variables, which do not occur in* ψ, ψ_1 *and* ψ_2*. Then we have:*

$$\forall x\,(\psi_1 \wedge \psi_2) \approx \left(\forall x\,\psi_1 \wedge \forall x'\,\psi_2\,[x'/x]\right) \tag{a}$$

$$\forall x\,(\psi_1 \wedge \psi_2) \approx \left(\psi_1^{-x} \wedge \forall x\,\psi_2\right), \text{ if } x \notin V_{\psi_1} \tag{b}$$

$$\forall x\,(\psi_1 \diamond \psi_2) \approx (\psi_1 \diamond \forall x\,\psi_2), \text{ if } x \notin V_{\psi_1} \text{ and } x \notin D_y \text{ for all } y \in V_{\psi_1}^{\exists} \tag{c}$$

$$\exists y(D_y)\,(\psi_1 \vee \psi_2) \approx \left(\exists y(D_y)\,\psi_1 \vee \exists y'(D_y)\,\psi_2\,[y'/y]\right) \tag{d}$$

$$\exists y(D_y)\,(\psi_1 \diamond \psi_2) \approx \left(\psi_1 \diamond \exists y(D_y)\,\psi_2\right), \text{ if } y \notin V_{\psi_1} \tag{e}$$

$$\exists y_1(D_{y_1})\exists y_2(D_{y_2})\,\psi \approx \exists y_2(D_{y_2})\exists y_1(D_{y_1})\,\psi \tag{f}$$

$$\forall x_1 \forall x_2\,\psi \approx \forall x_2 \forall x_1\,\psi \tag{g}$$

$$\forall x \exists y(D_y)\,\psi \approx \exists y(D_y) \forall x\,\psi, \text{ if } x \notin D_y. \tag{h}$$

To make these rules more usable for DQBF in NNF we have to show that they also work while replacing subformulas. Except for (b), this was not proven by Ge-Ernst et al. [10] even though they use them for replacing subformulas. As we will show, this was used erroneously because the rule (d) does not generally hold. However, for other rules, there is no problem as we show in Theorem 3.4. But first, we give some helpful notions and lemmas we use for the proofs of the following theorems.

In the proofs we work with Skolem function mappings, which turn DQBFs to BFs. We will want to show, that some subformulas are "important", i.e. changing their evaluation change the evaluation of the whole formula. Therefore we define the notion of dependency of Boolean formula on its subformula.

Definition 11. Let $\phi, \phi' \in \mathbf{\Phi}_V^{\mathrm{BF}}$, $v \in \mathbf{A}_V$ and ϕ' is subformula of ϕ. We say that ϕ *depends on* ϕ' *in* v, if $v(\phi\,[^0/_{\phi'}]) \neq v(\phi\,[^1/_{\phi'}])$.

This means that if ϕ depends on its subformula ϕ' in v, the value of $v(\phi)$ would change if the value of $v(\phi')$ was different.

Remark. If $v(\phi') = 1$ then $v(\phi) = v(\phi\,[^1/_{\phi'}])$ and if $v(\phi') = 0$ then $v(\phi) = v(\phi\,[^0/_{\phi'}])$.

The first lemma we prove shows that if the subformula ϕ' occurs in the tree of operations in ϕ without negation anywhere before it, then $v(\phi) = v(\phi')$.

Lemma 3.2. *Let $\phi, \phi' \in \mathbf{\Phi}_V^{\mathrm{BF}}$, $v \in \mathbf{A}_V$ where ϕ' is subformula of ϕ and ϕ depends on ϕ' in v. Assume that for each subformula $\neg\phi''$ of ϕ we have that ϕ' is not subformula of ϕ''. Then $v(\phi) = v(\phi')$.*

Proof. From assumptions, it holds that ϕ' is only in conjunctions and disjunctions in the tree of operations. Therefore it has to hold that if $v(\phi\,[^0/_{\phi'}]) = 1$ then also $v(\phi\,[^1/_{\phi'}]) = 1$. But because ϕ depends on ϕ' in v, this cannot be, which means that $v(\phi\,[^0/_{\phi'}]) = 0$ and $v(\phi\,[^1/_{\phi'}]) = 1$. This means that $v(\phi) = v(\phi')$. □

Next, we prove a lemma that says that if ϕ depends on its subformula ϕ' in a valuation v, then ϕ also depends on ϕ' in a valuation v', which differs from v only in the variables that do not occur outside ϕ'.

Lemma 3.3. *Let $\phi, \phi' \in \mathbf{\Phi}_V^{\mathrm{BF}}$, $v \in \mathbf{A}_V$ where ϕ' is subformula of ϕ and ϕ depends on ϕ' in v. Let $v' \in \mathbf{A}_V$ be a valuation where $v'(x) = v(x)$ for every x occurring outside the subformula ϕ'. Then ϕ depends on ϕ' in v'.*

Proof. Because we can only change the values of variable inside ϕ', it still holds that $v'(\phi\,[^1/_{\phi'}]) = v(\phi\,[^1/_{\phi'}])$ and $v'(\phi\,[^0/_{\phi'}]) = v(\phi\,[^0/_{\phi'}])$. Then, from the fact that ϕ depends on ϕ' in v, we have $v'(\phi\,[^1/_{\phi'}]) = v(\phi\,[^1/_{\phi'}]) \neq v'(\phi\,[^0/_{\phi'}]) = v(\phi\,[^0/_{\phi'}])$ which means that ϕ depends on ϕ' in v'. □

Theorem 3.4. *Let ϕ be the left side formula and ϕ' be the right side formula of the same equisatisfiability other than* (d) *from Theorem 3.1. Let $\psi \in \mathbf{\Phi}_V^{\mathrm{DQBF}}$ such that ϕ is a subformula of ψ and ψ', which results from ψ by replacing ϕ with ϕ', is a valid DQBF. Then $\psi \approx \psi'$.*

Proof. (a) $\phi = \forall x\,(\psi_1 \wedge \psi_2)$ and $\phi' = (\forall x\,\psi_1 \wedge \forall x'\,\psi_2\,[x'/x])$

We show that $[\![\psi]\!] \neq \emptyset$ iff $[\![\psi']\!] \neq \emptyset$. First, let $s \in [\![\psi]\!]$. We now show that for s', where

- $s'(y) = s(y)\,[x'/x]$[1] for $y \in V_{\psi_2}^{\exists}$ and
- $s'(y) = s(y)$ otherwise,

it holds $s' \in [\![\psi']\!]$. Note that for $y \in V_{\psi_1}^{\exists}$ we have $s'(y) = s(y)$ as, according to the way DQBFs are defined, ψ_1 and ψ_2 cannot share the same quantification, therefore $V_{\psi_1}^{\exists} \cap V_{\psi_2}^{\exists} = \emptyset$.

Let $v \in \mathbf{A}_{V_{\psi'}^{\forall}}$. We need to show that $v(s'(\psi')) = 1$. Because $s \in [\![\psi]\!]$, it holds that $v(s(\psi)) = 1$. For the case that $v(x') = v(x)$ we get $v(s(\psi)) = v(s'(\psi'))$, because BFs $s(\psi)$ and $s'(\psi')$ differ only in some replacements of x by x'. Therefore we assume that $v(x') \neq v(x)$. We know that ψ and ψ' differ only in the formulas ϕ and ϕ', therefore we can only focus on the case where $s(\psi)$ depends on $s(\phi)$ in v (otherwise $v(s(\psi)) = v(s'(\psi'))$). Also, ψ is in NNF, which means that in the tree of operations, negation cannot be before ϕ. We can then apply Lemma 3.2 and we get $v(s(\phi)) = 1$. This means we need to show that $v(s'(\phi')) = 1$.

We have that for $v(s(\phi)) = v(s(\forall x\,(\psi_1 \wedge \psi_2))) = v(s(\psi_1) \wedge s(\psi_2))$ to be equal to one, it has to hold that $v(s(\psi_1)) = 1$ and $v(s(\psi_2)) = 1$. Also, for v' which is equal to v, except that $v'(x) \neq v(x)$, it holds that $v'(s(\psi)) = 1$ (from $s \in [\![\psi]\!]$). Because x is not outside $s(\phi)$, we can use Lemma 3.3 to see that ψ depends on ϕ in v' too, and by applying Lemma 3.2 we get $v'(s(\phi)) = 1$. This means that $v'(s(\psi_2)) = 1$. Also $v'(s(\psi_2)) = v(s'(\psi_2\,[x'/x]))$, because $v'(x) = v(x')$. All in all, we get $v(s(\psi_1)) = v(s'(\psi_1)) = 1$ and $v(s'(\psi_2\,[x'/x])) = 1$ therefore $v(s'(\phi')) = v(s'(\forall x\,\psi_1 \wedge \forall x'\,\psi_2\,[x'/x])) = v(s'(\psi_1) \wedge s'(\psi_2\,[x'/x])) = 1$.

1. By $s(y)\,[x'/x]$ we mean the BF representing $s(y)$ in which every occurrence of x is replaced by x' and to $s(y')$ we assign Boolean function that $s(y)\,[x'/x]$ represents.

For the other direction, let $s' \in [\![\psi']\!]$. It can be easily shown by similar argument that s, where $s(y) = s'(y)\,[x/x']$ for $y \in V^{\exists}_{\psi_2}$ and $s(y) = s'(y)$ otherwise, is a satisfying Skolem functions mapping of ψ.

(b) This was proven by Ge-Ernst et al. [10, Theorem 4].

(c)(e)(f)(g)(h) Because $V^{\forall}_{\psi} = V^{\forall}_{\psi'}$, $V^{\exists}_{\psi} = V^{\exists}_{\psi'}$, and $V^{\text{free}}_{\psi} = V^{\text{free}}_{\psi'}$ and the dependency sets do not change, it holds that $\mathbb{S}_{\psi} = \mathbb{S}_{\psi'}$. Also, for any $s \in \mathbb{S}_{\psi}$ it holds that $s(\psi)$ is the same BF as $s(\psi')$ which means that $[\![\psi]\!] = [\![\psi']\!]$, therefore $\psi \approx \psi'$. □

As we already mentioned, the rule (d) as it stands cannot be generally used for replacing subformulas. We show it on the following formula:

$$\forall x \exists y(\emptyset)\,((x \wedge y) \vee (\neg x \wedge \neg y)).$$

It is obviously not satisfiable because whether $s(y) = 0$ or $s(y) = 1$ we can always find valuation in which both disjuncts are not true. However if we assumed that the rule (d) can be also used for subformulas, we would get

$$\forall x\,(\exists y(\emptyset)\,(x \wedge y) \vee \exists y'(\emptyset)\,(\neg x \wedge \neg y')),$$

which is satisfiable for s where $s(y) = 1$ and $s(y') = 0$. To fix this, we need to restrict the variables that can occur in the disjuncts. When we look at this formula, it would appear that the problem arises from having universal variable x on which y is not dependent in both disjuncts. However, we can find an example when even if both disjuncts contain different sets of universal variables (other than the ones from D_y), this rule does not work. For example for unsatisfiable formula

$$\forall x_1 \forall x_2\,(\exists y(\emptyset)\,((x_1 \wedge y) \vee (x_2 \wedge \neg y)) \vee (\neg x_1 \wedge \neg x_2)),$$

we have that $(x_1 \wedge y)$ contains x_1, which is not in $(x_2 \wedge \neg y)$. and x_2 is only in the second disjunct. However, after applying the rule (d) we get

$$\forall x_1 \forall x_2\,((\exists y(\emptyset)\,(x_1 \wedge y) \vee \exists y'(\emptyset)\,(x_2 \wedge \neg y')) \vee (\neg x_1 \wedge \neg x_2)),$$

which becomes satisfiable. The problem here is that x_1 and x_2 are in $(\neg x_1 \wedge \neg x_2)$, which is outside the subformula we are applying the rule (d) on. Therefore, we can put in a condition that the sets of universal

variables (except the ones from D_y) from the disjuncts are disjoint and, as the next theorem shows, it is enough that universal variables from one of these disjuncts are not outside this subformula. We also need to not only look at the universal variables occurring in the disjuncts, but also at the ones from the dependency sets of existential variables occurring in the disjuncts.

Theorem 3.5. *Let $\psi, \psi_1, \psi_2 \in \mathbf{\Phi}_V^{\text{DQBF}}$ such that $\exists y(D_y)\,(\psi_1 \vee \psi_2)$ is a subformula of ψ and y' be a fresh variable not occurring in ψ. Let*

$$A_{\psi_1} = \left\{ x \in V_\psi^\forall \setminus D_y \;\middle|\; x \in V_{\psi_1} \text{ or } x \in D_{y_1} \text{ for some } y_1 \in V_{\psi_1} \cap V_\psi^\exists \right\}$$

and

$$A_{\psi_2} = \left\{ x \in V_\psi^\forall \setminus D_y \;\middle|\; x \in V_{\psi_2} \text{ or } x \in D_{y_2} \text{ for some } y_2 \in V_{\psi_2} \cap V_\psi^\exists \right\}.$$

Assume that $A_{\psi_1} \cap A_{\psi_2} = \emptyset$ and each variable x from A_{ψ_1} can only occur in ψ outside the subformula $\exists y(D_y)\,(\psi_1 \vee \psi_2)$ as quantification $\forall x$ or in a dependency set, but not in the dependency set of any existential variable in ψ that occurs outside the subformula. Then $\psi \approx \psi'$ where ψ' results from ψ by replacing the subformula $\exists y(D_y)\,(\psi_1 \vee \psi_2)$ by

$$\left(\exists y(D_y)\,\psi_1 \vee \exists y'(D_y)\,\psi_2\,[y'/y]\right).$$

Proof. We prove that $[\![\psi]\!] \neq \emptyset$ iff $[\![\psi']\!] \neq \emptyset$. Let $\phi = \exists y(D_y)\,(\psi_1 \vee \psi_2)$ and $\phi' = (\exists y(D_y)\,\psi_1 \vee \exists y'(D_y)\,\psi_2\,[y'/y])$.

First, let $s \in [\![\psi]\!]$. It is easy to see that s', where $s'(y') = s(y)$ and for other variables it maps to the same Skolem function as s, is a satisfying Skolem functions mapping of ψ', because $s(\psi)$ and $s(\psi')$ are the same BFs.

Now, assume that $s' \in [\![\psi']\!]$. In the rest of the proof, for every valuation $v \in \mathbf{A}_{V_\psi^\forall}$ (where it also holds that $v \in \mathbf{A}_{V_{\psi'}^\forall}$, because $V_\psi^\forall = V_{\psi'}^\forall$) we use v_{D_y} to denote the restriction of v to D_y. Let s be a Skolem function mapping of ψ such that for $v \in \mathbf{A}_{V_\psi^\forall}$

- $s(y)(v) = s'(y)(v)$ if for all valuations $v' \in \mathbf{A}_{V_\psi^\forall}$, for which it holds that $v'_{D_y} = v_{D_y}$ and $s'(\psi')$ depends on the subformula $s'(\phi')$ in v', we have $v'(s'(\psi_1)) = 1$,

- $s(y)(v) = s'(y')(v)$ otherwise.

For other variables, s maps to the same Skolem function as s'. We show that $s \in [\![\psi]\!]$.

Let $v \in \mathbf{A}_{V_\psi^\forall}$ be a valuation. We need to show that $v(s(\psi)) = 1$. With similar reasoning as in the proof of case (a) of Theorem 3.4, we can assume that $s'(\psi')$ depends on the subformula $s'(\phi')$ in v and $v(s'(\phi')) = 1$. Because the only difference between $s(\psi)$ and $s'(\psi')$ is in subformulas $s(\phi)$ and $s'(\phi)$, we only need to show that $v(s(\phi)) = 1$.

Because

$$s'(\phi') = s'(\exists y(D_y)\, \psi_1 \vee \exists y'(D_y)\, \psi_2\, [y'/y]) = s'(\psi_1) \vee s'(\psi_2\, [y'/y])$$

is a disjunction, it has to hold that $v(s'(\psi_1)) = 1$ or $v(s'(\psi_2\, [y'/y])) = 1$. If they are both equal to 1, then it does not matter whether $s(y)(v) = s'(y)(v)$ or $s(y)(v) = s'(y')(v)$, it will always hold that $v(s(\phi)) = 1$. If we have $v(s'(\psi_1)) = 0$ and $v(s'(\psi_2\, [y'/y])) = 1$, then from the definition of s, we get $s(y)(v) = s'(y')(v)$ and so $v(s(\psi_2)) = v(s'(\psi_2\, [y'/y])) = 1$ and therefore $v(s(\phi)) = 1$.

For the last case, $v(s'(\psi_1)) = 1$ and $v(s'(\psi_2\, [y'/y])) = 0$, we need to show that $s(y)(v) = s'(y)(v)$, because then $v(s(\psi_1)) = v(s'(\psi_1)) = 1$ therefore $v(s(\phi)) = 1$. For that, we need to show that for each valuation $v' \in \mathbf{A}_{V_\psi^\forall}$ where $v'_{D_y} = v_{D_y}$ and $s'(\psi')$ depends on $s'(\phi')$ in v', it holds $v'(s'(\psi_1)) = 1$. Assume that this does not hold meaning that there exists some v' fulfilling these conditions where $v'(s'(\psi_1)) = 0$. We now show contradiction.

Let $v'' \in \mathbf{A}_{V_\psi^\forall}$ be a valuation such that $v''(x) = v'(x)$ for all $x \in A_{\psi_1}$ and $v''(x) = v(x)$ otherwise. This means that $v''_{D_y} = v_{D_y} = v'_{D_y}$. Because $s'(\psi_1)$ contains only variables from $A_{\psi_1} \cup D_y$ we have $v''(s'(\psi_1)) = v'(s'(\psi_1)) = 0$. Moreover, because $A_{\psi_1} \cap A_{\psi_2} = \emptyset$ (from the assumption of the theorem) and $s'(\psi_2\, [y'/y])$ contains only variables from $A_{\psi_2} \cup D_y$, we have $v''(s'(\psi_2\, [y'/y])) = v(s'(\psi_2\, [y'/y])) = 0$. Together, this means $v''(s'(\phi')) = 0$.

Furthermore, because variables from A_{ψ_1} are not "outside" the subformula $s'(\phi')$ (from the assumption of the theorem) and all the "outside" variables are set to the same value as v, we can use Lemma 3.3 to show that $s'(\psi')$ depends on the subformula $s'(\phi')$ in v'' (as was

the case for v). However, because ψ' is in NNF and so $s'(\phi')$ does not have negation anywhere in front of it, we can use Lemma 3.2 to show that $v''(s'(\phi')) = v''(s'(\psi')) = 1$ (because $s' \in [\![\psi']\!]$) which is a contradiction. □

Remark. The condition that variables from A_{ψ_1} cannot occur outside the subformula $\exists y(D_y)\ (\psi_1 \vee \psi_2)$ can be replaced with the same condition for A_{ψ_2}. The proof would then have just minor differences.

3.2 Quantifier Elimination

In this section, we show how we can eliminate both universal and existential quantifiers. Firstly, we show that for a simple case where the quantified variable is not in the formula, we can just remove it.

Lemma 3.6. *Let $\psi_1 \in \mathbf{\Phi}_V^{\mathrm{DQBF}}$. Assume that variables x and y are not quantified in ψ_1 and y is not in any dependency set in ψ_1. Then*

$$\forall x\, \psi_1 \approx \psi_1^{-x}, \text{ if } x \notin V_{\psi_1} \tag{i}$$

$$\exists y(D_y)\, \psi_1 \approx \psi_1, \text{ if } y \notin V_{\psi_1}. \tag{j}$$

Let $\psi \in \mathbf{\Phi}_V^{\mathrm{DQBF}}$ be a formula such that the left side of one of the previous equisatisfiabilities is its subformula. Then $\psi \approx \psi'$ where ψ' results from ψ by replacing this subformula with the equisatisfiable right side.

Proof. We show the general case $\psi \approx \psi'$ for both equisatisfiabilities.

(i) Again, we want to prove $[\![\psi]\!] = [\![\psi']\!]$. For $s \in [\![\psi']\!]$ it is obvious that also $s \in [\![\psi]\!]$. For $s \in [\![\psi]\!]$ we can create s' where $s'(y) = s_{x=0}(y)$ where $s_{x=0}(y)$ is a Boolean function that behave like s_y where x is set to 0. Then $s' \in [\![\psi']\!]$ because for each valuation $v \in \mathbf{A}_{V_\psi^\forall}$ where $v(x) = 0$, we have $v(s'(\psi')) = v(s(\psi)) = 1$.

(j) Stems from the fact that $s(\psi)$ and $s(\psi')$ are the same BF for any $s \in \mathbb{S}_\psi$. □

The next theorem shows how to eliminate universal quantifiers generally by universal expansion. This can be always applied to any universal quantifier but by doing so the resulting DQBF can contain new copies of existential variables. The theorem is based on universal expansion for DQBFs in PNF [1, 3], here we generalise it to the case of subformulas in DQBFs in NNF.

Theorem 3.7 (Universal Expansion)**.** *Let $\psi, \psi_1 \in \mathbf{\Phi}_V^{\text{DQBF}}$ such that $\forall x\, \psi_1$ is a subformula of ψ where ψ_1 does not include any quantifications other than the ones bounded to variables from the set $E_x = \{\, y \in V_\psi^\exists \mid x \in D_y \,\}$ of all existential variables dependent on x. Let ψ_2 be a DQBF that results from ψ_1 by substituting each $y \in E_x$ with a fresh variable y' not occurring in ψ where we set $D_{y'} = D_y$. Then $\psi \approx \psi'$ where ψ' results from ψ by replacing the subformula $\forall x\, \psi_1$ by*

$$\xi = (\psi_1^{-x}\,[0/x] \wedge \psi_2^{-x}\,[1/x]).$$

Proof. Note that each $y \in E_x$ must be entirely inside ψ_1 (even the "$\exists y$" bit). This stems from the way the set $\mathbf{\Phi}_V^{\text{DQBF}}$ is constructed; the dependency sets of new existential variables can only be formed from free or completely new variables and if "$\exists y$" was outside $\forall x\, \psi_1$, then $x \in D_y$ would have to be a free or new variable. Also, because of the assumption that only quantifiers bounded to existential variables from E_x are in ψ_1 it is obvious that $\psi' \in \mathbf{\Phi}_V^{\text{DQBF}}$. With that we can now prove that $[\![\psi]\!] \neq \emptyset$ iff $[\![\psi']\!] \neq \emptyset$.

First we prove that if $[\![\psi]\!] \neq \emptyset$, then $[\![\psi']\!] \neq \emptyset$. Let $s \in [\![\psi]\!]$. We build a satisfying Skolem functions mapping s' of ψ'. For $y \in E_x$ and its copy $y' \in V_{\psi_2}^\exists$ we set $s'(y) = s_{x=0}(y)$ and $s'(y') = s_{x=1}(y)$ where $s_{x=0}(y)$ and $s_{x=1}(y)$ denote Boolean functions that behave like $s(y)$ where x is set to 0 or 1 respectively. Otherwise we set $s'(y) = s(y)$. Then s' is a Skolem functions mapping of ψ'.

To show that it is satisfying, we have to prove that BF $s'(\psi')$ is true in all valuations over $V_{\psi'}^\forall$. Notice that $V_{\psi'}^\forall = V_\psi^\forall \setminus \{x\}$. Let v be one such valuation and denote $v_{x\mapsto 0}$ and $v_{x\mapsto 1}$ valuations over $V_\psi^\forall$ such that $v(x') = v_{x\mapsto 0}(x') = v_{x\mapsto 1}(x')$ for $x' \in V_{\psi'}^\forall$, $v_{x\mapsto 0}(x) = 0$ and $v_{x\mapsto 1}(x) = 1$. It holds that $v_{x\mapsto 0}(s(\forall x\, \psi_1)) = v(s'(\psi_1^{-x}\,[0/x]))$ and $v_{x\mapsto 1}(s(\forall x\, \psi_1)) = v(s'(\psi_2^{-x}\,[1/x]))$.

Therefore, if $v_{x\mapsto 0}(s(\forall x\, \psi_1)) = v_{x\mapsto 1}(s(\forall x\, \psi_1)) = \kappa$, where $\kappa \in \{0,1\}$, then $v(s'(\xi)) = \kappa$. If $v_{x\mapsto 0}(s(\forall x\, \psi_1)) \neq v_{x\mapsto 1}(s(\forall x\, \psi_1))$ then $v(s'(\xi)) = 0$ where either $v_{x\mapsto 0}(s(\forall x\, \psi_1)) = 0$ or $v_{x\mapsto 1}(s(\forall x\, \psi_1)) = 0$. This means that $v(s'(\xi))$ is equal to $v_{x\mapsto 0}(s(\forall x\, \psi_1))$ or $v_{x\mapsto 1}(s(\forall x\, \psi_1))$. Also, because $s \in [\![\psi]\!]$, it has to hold that $v_{x\mapsto 0}(s(\psi)) = v_{x\mapsto 1}(s(\psi)) = 1$. Furthermore, the only difference in $s(\psi)$ and $s(\psi')$ are the subformulas $s(\forall x\, \psi_1)$ and $s'(\xi)$. This all means that $v(s'(\psi')) = 1$.

Now we prove the other direction. Let $s' \in [\![\psi']\!]$. We build a satisfying Skolem functions mapping s of ψ. For $y \in E_x$ with a copy $y' \in V^{\exists}_{\psi_2}$ we set $s(y)$ to Boolean function that is represented by BF $((\neg x \wedge s'(y)) \vee (x \wedge s'(y')))$ where $s'(y)$ and $s'(y')$ are replaced with BFs representing them. Otherwise we set $s(y) = s'(y)$. It is obviously a Skolem functions mapping, thus we need to prove that $s(\psi)$ is true in all valuations. Let v be a valuation over $V^{\forall}_{\psi}$. Without loss of generality assume that $v(x) = 0$. Then $v(s(y)) = v(s'(y))$. This means that $v(s(\forall x\, \psi_1)) = v(s(\psi_1^{-x}\,[0/x]))$. Again we show that $v(s(\psi)) = 1$ from the facts that $v(s'(\psi')) = 1$ (because $s' \in [\![\psi']\!]$) and that the only difference in $s(\psi)$ and $s(\psi')$ are the subformulas $s(\forall x\, \psi_1)$ and $s'(\xi)$. If $v(s'(\xi)) = 1$ then $v(s(\forall x\, \psi_1)) = v(s'(\psi_1^{-x}\,[0/x]) = 1$ and from the previous facts it means that $v(s(\psi)) = 1$. For the case when $v(s'(\xi)) = 0$ we have to realise that because ψ' is in NNF, $s'(\xi)$ occurs in $s'(\psi')$ in a tree of $\wedge$ and $\vee$ operations. That means that even if we replaced $s'(\xi)$ with $s(\forall x\, \psi_1)$ and $v(s(\forall x\, \psi_1)) = 1$, this cannot change the value of the resulting formula $v(s(\psi))$ from 1 to 0, therefore $v(s(\psi)) = 1$. □

Example 11. Let

$$\psi = \forall x_1 \forall x_2\, ((x_1 \Leftrightarrow x_2) \vee \exists y(x_2)\, \neg(x_1 \Leftrightarrow y)).$$

be the DQBF from Example 7. Using universal expansion we can transform it to

$$\psi' = \forall x_1\, ((x_1 \Leftrightarrow 0) \vee \exists y(\emptyset) \neg(x_1 \Leftrightarrow y)) \wedge ((x_1 \Leftrightarrow 1) \vee \exists y'(\emptyset)\, \neg(x_1 \Leftrightarrow y'))$$

which (like ψ) is satisfiable.

The conditions when existential elimination is possible are more strict. We can eliminate an existential variable y from subformula $\exists y(D_y)\, \psi_1$ of ψ only if ψ_1 is quantifier-free and universal variables occurring in ψ_1 or in dependency sets of existential variables in ψ_1 are all in D_y.

Theorem 3.8 ([10, Theorem 5])**.** *Let $\psi, \psi_1 \in \mathbf{\Phi}^{\mathrm{DQBF}}_V$ such that $\exists y(D_y)\, \psi_1$ is a subformula of ψ where ψ_1 does not include any quantification and includes only variables from $D_y \cup V^{\mathrm{free}}_{\psi} \cup \{\, y' \in V^{\exists}_{\psi} \mid D_{y'} \subseteq D_y \,\}$. Then $\psi \approx \psi'$ where ψ' results from ψ by replacing the subformula $\exists y(D_y)\, \psi_1$ by*

$$(\psi_1\,[0/y] \vee \psi_1\,[1/y]).$$

Example 12. Let

$$\psi = \forall x_1 \forall x_2 \exists y_1(x_1)\,(((x_1 \wedge x_2) \Leftrightarrow y_1) \vee \exists y_2(x_1, x_2)\,((x_1 \Leftrightarrow y_2) \wedge (y_1 \Leftrightarrow y_2)))$$

be a DQBF that says that either y_1 has the same value as $x_1 \wedge x_2$ (which is impossible because y_1 depends only on x_1) or there is some y_2 which is equal to both y_1 and x_1 (which is possible because both y_1 and y_2 depend on x_1). This means that this formula is satisfiable. Because the subformula $\exists y_2(x_1, x_2)\,(x_1 \Leftrightarrow y_2) \wedge (y_1 \Leftrightarrow y_2)$ fulfills the conditions from the theorem, ψ can be transformed to

$$\psi = \forall x_1 \forall x_2 \exists y_1(x_1)\,(((x_1 \wedge x_2) \Leftrightarrow y_1) \vee (((x_1 \Leftrightarrow 0) \wedge (y_1 \Leftrightarrow 0)) \vee ((x_1 \Leftrightarrow 1) \wedge (y_1 \Leftrightarrow 1))))$$

which is still satisfiable.

4 State of the Art

In this chapter, we give an overview of existing solutions and solvers for the satisfiability problem of DQBF. Because there are already multiple surveys about DQBF [21, 22, 23], we give only a short overview of most solvers and techniques. However, we go into more details for the solver HQS, because it is currently the best performing solver (winner of the DQBF track of the QBF Evaluation 2018 [12] and 2019 [13]) and the methods we use for developing our solver are based on the workings of this solver.

4.1 First Solution – DQDPLL

The first solver that tackled the satisfiability problem for DQBF was based on the DPLL algorithm [24] which is successfully used for BF and QBF solvers. This algorithm works on Boolean formulas in CNF by searching for a satisfying assignment based on the clauses in CNF. By recursively choosing an assignment for literals and checking whether the remaining clauses contain an empty one (which implies unsatisfiability), DPLL searches trough all assignments until a satisfying one is found or all of them are decided to be unsatisfying. An adaptation called DQDPLL [5] was introduced for DQBF (in PCNF) which extended existing solutions for QBF by adding so-called Skolem clauses which encode the dependencies between existential and universal variables. However, by doing so the algorithm becomes too slow and results in an uncompetitive solver.

4.2 Instantiation – iDQ and iProver

The first efficient DQBF solver, called iDQ [6], is based on an instantiation technique used for solving *Effectively Propositional Logic* (EPR) [7]. This solver also works only on formulas in PCNF where in each step of the algorithm it tries to create a BF ϕ that is an overapproximation of an input DQBF. This is done by instantiating some set of clauses, that is it applies universal expansion locally to them. This BF ϕ is then checked for satisfiability, where if ϕ is unsatisfiable then the input

DQBF is also not satisfiable, while if it is satisfiable, it must be checked if the resulting valuation is valid for the input DQBF. If it is not, then it is used to create more clause instances which are then used to refine this overapproximation.

Furthermore, a solver for EPR can also be used directly for DQBF. Because EPR belongs to the same complexity class as DQBF, there exists a polynomial-time reduction from DQBF to EPR [6]. This is used by EPR solver iProver [7] which transforms a DQBF in PCNF to an EPR instance and then solves this.

4.3 Clausal Abstraction – dCAQE

Another DQBF solver called dCAQE [11] is based on clausal abstraction [25]. This solver — working again on DQBF in PCNF — first puts universally and existentially quantified variables to some sets (called nodes) which are then divided into levels based on the ordering of dependencies of existential variables. The algorithm then constructs for each node on each level a BF that represents which clauses in CNF it can satisfy (for existential node) or falsify (for universal node). The algorithm then builds a candidate valuation by processing each level. For each level either this valuation is extended, or there is some conflict which means that the algorithm has to backtrack to some lower level and refine the abstraction, or the candidate valuation is a satisfying one which ends the algorithm.

4.4 Quantifier Elimination – HQS

The next solver is based on quantifier elimination. This solver's basic premise is simple — it iteratively chooses some universal variable for universal expansion (Theorem 3.7) thus eliminating it and then eliminates all existential quantifiers that are dependent on all leftover universal quantifiers using Theorem 3.8.

Gitina et al. [1] introduced the basic algorithm for solving DQBF in PCNF in this way. Algorithm 1 shows simple pseudocode of this. In every step, this algorithm chooses (using some heuristic) a universal variable to eliminate. The authors used a heuristic where the universal variable is chosen based on the number of existential ones that

Algorithm 1 Quantifier elimination algorithm

```
1: function SolveDQBF(DQBF ψ in PCNF)
2:     while V_ψ^∀ is not empty do
3:         choose x from V_ψ^∀
4:         ψ = ∀-expansion(x, ψ)                    ▷ Theorem 3.7
5:         for all y ∈ V_ψ^∃ s. t. D_y = V_ψ^∀ do
6:             ψ = ∃-elimination(y, ψ)              ▷ Theorem 3.8
7:         end for
8:     end while
9:     return SAT(ψ)
10: end function
```

depend on it — they choose the one that has the minimal number of dependencies. After that, all the existential variables that depend on everything are also eliminated. To use this algorithm it is important to choose a good representation of the matrix of the DQBF. For this, the authors chose and-inverter graphs [14].

Following this technique, solver HQS was introduced [8] which was enhanced by using QBF solver as a subprocedure. They still eliminate quantifiers in a similar fashion but now they do it until the formula can be transformed to QBF (that is the dependency sets are linearly ordered). On this QBF they then run already existing solver for QBF called AIGSolve [26] which can use more effective techniques developed for QBFs. Universal quantifiers to eliminate are chosen in the beginning in such a way that the number of universal eliminations is as small as possible while still the resulting formula is QBF. For this they build a dependency graph in which nodes are existential variables where y_i is connected to y_j if for their dependency sets it hold that $D_{y_i} \not\subseteq D_{y_j}$. They noticed that if this graph is acyclic, the formula can be seen as QBF. Furthermore, there is a cycle in this graph iff there is a simple cycle (between two nodes) in the graph. They use this to create an instance of MaxSAT problem which is a problem of finding a valuation of one BF while maximizing another formula. This instance then encodes which universal variables have to be eliminated so the dependency graph becomes acyclic while minimising the number of universal variables to eliminate. After finding the set of variables to

eliminate, they follow with an improved quantifier elimination algorithm. They also improved it by adding a preprocessing step (see Section 4.5) and elimination of special types of variables called unit and pure.

The algorithm was changed a bit again by Wimmer et al. [9]. The authors were interested in whether it is possible to somehow remove specific dependencies from the list of dependencies of an existential variable. They have shown that it is possible, albeit at the expense of adding a new existential variable. This is similar to the universal expansion (Theorem 3.7), but by removing just one dependency we add just one new existential variable. Let for example

$$\psi = \forall x_1 \dots \forall x_n \exists y_1(D_{y_1}) \dots \exists y_m(D_{y_m})\, \phi$$

be a prenex DQBF where ϕ is a BF and $x_1 \in D_{y_1}$. We can now remove x_1 from D_{y_1} by adding a copy of y_1 resulting in an equisatisfiable DQBF

$$\forall x_1 \dots \forall x_n \exists y_1^0(D_{y_1} \setminus \{x_1\}) \exists y_1^1(D_{y_1} \setminus \{x_1\}) \exists y_2(D_{y_2}) \dots \exists y_m(D_{y_m}) \\ \phi \left[((\neg x_1 \wedge y_1^0) \vee (x_1 \wedge y_1^1))/y_1 \right].$$

Using this the authors do not look for the set of universal variables to eliminate, but for partial dependencies whose removal will result in a DQBF which can be transformed into QBF. Again, they create a dependency graph which is acyclic iff DQBF can be seen as QBF. But now they create a bipartite graph whose two sets of nodes are the set of universal variables and the set of existential variables which are connected based on whether a universal variable is in the dependency set of existential one. Similarly to previous, they use this graph to solve an optimization problem that encodes which dependencies to remove to get QBF which can then be solved with AIGSolve.

Last but not least, Ge-Ernst et al. [10] have given theoretical foundations for non-prenex DQBFs and the possibility to push and eliminate quantifiers inside formulas as explained in Sections 3.1 and 3.2. Using this they improve HQS by adding another step in the algorithm, so-called *localisation*, where they first push quantifiers as much as possible into the formula, eliminate the existential variables using Theorem 3.8 and then return the formula to the prenex form. After

this, they start with the elimination of universal quantifiers as was explained in previous paragraphs.

4.5 Preprocessing

In previous sections, we gave an overview of techniques used for implementations of different solvers. This section explains some techniques that are used before the actual solvers run, called fittingly preprocessing.

4.5.1 HQSpre

The first and only preprocessing engine for DQBF, called HQSpre [27, 28, 29], took many techniques used for QBF preprocessing and lifted them to the DQBF case. It takes as an input a DQBF in PCNF and transforms it into a simpler DQBF. It can also sometimes solve the input formula during this simplification.

We explain here only one technique, called *gate extraction*. As was mentioned in Section 2.1.1, Tseitin transformation of BF to an equisatisfiable formula in CNF can result in a polynomial increase of the size of the formula by adding new variables. Gate extraction works in the opposite direction — it tries to find and extract these newly introduced variables and turn the formula back into one that is not in CNF. For example, if we had $(x_1 \wedge x_2)$ somewhere in the original non-CNF formula, Tseitin transformation would swap it with a new variable y and add new clauses $(\neg y \vee x_1)$, $(\neg y \vee x_2)$, and $(y \vee \neg x_1 \vee \neg x_2)$. Gate extraction then looks for these clauses and if it finds them, it removes them and replace the variable y back with $x_1 \wedge x_2$.

However, this technique can only be used by solvers which can work on DQBFs whose matrix is not in CNF. Currently, that is only HQS. For a thorough explanation of other techniques used by HQSpre, we refer the reader to the work by Wimmer et al. [29].

4.5.2 Approximations

Another technique which is used in some solvers as preprocessing step is to find some BF or QBF approximations which can be used with existing solvers for BFs/QBFs to potentially give us faster result about the

input DQBF (satisfiability for underapproximations, unsatisfiability for overapproximations). First QBF approximations used for DQBF [1] were created by changing the dependencies of existential variables that result in QBF with the same matrix. However, these approximations can be precise only up to some level. By adding some information to these QBF overapproximations (thus changing the matrix), it is possible to create more and more precise QBF overapproximations [30]. For a more thorough explanation of these approximations see again the work by Wimmer et al. [29, Section 3].

4.5.3 PSPACE Subclass

The last preprocessing technique is simply put just a check whether DQBF does not belong to a subclass of problems which are in PSPACE and if it does, it is transformed to QBF (which is PSPACE-complete) on which an existing QBF solver is run.

The simplest such subclass would be those DQBFs whose dependency sets are linearly ordered. We can check for this by comparing all the dependency sets of the same size. If they are the same, then we can check if they form a linear order and if they do reorder the quantifiers based on it, thus creating QBF.

Scholl et al. [31] investigated whether there is another DQBF subclass which is in PSPACE. They show that DQBFs where the dependency sets are either equal or pairwise disjoint can be transformed into QBFs with just linear increase in size. Thus having a check on whether an input DQBF belongs to this class before running a DQBF solver and if it does transforming it into QBF and running a QBF solver can improve the efficiency. They show this by adding this preprocessing step to HQS which increases the number of solved instances.

5 Suggested Algorithm

In this chapter, we describe the algorithm for solving the satisfiability of DQBF using BDDs. This algorithm is based on the localisation algorithm for HQS [10] which follows in a similar vein. However, we update it with the corrected procedure for localisation of quantifiers as explained in Section 3.1 and we add more thorough elimination of quantifiers using theorems from Section 3.2.

5.1 High-Level Definition

Algorithm 2 DQBF solver

```
1: function Solve(DQBF ψ in NNF as quantifier tree rooted in r)
2:     Localise(r)
3:     φ = TurnToBDD(r)
4:     EliminateAllQuantifiers(r, φ)
5:     return φ
6: end function
```

Algorithm 2 gives a high-level definition of our algorithm. In the next sections, we explain more thoroughly each part of the algorithm. It takes as an input a DQBF ψ in NNF represented using a quantifier tree. We assume that it does not contain free variables and all variables are quantified in $Q(r)$. Firstly, function Localise recursively pushes the quantifiers inside the formula by using Theorems 3.1, 3.4 and 3.5. Next, we transform this quantifier tree into a BDD by recursively turning children of each node into BDDs and then combining them. While doing this, quantifiers are pulled from formula and potentially eliminated using theorems for quantifier eliminations (Section 3.2). After that we end up with a DQBF with updated prefix $Q(r)$ with all the quantifiers (which were not eliminated) pulled back and BDD ϕ representing the matrix of the formula. Finally, the leftover quantifiers in the prefix are eliminated resulting in simple BDD which represents either 0 or 1 which indicates formula satisfiability.

5.2 Localising Quantifiers

In this section, we give an explanation of function Localise from Algorithm 2 which uses the rules of Theorem 3.1 to push quantifiers as deep inside a formula as it is possible.

Algorithm 3 Quantifier localisation

```
1:  function Localise(node n)
2:      if l(n) = ∧ then
3:          LocaliseAND(n)
4:      else if l(n) = ∨ then
5:          LocaliseOR(n)
6:      else if l(n) = v or l(n) = ¬v then
7:          if v ∈ Q_∀(n) then
8:              Q(n) = ∀v
9:          else if v ∈ Q_∃(n) then
10:             Q(n) = ∃v(D_v \ Q_∀(n))
11:         else
12:             set Q(n) to empty prefix
13:         end if
14:     end if
15:     for all n' ∈ children(n) do
16:         Localise(n')
17:     end for
18: end function
```

Algorithm 3 shows the implementation of localising quantifiers. On a nonterminal node, function Localise calls either LocaliseAND (line 3) or LocaliseOR (line 5) according to the operation assigned to the node. These functions push variables from $Q(n)$ to the children of n based on the rules of Theorem 3.1. For terminal nodes representing a literal with a variable x, it uses rules (i) and (j) of Lemma 3.6 to remove all quantifiers which are not quantifying x. After that, the algorithm recursively runs Localise on the children of n.

The implementation of LocaliseAND is shown in Algorithm 4. We first push existential variables on lines 2-13 and then universal on lines 15-25. We do it in this order because according to the rule (h) of Theorem 3.1 we can only push universal variables if there are no

Algorithm 4 Quantifier localisation for conjunction

```
1:  function LocaliseAND(n)
2:      while Q_∃(n) ≠ ∅ do
3:          y ← GetNextExistVarToPush()
4:          yChildren ← { n' ∈ children(n) | y ∈ V_n' }
5:          if yChildren = children(n) then
6:              break
7:          else
8:              n_y ← CombineChildren(yChildren, ∧)
9:              children(n) ← (children(n) \ yChildren) ∪ {n_y}
10:             PushVar(y, n_y)
11:             remove y from Q(n)
12:         end if
13:     end while
14:     U ← { x ∈ Q_∀(n) | x ∉ D_y for all y ∈ Q_∃(n) }
15:     for all x ∈ U do
16:         for all n' ∈ children(n) do
17:             if x ∈ V_n' then
18:                 PushVar(x, n')
19:                 ReplaceVarWithFreshVar(x, n')
20:             else
21:                 remove x from dependencies of all y ∈ V_n'^∃
22:             end if
23:         end for
24:         remove x from Q(n)
25:     end for
26: end function
```

existential variables in the prefix that depend on them. By pushing existential quantifiers first, the number of universal variables that we can push might increase. Also, according to (f) and (g) it does not matter in what order we push variables.

In the part that pushes existential variables, we want to apply the rule (e) for as many existential variables from $Q(n)$ as possible. On line 3 we first get the existential variable y to apply the rule on. We choose the one that occurs in fewest children of n, that is the size of the set *yChildren* on line 4 is minimal from all the existential variables

in $Q(n)$. This is because to apply the rule (e) we need to combine the children from the set *yChildren* to create a new child n_y (lines 8 and 9) and choosing the variable that combines the fewest children could possibly allow us to push other variables in more children. We use function CombineChildren(*children*, $\diamond$) which returns a new node n' where $l(n') = \diamond$, $Q(n')$ is empty prefix and $children(n') = children$. However, if *children* contains only one child n', it does not create a new node but just returns n'.

We then apply the rule (e) and push variable y into n_y (lines 10 and 11) using function PushVar(v, n'). This function pushes the variable v with its quantifier and possible dependency set to the beginning of $Q(n')$. Therefore, the new child n_y is a root of a quantifier tree with $l(n_y) = \wedge$, $Q(n_y) = \exists y(D_y)$ and $children(n_y) = yChildren$.

The sets of children containing each existential variables are actually computed before the while loop so we can easily find the variable with the minimal set. They are also only updated, not recomputed, whenever we combine children to a new child on lines 8 and 9.

The while loop finishes after either all existential variables were pushed inside or we got to a variable that is in every child (line 5) and therefore it is not possible to push it inside. Also, when we find such a variable, because all other variables have to have the sets *yChildren* at least as large as the found variable, this means that all remaining variables are also in all children.

After pushing all possible existential variables we can start pushing universal ones using the rules (a) and (b). First, we find the set U of all universal variables in $Q(n)$ such that no leftover existential variable in $Q(n)$ depends on them. According to (h), we can only push these. Therefore we push each such variable x into every child (line 18) using the rule (a), or if a child does not contain x, we can use the rule (b) and delete x from all dependency sets in the child (line 21). Here, we use function ReplaceVarWithFreshVar(x, n') which replaces each occurrence of x in the subtree rooted in n' (meaning in prefixes, dependency sets or terminal nodes) with some fresh variable not occurring anywhere else. This comes from the rule (a).

Algorithm 5 shows the function LocaliseOR. This function first tries to push existential quantifiers using the rule (d) according to Theorem 3.5 (lines 2-11). Then, using the rule (e), we try to do the same thing with leftover existential and universal variables as was

Algorithm 5 Quantifier localisation for disjunction

```
1:  function LocaliseOR(n)
2:      for all y ∈ Q∃(n) do
3:          yChildren ← { n′ ∈ children(n) | y ∈ V_n′ }
4:          if yChildren fulfill the conditions from Theorem 3.5 then
5:              for all n′ ∈ yChildren do
6:                  PushVar(y, n′)
7:                  ReplaceVarWithFreshVar(y, n′)
8:              end for
9:              remove y from Q(n)
10:         end if
11:     end for
12:     while Q(n) is not empty do
13:         v ← GetNextVarToPush()
14:         vChildren ← { n′ ∈ children(n) | v ∈ V_n′ or v ∈ D_y for
                                                    some y ∈ V_n′ }
15:         if vChildren = children(n) then
16:             break
17:         else
18:             n_v ← CombineChildren(vChildren, ∨)
19:             children(n) ← (children(n) \ vChildren) ∪ {n_v}
20:             PushVar(v, n_v)
21:             remove v from Q(n)
22:         end if
23:     end while
24: end function
```

done with existential variables in LocaliseAND: we take the variable occurring in the smallest number of children, create a new child from these children and push this variable inside (lines 12-23). However, by pushing both universal and existential variables at the same time, we need to look for universal variables which can become pushable according to the rule (h).

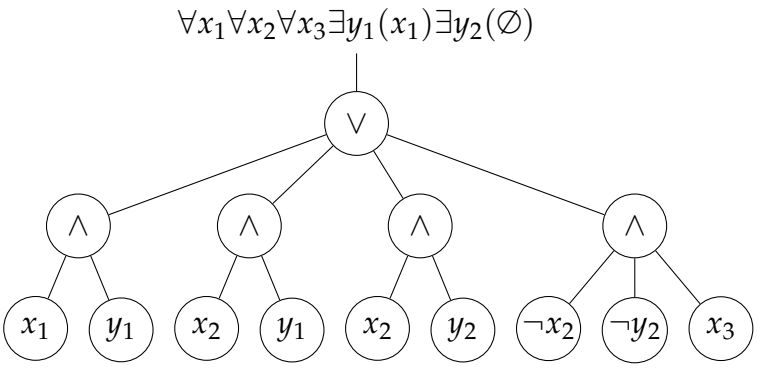
In the part where we try to apply the rule (e) according to Theorem 3.5 we go trough each existential variable y and check if it fulfils the conditions of Theorem 3.5. However, we can first apply the rule (e) to divide children of y to those that do not contain y and those that

contain it and only work with those that contain it. On line 3 we find the set of children containing y and then check if they fulfill the conditions from Theorem 3.5 (line 4). To fulfil them, we find for each child $n' \in yChildren$ the set $A_{n'}$ of universal variables which are not in D_y such that they occur in the subtree rooted in n' (as literal in terminal node) or in the dependency set of some existential variable in $V_{n'}$. If all these sets are pairwise disjoint and variables from at most one of these sets are occurring outside the subtree rooted in the corresponding node, we can push y into every child in $yChildren$ (line 6). Again, according to (d) we need to create a copy of y, which is accomplished by the call to function ReplaceVarWithFreshVar on line 7.

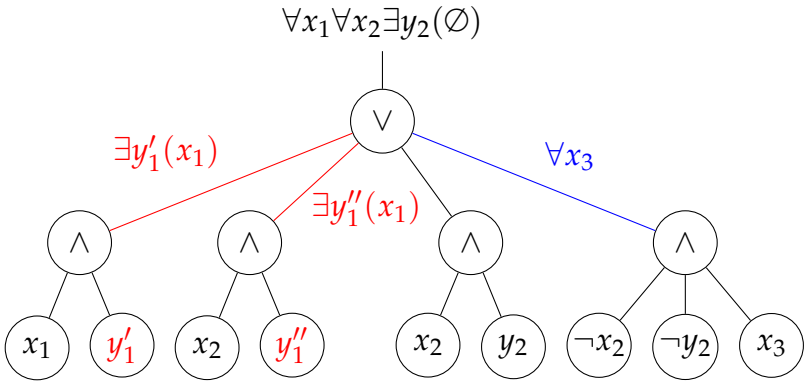
After doing this, we start applying the rules (c) for universal and (e) for existential variables similarly as was done in Algorithm 4. We do it at the same time, so we always push the variable with fewest number of children to combine. This means that function GetNextVarToPush on line 13 returns a variable for which the set $vChildren$ on line 14 has fewest number of elements. However, it does not take all universal variables into consideration, some of them might not be pushable per the rule (h). Therefore it computes the set $\{ x \in Q_\forall(n) \mid x \notin D_y \text{ for all } y \in Q_\exists(n) \}$ from which universal variables can be chosen (the same set is used on line 14 of Algorithm 4).

The set $vChildren$ is then for an existential variable, by the rule (e), the set of all children containing v, while for a universal variable, by the rule (c), a set of all children containing v or containing some existential variable dependent on v. Again, we stop when we get to a variable whose $vChildren$ is the set of all children (line 15), otherwise we create a new child combining the children from $vChildren$ (except in the case that $vChildren$ contains only one child) and push v into it (lines 18-21).

Example 13. Figure 5.1 shows an example of localisation. We start by calling LocaliseOR on the root of the tree in Figure 5.1(a). This first tries to push existential quantifiers y_1 and y_2 by using (d) and Theorem 3.5. This is only possible for y_1. For y_2, both children nodes of root that contain y_2, contain also universal variable x_2, which is not in its dependency set. In Figure 5.1(b) the aftermath of this action is shown in red. Notice that y was renamed not only in the prefix but also in terminal nodes. In blue we show where x_3 was pushed

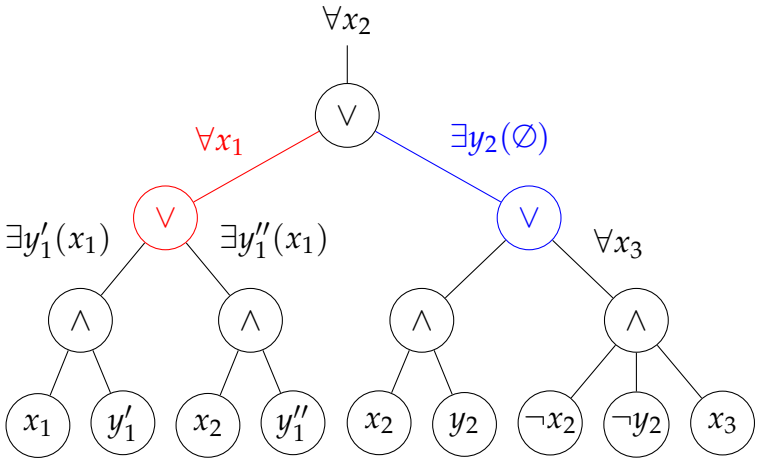


(a) The starting tree

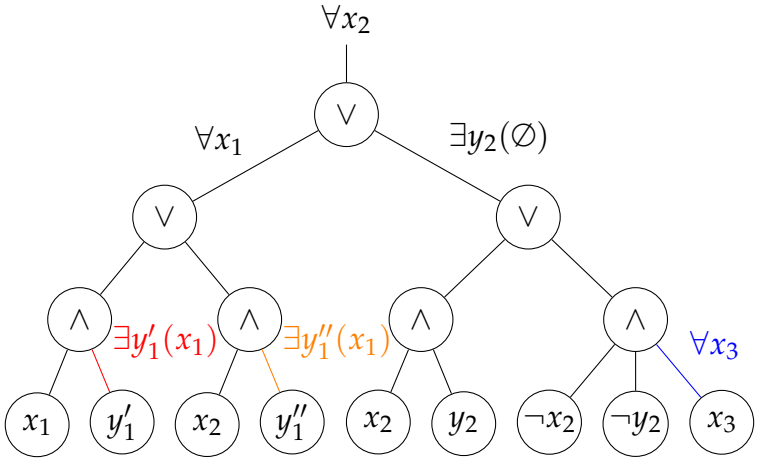


(b) After pushing y_1 and x_3

Figure 5.1: An example of localisation



(c) After localising the root



(d) The final tree

Figure 5.1: An example of localisation (cont.)

after finishing the processing of existential variables and starting the second part of the algorithm. Here, we first push x_3, because only one child contains it. Also, for the same reason, we do not create a new child for it. This is not true for y_2 and x_1 which are both in two children, therefore we have to create new children for them combining the children that contain them. In Figure 5.1(c) we can see the result of LocaliseOR for the root. In red is the newly created child to which x_1 was pushed and in blue the child to which y_2 was pushed. Variable x_2 was not pushed, because both resulting children contain it. After this, we start localising for children and their children which results in the tree in Figure 5.1(d). It shows how function LocaliseAND pushed variables for each $\wedge$ node.

5.3 Transformation to BDD

In this section, we give an overview of the function TurnToBDD which as the name suggests turns the matrix of the formula to BDD. During this, local elimination of quantifiers in subformulas can occur, which may result in a simpler final BDD.

Algorithm 6 Quantifier tree to BDD

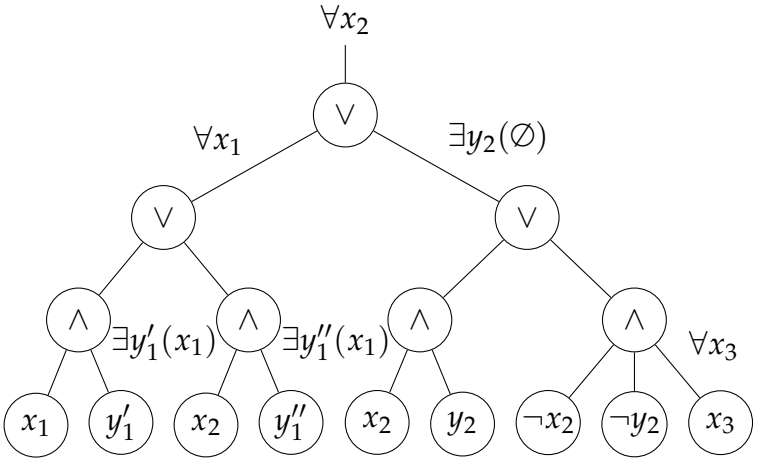
```
1:  function TurnToBDD(node n)
2:      if l(n) is a literal then
3:          return BDD for l(n)
4:      else
5:          childBDDs = ∅
6:          for all n' ∈ children(n) do
7:              φ_n' = TurnToBDD(n')
8:              EliminateQuantifiers(n', φ_n')
9:              Q(n') = RenameVarsBack(Q(n'))
10:             Q(n) = Q(n)Q(n')
11:             childBDDs = childBDDs ∪ {φ_n'}
12:         end for
13:         return ApplyOperation(l(n), childBDDs)
14:     end if
15: end function
```

Algorithm 6 shows an implementation of this function. Based on the type of the node n, it decides what to do. For terminal nodes, it just return the BDD representing the literal $l(n)$ (line 3). For non-terminal nodes, it calls itself recursively for all children of n (line 7), eliminates some variables in it (line 8), and finally pulls quantifiers from them (line 10, $Q(n)Q(n')$ is the concatenation of the two prefixes). The quantifiers can be pulled from the children based on the rules (a), (c), (d) and (e) of Theorem 3.1. Also, according to (a) and (d) we can rename the copies of variables which we created during localising back to the original name and we do so on line 9. On line 13, the algorithm returns the resulting BDD which is the application of the operation $l(n)$ on the children BDDs.

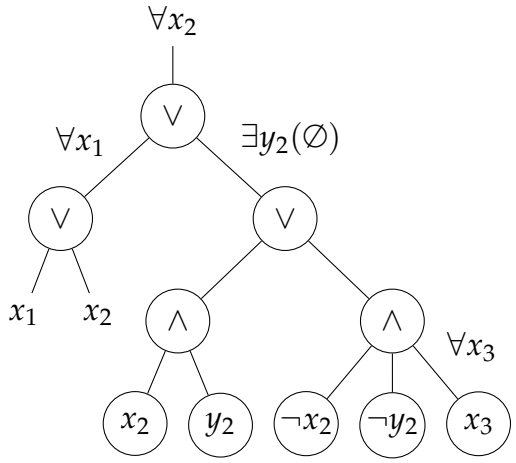
Function EliminateQuantifiers on line 8 can eliminate these quantifiers:

- **None** – we do not eliminate any variables from the BDD,
- **Simple** – we eliminate the universal variables using Theorem 3.7 only if they do not create new variables (i.e., no existential variable depends on them) and all possible existential variables using Theorem 3.8 (this elimination technique is used in HQS [10]),
- **All** – we iteratively eliminate universal variables using Theorem 3.7 and again all possible existential variables using Theorem 3.8. This is the same elimination as in function EliminateAllQuantifiers on line 4 of Algorithm 2, see Section 5.4 for details.

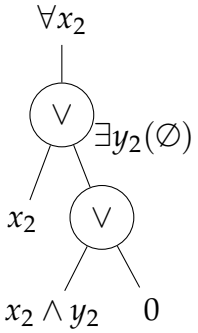
Example 14. Figure 5.2 shows the transformation to BDD for the quantifier tree from Example 13 which uses **Simple** quantifier elimination. The algorithm starts with the leftmost nodes where it transform the terminal node that represents $\exists y'_1(x_1)\, y'_1$ to $0 \vee 1 \equiv 1$ by eliminating y'_1 using Theorem 3.8. Similarly for the y'' node. Figure 5.2(b) then shows the tree after transforming the $\wedge$ nodes in the left subtree to BDDs. The nodes x_1 and x_2, which are not in circle, actually represent the BDDs which are in *childBDDs* of the left $\vee$ node. After this, we create the BDD for $x_1 \vee x_2$ and perform universal expansion for x_1 (as no existential variable depends on it) which results in $(0 \vee x_2) \wedge (1 \vee x_2) \equiv x_2$. Moving to the right subtree, we turn the left $\wedge$ node to the BDD for



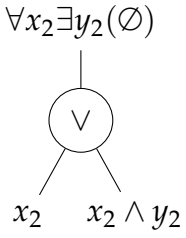
(a) The starting tree



(b) After transforming $\wedge$ nodes in the left subtree



(c) After transforming further nodes



(d) The final tree

Figure 5.2: Transformation of a quantifier tree to BDD

$x_2 \wedge y_2$. In the right one, we can eliminate x_3, resulting in BDD for $\neg x_2 \wedge \neg y_2 \wedge 0 \wedge 1 \equiv 0$. Figure 5.2(c) shows the state of the tree after these transformations. Here, we cannot eliminate y_2 in the $\vee$ node, because it contains x_2, which is not in the dependency set of y_2. That is why we extract it back to root resulting in the tree in Figure 5.2(d). And because $x_2 \vee (x_2 \wedge y_2) \equiv x_2 \wedge y_2$, the resulting BDD represents the formula $\forall x_2 \exists y_2(\emptyset)\,(x_2 \wedge y_2)$.

5.4 Quantifier Elimination

In this section we present the algorithm for eliminating all possible quantifiers. This algorithm is implemented as function EliminateAllQuantifiers, which is used in Algorithm 2 for eliminating all quantifiers in the final formula (line 4) and can also be used to eliminate quantifiers during transformation to BDDs (line 8 of Algorithm 6, it is an implementation of **All** option).

Algorithm 7 Quantifier elimination algorithm

```
1:  function EliminateAllQuantifiers(n, φ_n)
2:      while Q_∀(n) is not empty do
3:          choose x from Q_∀(n)
4:          φ_n = ∀-expansion(x, φ_n)                    ▷ Theorem 3.7
5:          add created copies of existential variables to Q(n)
6:          remove x from Q(n) and all dependency sets
7:          Q_∀ = { x ∈ V_φn | x ∈ Q_∀(n') for some ancestor[1] n' of n }
8:          Q_∃ = { y ∈ V_φn | y ∈ Q_∃(n') for some ancestor[1] n' of n }
9:          E = { y ∈ Q_∃(n) | Q_∀ ⊆ D_y and D_y' ⊆ D_y for all y' ∈ Q_∃ }
10:         for all y ∈ E do
11:             φ_n = ∃-elimination(y, φ_n)              ▷ Theorem 3.8
12:             remove y from Q(n)
13:         end for
14:     end while
15: end function
```

Algorithm 7 takes as input a node n and a BDD ϕ_n representing the matrix of DQBF given by the subtree rooted in n. This algorithm

1. we assume that n is an ancestor to itself

is similar to Algorithm 1 which is used in HQS. However, while that one was used only for DQBF in PNF, this algorithm works for any subformula. It iteratively eliminates universal variables occurring in prefix $Q(n)$ using universal expansion (Theorem 3.7) and also all possible existential variables using Theorem 3.8.

The next universal variable x to eliminate is chosen on line 3 by some heuristic after which we eliminate it from ϕ_n and remove it from the quantifier prefix of n (lines 4 and 6). We can do this, because according to (f), (g) and (h), we can reorder the quantifiers in the prefix $Q(n)$ so that at the end of the prefix we will have the variable x followed by only those existential variables that are dependent on x which is the condition of Theorem 3.7. Also, we use (e) to pull the existential variables and its created copies back to $Q(n)$ on line 5.

On lines 7 and 8 the sets $Q_\forall$ and $Q_\exists$ are initialized to all variables occurring in ϕ_n that are universally or existentially quantified in the quantifier tree. We find them by going through quantifier prefixes of all ancestors of n (we assume that n is an ancestor to itself). On line 9 we use these sets to compute the set E of all existential variables from prefix $Q(n)$ which fulfill the conditions set by Theorem 3.8: for each $y \in E$, it holds that all universal variables in ϕ_n and all universal variables in dependency sets of existential variables in ϕ_n are in D_y. The variables from E are then eliminated on lines 10-13.

We use three heuristics for choosing the next universal variable to eliminate on line 3:

- **AtBeginning**: The order of universal variables from $Q_\forall(n)$ is set at the beginning based on the number of existential variables that depend on them (from lowest to highest). We do this to keep the number of newly created existential variables low. As an example, let

  $$\forall x_1 \forall x_2 \forall x_3 \exists y_1(x_1, x_2) \exists y_2(x_1, x_2) \exists y_3(x_3) \exists y_4(x_3) \exists y_5(x_3)$$

  be the quantifier prefix $Q(n)$. The order is either x_1, x_2, x_3 or x_2, x_1, x_3, because for both x_1 and x_2 only two existential variables y_1 and y_2 depend on them while for x_3 it is three (y_3, y_4, y_5). This heuristic was introduced with HQS [8],

- **CurrentLowest**: The next heuristic is similar, but the order is not set at the beginning. The next universal variable to eliminate is

then the one with the current lowest number of existential variables that depend on it. This means that the existential variables which were created as copies during universal expansion also count here. For example, take $Q(n)$ from the previous example. In this heuristic, at the beginning, either x_1 or x_2 is chosen as in the previous one because two existential variables depend on them while on x_3 three depend. Assume that x_1 was chosen and eliminated without any other variable getting removed. The quantifier prefix $Q(n)$ is then

$$\forall x_2 \forall x_3 \exists y_1(x_2) \exists y_2(x_2) \exists y_1'(x_2) \exists y_2'(x_2) \exists y_3(x_3) \exists y_4(x_3) \exists y_5(x_3)$$

where y_1' and y_2' were created as copies of y_1 and y_2 during universal expansion. Now, in the previous heuristic x_2 would be chosen. However, here x_3 is chosen because it still depends only on three variables, while x_2 now depends on four. This heuristic was introduced by Gitina et al. [1].

- **VarsInConjuncts**: Another heuristic we use is based on the number of variables in the BDDs representing the two conjuncts used for universal expansion in Theorem 3.7. That is, let $x \in Q_\forall(n)$ and $\psi_1^{-x}\,[0/x]$, $\psi_2^{-x}\,[1/x]$ be the two conjuncts from Theorem 3.7 as if we were doing universal expansion. Because the operation of replacing variables with constants is fast for BDDs, we can create BDDs for these two formulas and check for the number of variables in both of them, that is the number of variables in the set $V_{\psi_1^{-x}[0/x]} \cup V_{\psi_2^{-x}[1/x]}$. Because BDDs remove the variables which are not needed, the number of variables in this union for BDDs can be smaller than we would expect. We can do this for all universal variables and choose the one with the lowest number of variables in the corresponding union. Notice, that this heuristic behaves like the previous one if no variables are removed in BDDs.

6 Implementation: DQBDD

We have implemented the algorithm of Chapter 5 in a new tool called DQBDD[1] (see Appendix B for documentation). The tool is written in C++ using version 3.0.0 of the CUDD[2] package [32] for BDD manipulation, version 2.2.0 of the cxxopts[3] library for parsing command-line arguments, and preprocessor HQSpre [28], more specifically the version that was used with HQS with quantifier localisation [10].

The tool takes as an input a file in `DQDIMACS` format [6] (see Appendix C) on which we can run the preprocessor HQSpre. As was explained in Section 4.5, HQSpre can extract logical gates from variables in the formula that can be then replaced by them resulting in formula with fewer variables. This results in a formula that may not be in PCNF, which most solvers cannot handle. However, as HQS, our solver can work with non-PCNF formulas, stemming from the fact that we represent DQBFs as quantifier trees. This means that we can use this feature of HQSpre similarly as it is used in HQS.

We run preprocessor twice. In the first run, we run it normally on a copy of the formula, without extracting any gates, just to check if HQSpre can solve it. If it cannot, we run the preprocessor with gate preservation set on. This will ensure that during preprocessing, no clause, from which these gates can be extracted, is touched and at the end of preprocessing, we can extract these gates while creating a quantifier tree. However, if there are just a few gates (we use less than 5), then their preservation does not make sense and we run preprocessor just once, normally, without extracting any gates. During this creation, we use the equalities from Section 2.1.2 to push negations to variables so that the resulting formula is in NNF.

We then run Algorithm 2 on the preprocessed formula with chosen heuristics for the elimination of quantifiers (Section 5.3) and for the choice of the next universal variable to eliminate (Section 5.4). If we choose **None** as the elimination heuristic as explained in Section 5.3, we can skip localisation and create BDD directly from the output

1. the source code can be found at `https://github.com/jurajsic/DQBDD` and also as an attachment of this thesis – see Appendix A
2. `https://github.com/ivmai/cudd`
3. `https://github.com/jarro2783/cxxopts`

of the preprocessor. Also, during the final elimination of quantifiers (line 4 of Algorithm 2), we can stop after the final universal variable is eliminated and we can keep all the existential variables. Then if the resulting BDD is 0, the formula is unsatisfiable. Otherwise, it is satisfiable. This stems from the fact that for unsatisfiable formulas all paths over the leftover existential variables in the resulting BDD go to 0, while for satisfiable ones, there is at least one path going to 1.

Furthermore, while applying any operation on BDDs, some variables can disappear from it, because the formula does not depend on their value. We can use this and apply the rules (i) and (j) to remove unneeded quantifiers whenever we change the formula. This can speed up quantifier elimination because some irrelevant dependencies can be removed.

As explained in Section 2.4, the size of a BDD depends on the order of the variables. This is why we also use dynamic reordering provided by CUDD which reorders the variables using some heuristics whenever the BDDs grow too big. We use CUDD's implementation of Rudell's sifting algorithm [33].

Additionally, as an optimisation, we do not create copies of variables on line 19 of Algorithm 4 or on line 7 of Algorithm 5. Even though this results in something that is not a valid DQBF, it does not cause problems as we would rename them back on line 9 of Algorithm 6 anyway. However, we do lose some ability in pushing existential variables according to Theorem 3.5, because the variables from the set A_{ψ_1} of this theorem can sometimes occur outside the subformula, while normally there would be only copies of them outside the subformula.

We can also optimise Algorithm 6 by checking on line 7 whether the child BDD corresponds to 0 (or 1). If it does and $l(n) = \wedge$ (or $\vee$), then we can directly return 0 (or 1).

7 Experimental Results

This chapter shows the experimental results of DQBDD heuristics comparison and its comparison with other publicly available DQBF solvers (iDQ v1.0[1], iProver v3.1[2], dCAQE v4.0.1[3], and HQS with the erroneous quantifier localisation[4]). All our benchmarks were run on 24 core machine with 2.10 GHz Intel Xeon CPU (with at most 6 different runs in parallel) using BenchExec v2.2 [34], a framework for reliable benchmarking and rescource measurement, where we set the runtime limit at 900 s of CPU time and the memory consumption limit at 4 GB per benchmark.

All used benchmarks with the results and BenchExec definitions can be found in a github repository[5] and as an attachment of this thesis (see Appendix A).

7.1 Benchmark Sets

We use these benchmark sets for the experimental evaluation:

PEC1 The first set consists of 1200 DQBF encodings of *partial equivalence checking* (PEC) problem [35] for incomplete circuits used by Fröhlich et al. for evaluating their DQBF solver iDQ [6]. Incomplete circuits are combinational circuits containing some missing parts, so-called black-boxes. For these we only know their input and output signals, the functionality is unknown. The partial equivalence checking problem answers the question whether there exists an implementation of black-boxes in some given incomplete circuit such that a different complete circuit is equivalent to it.

---

1. `http://fmv.jku.at/idq/`
2. `http://www.cs.man.ac.uk/~korovink/iprover/`, run with `"--qbf_mode true --inst_out_proof false --res_out_proof false"`
3. `https://github.com/ltentrup/caqe/`
4. `https://abs.informatik.uni-freiburg.de/src/projects_view.php?projectID=21`
5. `https://github.com/jurajsic/DQBFbenchmarks`

PEC2 This is another set of DQBF encodings for PEC, these are from Finkbeiner et al. [30]. This set contains 2000 instances with overproportionally many unsatisfiable ones.

PEC3 One more set of 1116 PEC encodings [1, 35].

CSP This set consists of 461 instances of *controller synthesis problem* (CSP) [36] in which we try to find a safe controller for a *controllable transition system*. A controllable transition system is composed of states and two types of inputs – uncontrollable and controllable – which decide to which state the system moves. A controller decides the controllable inputs and a *safe* controller is such a controller that the transition system always stays in a given set of safe states.

SAT The next set is created from 34 SAT instances which are transformed into equisatisfiable DQBF with an exponentially smaller number of variables [37].

E19 The last benchmark set consists of 334 instances used for evaluation of DQBF solvers at QBF Evaluation 2019 [13], which is a selected combination of previous benchmarks with some extra instances not occurring in any of the previous sets.

7.2 Preprocessing

First, we run preprocessor HQSpre on all benchmarks. Table 7.1 shows the results with the number of solved (both **S**atisfiable and **U**nsatisfiable) and unsolved (**?**) instanced by HQSpre.

We removed the solved instances from the benchmarks sets and we run the solvers only on the remaining instances in the preprocessed form. The resulting times of these runs are then a combination of HQSpre and solver runtime. Technically, we do not run HQS and DQBDD on these simplified instances, as both of these solvers incorporate HQSpre in their solving routine (as explained in Chapter 6), therefore we run them on the original instances. We still run them only on the instances that were not solved by HQSpre. The time of

Table 7.1: Results of HQSpre, where **S** is the number of solved satisfiable instances, **U** the number of solved unsatisfiable instances, and **?** the number of unsolved instances.

	PEC1	PEC2	PEC3	CSP	SAT	E19
S	86	0	25	33	4	33
U	681	0	198	6	8	55
?	433	2000	893	422	22	246

HQSpre is then not added to these solvers, because the runtime of preprocessing is included in the runtime of those two solvers.

7.3 Heuristics Comparison

Before we compare DQBDD to other solvers, it is important to choose the right heuristics for choosing the universal variables to be eliminated (**AtBeginning**, **CurrentLowest**, or **VarsInConjuncts**, see Section 5.4) and for which quantifiers to eliminate in quantifier tree during transformation to BDD (**None**, **Simple**, or **All**, see Section 5.3). For this, we compare these heuristics on E19 benchmark set as it is a selected combination of other benchmarks and it is reasonably small.

Table 7.2 shows this comparison where we run every possible combination of heuristics. It shows the number of solved instances (**#**) and from these the number of **S**atisfiable and **U**nsatisfiable instances.

For the choice of which quantifiers to eliminate, it is obvious that using **None** is not as good as the other two heuristics. For **Simple** and **All**, we examine their comparison for each heuristic for choosing the next universal variable:

- **AtBeginning** – **Simple** solved all of those that were solved by **All** and four more (all satisfiable), while there was only one instance which **All** solved faster by more than one second (it was 2.54 s faster),
- **CurrentLowest** – **Simple** solved six more instances (all satisfiable) than **All**, while **All** solved two more (both satisfiable) and from those that were solved by both, **Simple** solved faster by

Table 7.2: Comparison of DQBDD heuristics for the choice of the next universal variable to eliminate (Section 5.4) and which quantifiers to eliminate during transformation to BDD (Section 5.3) on E19 benchmark with the total number of solved instances (**#**) and the numbers of satisfiable (**S**) and unsatisfiable (**U**) from these.

Order of the next	**Quantifiers to eliminate in Algorithm 6**								
universal variable	**None**			**Simple**			**All**		
to eliminate	**#**	**S**	**U**	**#**	**S**	**U**	**#**	**S**	**U**
AtBeginning	94	41	53	139	43	96	134	38	96
CurrentLowest	94	41	53	138	42	96	134	38	96
VarsInConjuncts	94	41	53	139	43	96	135	40	95

more than one second four instances (by 1.21 s, 1.76 s, 3.33 s, and 22.68 s), while **All** two (by 5.04 s and 7.33 s),

- **VarsInConjuncts** – **Simple** solved all except one instance (satisfiable) solved by **All** with further 5 (from which one was unsatisfiable), and from those that were solved by both, **Simple** solved faster by more than one second 13 instances (by up to 177.31 s), while **All** four (by up to 3.36 s).

From these we can see that **Simple** is better than **All** and we use it in the comparison with other solvers. However, **All** was not that much worse, and there were still some cases that were solved only by **All**.

For the heuristic that chooses the next universal variable to eliminate, it is not obvious which one is the best. The numbers of solved instances are pretty much the same (for a given heuristic of which quantifiers to eliminate) across all three heuristics **AtBeginning**, **CurrentLowest**, and **VarsInConjuncts**. We can, however, compare the total runtimes of all solved instances. In the **Simple** case, we get that there are 136 instances solved by all three heuristics and for these, **AtBeginning** took 3268.9 s to solve, **CurrentLowest** took 3258.76 s, and **VarsInConjuncts** took 4018.18 s.

Furthermore, looking at the number of instances which took at least one second faster for one heuristic to solve over another, we get that

- **AtBeginning** solved 4 instance faster than **CurrentLowest** (by up to 16.08 s) and 33 faster than **VarsInConjuncts** (by up to 125.53 s),
- **CurrentLowest** solved 4 instances faster than **AtBeginning** (by up to 17.28 s) and 32 faster than **VarsInConjuncts** (by up to 129.92 s), and
- **VarsInConjuncts** solved two and one instances faster over **AtBeginning** and **CurrentLowest** respectively (by up to 15.81 s).

We can therefore decide that **CurrentLowest** is not as useful as the other two. Both **AtBeginning** and **CurrentLowest** heuristics are comparable and we have decided to use **AtBeginning** as it solved one instance more than **CurrentLowest**.

To also show that it is better to use dynamic reordering of BDD variables, we run DQBDD (with the heuristics **Simple** and **AtBeginning**) without dynamic reordering. We get that the run without dynamic reordering solved only 84 instances (with 37 satisfiable and 47 unsatisfiable), which is much less than the run with dynamic reordering. When we also run using the other two heuristics for choosing universal variable (**CurrentLowest** and **VarsInConjuncts**) the number of solved instances was even less, for both of them it was 73.

7.4 Solvers Comparison

We can now compare DQBDD with the chosen heuristics (**Simple** and **AtBeginning**) with other solvers. We show results for each benchmark set in Tables 7.3-7.8 which for each solver show the number of solved instances (**#**) and the total time of solving these. They also show the number of solved **S**atisfiable and **U**nsatisfiable instances with the number of uniquely (*****) solved instances. Furthermore, they show the reasons for unsolved instances where solvers either reached their **T**ime or **M**emory limit or they did not finish for some **O**ther reason. Only HQS sometimes does not finish for other reason, which is either an error during computation or termination of computation caused by the solver for the optimisation problem used for computing the set of universal variables to eliminate (to end up with QBF) as explained in Section 4.4.

7.4.1 PEC1

Table 7.3 shows the comparison of all solvers for PEC1 benchmarks that were not solved by HQSpre and Figure 7.2(a) shows the cactus plot for PEC1 where we plot the sorted solving times for each solver. All solvers solved most of the instances, while dCAQE solved *all* 433 instances. This is followed by DQBDD and iProver which did not solve only three or six instances respectively. For the three instances that DQBDD did not solve, it reached the memory limit during quantifier localisation. Furthermore, the total runtime of DQBDD for 424 instances that were solved by all three of these is 18.17 s, while for dCAQE it is 123.95 s and for iProver 9083.85 s. This means that, on average, it took DQBDD 0.04 s to solve one instance from these, while dCAQE took 0.29 s and iProver took 21.42 s per instance. The comparison of solving times for DQBDD and dCAQE is shown in Figure 7.1(a).

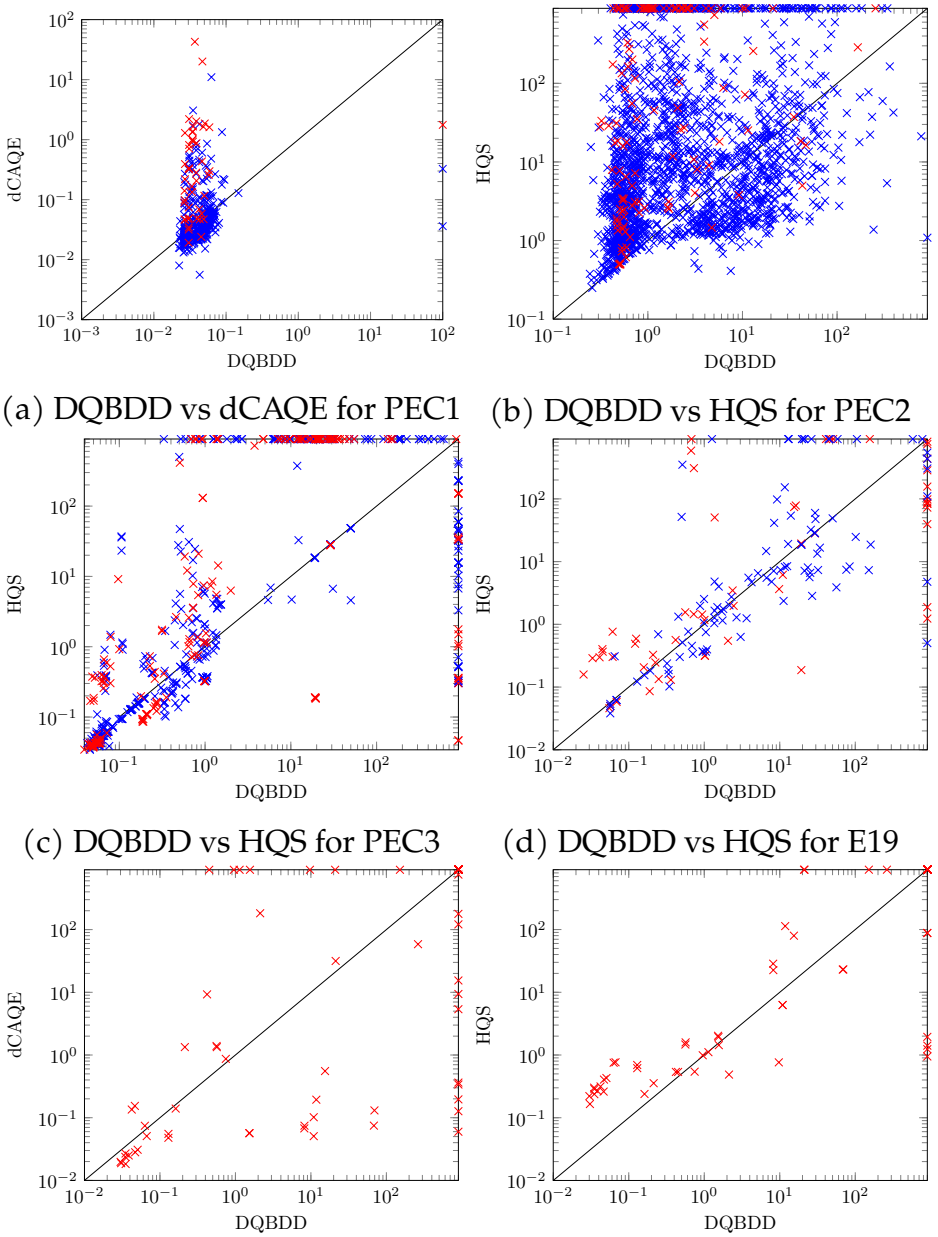
The remaining two solvers – HQS and iDQ – solved 413 instances each where HQS was 249 times faster than iDQ. Also notice that even though HQS solved fewer instances than DQBDD, DQBDD total runtime was still significantly smaller than HQS total runtime.

Table 7.3: Comparison of solvers for PEC1 benchmark set

Solver	Solved					Unsolved		
	#	Time (s)	S	U	*	T	M	O
dCAQE	**433**	134.89	**48**	**385**	0	0	0	0
DQBDD	430	18.38	47	383	0	0	3	0
HQS	413	68.06	44	369	0	2	0	18
iDQ	413	16 943.53	40	373	0	20	0	0
iProver	427	9118.53	43	384	0	6	0	0

7.4.2 PEC2

The comparison of all solvers on all 2000 benchmarks from PEC2 benchmark set (as none of them were solved by HQSpre) is shown in Table 7.4 while Figure 7.2(b) shows the cactus plot. Here, dCAQE, iDQ and iProver solved just a few instances whereas DQBDD and



(a) DQBDD vs dCAQE for PEC1

(b) DQBDD vs HQS for PEC2

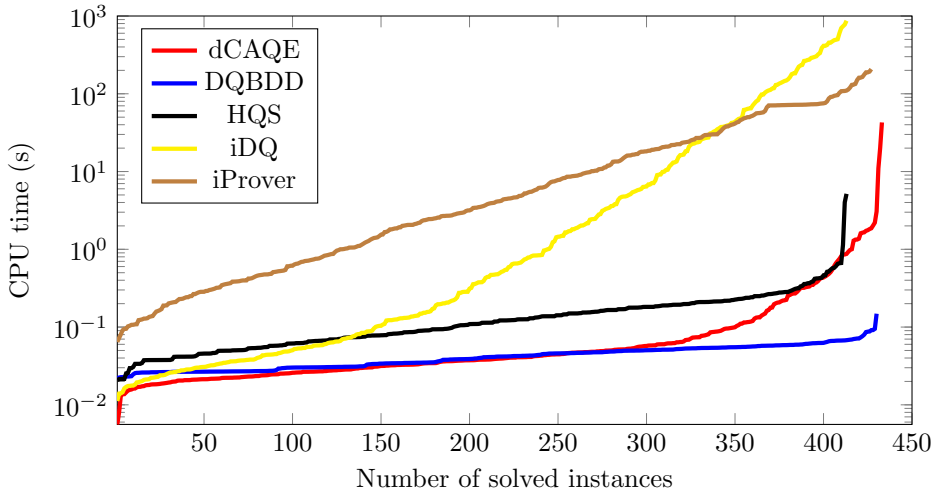
(c) DQBDD vs HQS for PEC3

(d) DQBDD vs HQS for E19

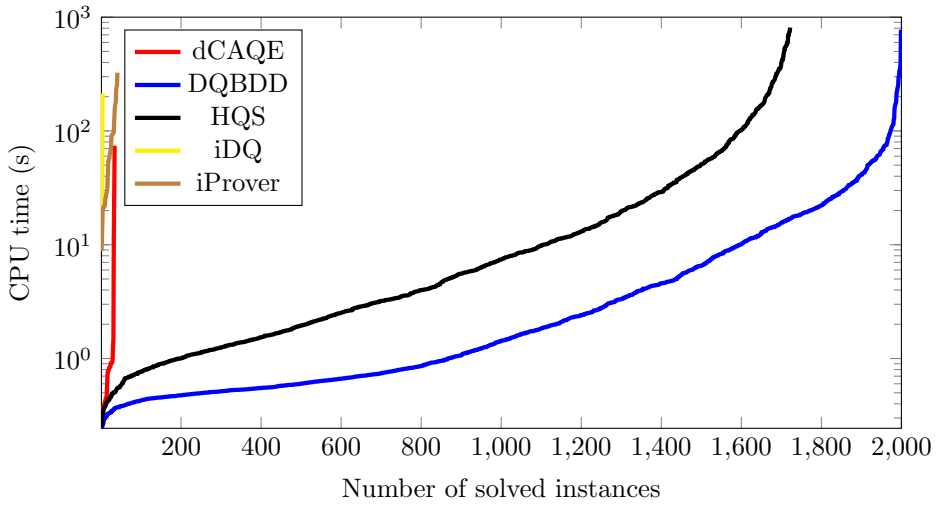
(e) DQBDD vs dCAQE for CSP

(f) DQBDD vs HQS for CSP

Figure 7.1: Comparison of solving times in seconds, where satisfiable instances are red and unsatisfiable are blue.



(a) PEC1



(b) PEC2

Figure 7.2: Cactus plots for PEC1 and PEC2

HQS solved most of them. For DQBDD, there was only one unsolved instance (caused by reaching memory limit during quantifier localisation) while HQS did not solve 278. Even with this, DQBDD was still more than twice as fast compared with HQS. Figure 7.1(b) shows the comparison of solving times of DQBDD and HQS.

The success of DQBDD and HQS for this benchmark set could be explained by gate extraction as explained in Section 4.5, which simplifies input formulas. This stems from the fact that PEC problems encode circuits, which means there are possibly many gates to extract which can result in much more simplified formula compared to the formula in PCNF.

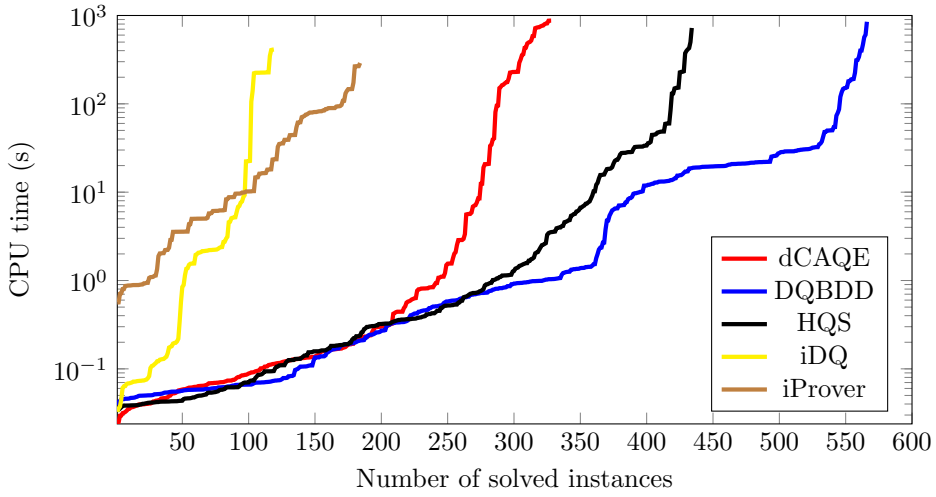
Table 7.4: Comparison of solvers for PEC2 benchmark set

Solver	Solved					Unsolved		
	#	Time (s)	S	U	*	T	M	O
dCAQE	34	134.12	0	34	0	1961	5	0
DQBDD	**1999**	19 618.36	**123**	**1876**	**278**	0	1	0
HQS	1722	51 769.75	80	1642	1	246	0	32
iDQ	4	292.66	0	4	0	1996	0	0
iProver	41	3382.37	1	40	0	9	1950	0

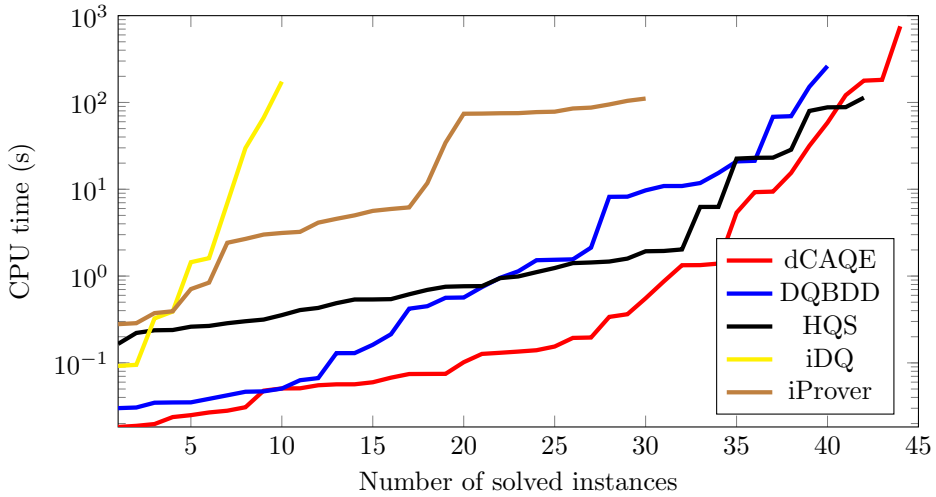
7.4.3 PEC3

The results for the last PEC benchmark set are shown in Table 7.5 and in the cactus plot in Figure 7.3(a). Again, DQBDD and HQS solved the most instances (possibly because of gate extraction) followed by dCAQE, iProver, and finally by iDQ. Furthermore, DQBDD solved 132 instances more than HQS, where, on 416 instances solved by both of them, DQBDD took 866.27 s, which is on average 2.08 s per instance, while HQS took 3427.75 s, which is on average 8.24 s per instance. Figure 7.1(c) shows the comparison of their runtimes for this benchmark set.

In this benchmark there were also three discrepancies: dCAQE solved three instances as satisfiable while HQS or DQBDD solved them as unsatisfiable (each solved two of them; iDQ and iProver did



(a) PEC3



(b) CSP

Figure 7.3: Cactus plots for PEC3 and CSP

not solve them). This cannot be explained by the erroneous implementation of quantifier localisation in HQS because it can only turn unsatisfiable formulas to satisfiable ones. As both HQS and DQBDD use similar techniques for solving, it is hard to tell whether there is a mistake in dCAQE or HQS and DQBDD. However, as we show in Appendix D, we can find a simple unsatisfiable DQBF which dCAQE determines as satisfiable. Also, because dCAQE works with the result of HQSpre without gate extraction and both HQS and DQBDD work with the result with gate extraction, the mistake could be in HQSpre. However, we tried running all three instances using DQBDD with only the preprocessing without gate extraction and it determined all three as unsatisfiable, therefore we dismiss the possibility of mistake in HQSpre. This is why we believe that these instances are unsatisfiable and dCAQE solved them incorrectly.

Table 7.5: Comparison of solvers for PEC3 benchmark set

Solver	Solved					Unsolved		
	#	Time (s)	S	U	*	T	M	O
dCAQE	328	19 271.73	96	232	3	565	0	0
DQBDD	**566**	10 293.71	**193**	**373**	**169**	222	105	0
HQS	434	6825.99	142	292	29	444	0	15
iDQ	119	4778.43	39	80	4	774	0	0
iProver	184	6769.89	55	129	0	8	701	0

7.4.4 CSP

Table 7.6 and Figure 7.3(b) shows the results and cactus plot for CSP benchmark set. For these, dCAQE, DQBDD, and HQS solved comparatively many instances (with dCAQE the most) while iProver and iDQ solved less. There are 31 instance solved by all three of those solvers, where dCAQE took 198.56 s to solve them, DQBDD took 211.84 s and HQS took 318 s. On average, this is for dCAQE 6.41 s per instance, for DQBDD 6.83 s per instance and for HQS 10.26 s per instance. Figures 7.1(e) and 7.1(f) show solving time comparison of DQBDD with dCAQE and HQS respectively.

Table 7.6: Comparison of solvers for CSP benchmark set

Solver	Solved					Unsolved		
	#	Time (s)	S	U	*	T	M	O
dCAQE	**44**	1373.30	**44**	0	**9**	378	0	0
DQBDD	40	681.86	40	0	2	334	48	0
HQS	42	504.51	42	0	1	82	0	298
iDQ	10	281.18	10	0	0	407	5	0
iProver	30	1032.42	30	0	0	0	392	0

7.4.5 SAT

The results for the SAT benchmark set are in Table 7.7. This set consists of only a few instances, therefore the comparison is not very interesting. However, we can see that DQBDD did not solve any formula from this set. This was caused by either not even finishing the creation of the quantifier tree (three cases where DQBDD reached the memory limit) or getting stuck while creating the BDD (the cases where DQBDD reached the time limit). This benchmark therefore shows that BDDs can sometimes suffer from scalability problems.

Table 7.7: Comparison of solvers for SAT benchmark set

Solver	Solved					Unsolved		
	#	Time (s)	S	U	*	T	M	O
dCAQE	7	2141.37	3	4	0	15	0	0
DQBDD	0	0	0	0	0	19	3	0
HQS	7	3030.71	**5**	2	**2**	15	0	0
iDQ	**9**	2546.24	4	**5**	1	13	0	0
iProver	8	3260.57	3	**5**	0	14	0	0

7.4.6 E19

Finally, Table 7.8 shows the comparison of solvers for E19 benchmark set with the cactus plot in Figure 7.4. As this set is a combination of previous sets (with also some new benchmarks), it is the most interesting for "real" comparison. However, this set is still dominated by PEC benchmarks, which shows up in the results: both DQBDD and HQS solved the most benchmarks, with 139 each (see Figure 7.1(d) for their runtime comparison). This is followed by dCAQE which solved 111 instances, then iProver with 50 instances, and finally iDQ with only 27 instances.

Even though DQBDD (with HQS) solved the most, it was still the fastest, taking only 3440.76 s for all of its solved instances. Looking at the 120 instances solved by both DQBDD and HQS, we get that DQBDD took 1405.04 s to solve them with 11.71 s on average while HQS took 2479.87 s with 20.67 s on average.

There was also a discrepancy in this benchmark set. One instance, which is not in other benchmark sets, was deemed satisfiable by dCAQE while HQS claimed its unsatisfiability. Again, we believe that dCAQE solved it incorrectly.

Table 7.8: Comparison of solvers for E19 benchmark set

Solver	Solved					Unsolved		
	#	Time (s)	S	U	*	T	M	O
dCAQE	111	4335.81	37	74	1	135	0	0
DQBDD	**139**	3440.76	43	**96**	**14**	90	17	0
HQS	**139**	6796.64	**51**	88	5	53	0	54
iDQ	27	3698.26	13	14	1	218	1	0
iProver	50	7490.16	23	27	0	17	179	0

7.5 Discussion

The heuristics comparison shows that for the choice of which quantifiers to eliminate it is best to use **Simple** heuristic and for the choice

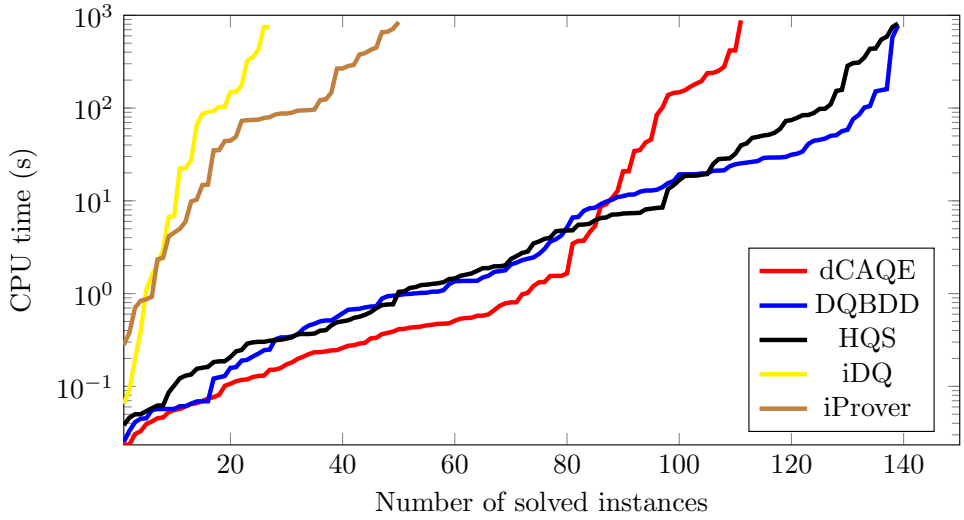


Figure 7.4: Cactus plot for E19

of the next universal variable to eliminate, it is (just a slightly) better to use **AtBeginning** heuristic.

The results also show that for PEC benchmarks, DQBDD is in the terms of total times and total solved instances (except for PEC1 benchmark set, where it solved only three instances less than dCAQE) the best solver. For CSP benchmark set, dCAQE is clearly the winner, with HQS and DQBDD not too far behind. The SAT benchmark set, where DQBDD was not able to solve anything, shows that the BDD based approach has its limits. Finally, DQBDD and HQS solved the most instances for the E19 benchmark set, with DQBDD being faster.

There were also some discrepancies between dCAQE and other solvers (three in PEC3 and one in E19) which we believe were caused by a bug in dCAQE. Furthermore, the error caused by pushing existential variables (as explained in Section 3.1) for HQS [10] was not detected in the experiments. We believe that this was caused by the data structures used in HQS, where they have something similar to quantifier trees (quantifier graphs), which can share children. They also do not push quantifier if all parents of the same child do not contain it, therefore the possible wrong pushes are limited. Note that for DQBFs in PCNF, the problem cannot occur, it can only be caused by gate extraction and usually, extracted gates have multiple parents.

8 Conclusion

In this thesis, we have proposed and implemented a DQBF solver DQBDD that uses binary decision diagrams as an underlying representation of formulas. This solver is based on quantifier elimination and uses quantifier localisation which we have corrected and enhanced by adding a possibility to eliminate universal variables in subformulas.

Additionally, we experimentally evaluated different heuristics used in DQBDD and compared the best one with other DQBF solvers. We have shown that DQBDD performs very well, especially for partial equivalence checking problem.

8.1 Future Work

For future work there are multiple things possible to do:

- We are planning to evaluate the impact of the ordering of variables in BDDs. We can check whether it would make sense to set up some initial ordering and to decide what this ordering should be. We are also planning to try different reordering techniques used for BDDs, especially those that are already implemented in CUDD. They can be used dynamically, as we do now, but we can also gauge whether it is not better to decide when exactly we should reorder the variables in the algorithm.

- Another line of thinking would be to somehow combine HQS and DQBDD, similarly how it is done in QBF solver AIGSolve [26]. This solver uses mainly AIGs to represent the matrix of the formula, but during quantifier elimination, it can be turned into BDD with some size limitations. If this BDD is created, quantifier elimination is done on it (as it is faster for BDDs than for AIGs) and then it is turned back to AIG, otherwise, quantifier elimination is done directly on AIG. We could also use AIG or some similar structure during the localisation phase, where DQBDD sometimes hits the memory limit. This structure would reuse subtrees representing the same subformula (reducing the memory usage), and they would be copied only if some variables

are being pushed into them. A similar structure is already used in HQS, called quantifier graph [10].

- We are also planning to use more preprocessing techniques, especially the QBF approximations as explained in Section 4.5.2 and the check whether an input formula can be effectively transformed into QBF as explained in Section 4.5.3. For these, we could also use some state of the art QBF solver.
- Another possible direction is to implement the quantifier elimination of only those universal variables which result in QBF as is done in HQS and possibly design and implement a QBF solver using BDDs which could solve it.

Bibliography

1. GITINA, Karina; REIMER, Sven; SAUER, Matthias; WIMMER, Ralf; SCHOLL, Christoph; BECKER, Bernd. Equivalence checking of partial designs using dependency quantified Boolean formulae. In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 2013, pp. 396–403. Available from DOI: 10.1109/ICCD.2013.6657071.
2. HENKIN, Leon. Some remarks on infinitely long formulas. In: *Infinitistic Methods: Proceedings of the Symposium on Foundations of Mathematics*. 1961, pp. 167–183.
3. BALABANOV, Valeriy; CHIANG, Hui-Ju Katherine; JIANG, Jie-Hong Roland. Henkin quantifiers and Boolean formulae: A certification perspective of DQBF. *Theoretical Computer Science*. 2014, vol. 523, pp. 86–100. Available from DOI: 10.1016/j.tcs.2013.12.020.
4. PETERSON, Gary; REIF, John; AZHAR, Salman. Lower bounds for multiplayer noncooperative games of incomplete information. *Computers & Mathematics with Applications*. 2001, vol. 41, no. 7, pp. 957–992. Available from DOI: https://doi.org/10.1016/S0898-1221(00)00333-3.
5. FRÖHLICH, Andreas; KOVÁSZNAI, Gergely; BIERE, Armin. A DPLL Algorithm for Solving DQBF. In: *Pragmatics of SAT (PoS 2012, aff. to SAT 2012)*. 2012. Available also from: http://fmv.jku.at/papers/FroehlichKovasznaiBiere-POS12.pdf.
6. FRÖHLICH, Andreas; KOVÁSZNAI, Gergely; BIERE, Armin; VEITH, Helmut. iDQ: Instantiation-Based DQBF Solving. In: *POS-14. Fifth Pragmatics of SAT workshop*. 2014, pp. 103–116. Available from DOI: 10.29007/1s5k.
7. KOROVIN, Konstantin. iProver – An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In: *Automated Reasoning*. 2008, pp. 292–298. Available from DOI: 10.1007/978-3-540-71070-7_24.

8. GITINA, Karina; WIMMER, Ralf; REIMER, Sven; SAUER, Matthias; SCHOLL, Christoph; BECKER, Bernd. Solving DQBF through quantifier elimination. In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2015, pp. 1617–1622. Available from DOI: 10.7873/DATE.2015.0098.

9. WIMMER, Ralf; KARRENBAUER, Andreas; BECKER, Ruben; SCHOLL, Christoph; BECKER, Bernd. From DQBF to QBF by Dependency Elimination. In: *Theory and Applications of Satisfiability Testing – SAT 2017*. 2017, pp. 326–343. Available from DOI: 10.1007/978-3-319-66263-3_21.

10. GE-ERNST, Aile; SCHOLL, Christoph; WIMMER, Ralf. Localizing Quantifiers for DQBF. In: *2019 Formal Methods in Computer Aided Design (FMCAD)*. 2019, pp. 184–192. Available from DOI: 10.23919/FMCAD.2019.8894269.

11. TENTRUP, Leander; RABE, Markus N. Clausal abstraction for DQBF. In: *Theory and Applications of Satisfiability Testing – SAT 2019*. 2019, pp. 388–405. Available from DOI: 10.1007/978-3-030-24258-9_27.

12. PULINA, Luca; SEIDL, Martina. *QBF Evaluation 2018* [online]. 2018 [visited on 2020-03-14]. Available from: http://www.qbflib.org/event_page.php?year=2018.

13. PULINA, Luca; SEIDL, Martina; SHUKLA, Ankit. *QBF Evaluation 2019* [online]. 2019 [visited on 2020-03-14]. Available from: http://www.qbflib.org/event_page.php?year=2019.

14. MISHCHENKO, Alan; CHATTERJEE, Satrajit; BRAYTON, Robert. *FRAIGs: A unifying representation for logic synthesis and verification*. 2005. Available also from: https://people.eecs.berkeley.edu/~alanmi/publications/2005/tech05_fraigs.pdf. Technical report. EECS Dept., UC Berkeley.

15. BRYANT, Randal E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*. 1986, vol. 35, no. 8, pp. 677–691. Available from DOI: 10.1109/TC.1986.1676819.

16. TSEITIN, Grigorii Samuilovich. On the Complexity of Derivation in Propositional Calculus. In: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. 1983, pp. 466–483. Available from DOI: 10.1007/978-3-642-81955-1_28.

17. EGLY, Uwe; SEIDL, Martina; TOMPITS, Hans; WOLTRAN, Stefan; ZOLDA, Michael. Comparing Different Prenexing Strategies for Quantified Boolean Formulas. In: *Theory and Applications of Satisfiability Testing*. 2004, pp. 214–228. Available from DOI: 10.1007/978-3-540-24605-3_17.

18. RABE, Markus N. A Resolution-Style Proof System for DQBF. In: *Theory and Applications of Satisfiability Testing – SAT 2017*. 2017, pp. 314–325. Available from DOI: 10.1007/978-3-319-66263-3_20.

19. BOLLIG, Beate; WEGENER, Ingo. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*. 1996, vol. 45, no. 9, pp. 993–1002. Available from DOI: 10.1109/12.537122.

20. ANDERSEN, Henrik Reif. *An Introduction to Binary Decision Diagrams*. IT University of Copenhagen, 1999. Available also from: https://www.cmi.ac.in/~madhavan/courses/verification-2011/andersen-bdd.pdf. Lecture notes for Efficient Algorithms and Programs.

21. KOVÁSZNAI, Gergely. *A Survey on DQBF: Formulas, Applications, Solving Approaches*. 2015. Available also from: http://fmv.jku.at/quantify15/Kovasznai_QUANTIFY2015.pdf. Talk given at International Workshop on Quantification – QUANTIFY 2015.

22. KOVÁSZNAI, Gergely. What is the state-of-the-art in DQBF solving. In: *MaCS-16. Joint Conference on Mathematics and Computer Science*. 2016. Available also from: http://ceur-ws.org/Vol-2046/kovasznai.pdf.

23. SCHOLL, Christoph; WIMMER, Ralf. Dependency Quantified Boolean Formulas: An Overview of Solution Methods and Applications. In: *Theory and Applications of Satisfiability Testing – SAT 2018*. 2018, pp. 3–16. Available from DOI: 10.1007/978-3-319-94144-8_1.

24. DAVIS, Martin; LOGEMANN, George; LOVELAND, Donald. A Machine Program for Theorem-Proving. *Commun. ACM*. 1962, vol. 5, no. 7, pp. 394–397. Available from DOI: 10.1145/368273.368557.

25. RABE, Markus N.; TENTRUP, Leander. CAQE: A Certifying QBF Solver. In: *Proceedings of the 15th Conference on Formal Methods in Computer-aided Design* (*FMCAD'15*). 2015, pp. 136–143. Available from DOI: 10.1109/FMCAD.2015.7542263.

26. PIGORSCH, Florian; SCHOLL, Christoph. An AIG-based QBF-solver using SAT for preprocessing. In: *Design Automation Conference*. 2010, pp. 170–175. Available from DOI: 10.1145/1837274.1837318.

27. WIMMER, Ralf; GITINA, Karina; NIST, Jennifer; SCHOLL, Christoph; BECKER, Bernd. Preprocessing for DQBF. In: *Theory and Applications of Satisfiability Testing – SAT 2015*. 2015, pp. 173–190. Available from DOI: 10.1007/978-3-319-24318-4_13.

28. WIMMER, Ralf; REIMER, Sven; MARIN, Paolo; BECKER, Bernd. HQSpre – An Effective Preprocessor for QBF and DQBF. In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2017, pp. 373–390. Available from DOI: 10.1007/978-3-662-54577-5_21.

29. WIMMER, Ralf; SCHOLL, Christoph; BECKER, Bernd. The (D)QBF Preprocessor HQSpre – Underlying Theory and Its Implementation. *Journal on Satisfiability, Boolean Modeling and Computation*. 2019, vol. 11, pp. 3–52. Available from DOI: 10.3233/SAT190115.

30. FINKBEINER, Bernd; TENTRUP, Leander. Fast DQBF Refutation. In: *Theory and Applications of Satisfiability Testing – SAT 2014*. 2014, pp. 243–251. Available from DOI: 10.1007/978-3-319-09284-3_19.

31. SCHOLL, Christoph; JIANG, Jie-Hong Roland; WIMMER, Ralf; GE-ERNST, Aile. A PSPACE Subclass of Dependency Quantified Boolean Formulas and Its Effective Solving. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. 2019, vol. 33, pp. 1584–1591. No. 1. Available from DOI: 10.1609/aaai.v33i01.33011584.

32. SOMENZI, Fabio. *CUDD: CU Decision Diagram Package Release 3.0.0*. Department of Electrical, Computer, and Energy Engineering, University of Colorado at Boulder, 2015. Available also from: https://pdfs.semanticscholar.org/cb3c/a92ebe93b1076aef5fcd6c8f215a06694424.pdf.

33. RUDELL, Richard. Dynamic variable ordering for ordered binary decision diagrams. In: *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. 1993, pp. 42–47. Available from DOI: 10.1109/ICCAD.1993.580029.

34. WENDLER, Philipp; BEYER, Dirk. *sosy-lab/benchexec: Release 2.2*. 2019. Version 2.2. Available from DOI: 10.5281/zenodo.3463154.

35. SCHOLL, Christoph; BECKER, Bernd. Checking equivalence for partial implementations. In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. 2001, pp. 238–243. Available from DOI: 10.1145/378239.378471.

36. BLOEM, Roderick; KÖNIGHOFER, Robert; SEIDL, Martina. SAT-Based Synthesis Methods for Safety Specs. In: *Lecture Notes in Computer Science*. 2014, pp. 1–20. Available from DOI: https://doi.org/10.1007/978-3-642-54013-4_1.

37. BALABANOV, Valeriy; JIANG, Jie-Hong Roland. *Reducing satisfiability and reachability to DQBF*. 2015. Available also from: http://fmv.jku.at/qbf15/qbf15-summary.pdf. Talk given at International Workshop on Quantified Boolean Formulas – QBF 2015.

38. BIERE, Armin. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*. 2008, vol. 4, pp. 75–97. Available also from: http://fmv.jku.at/papers/Biere-JSAT08.pdf.

39. *GNU Lesser General Public License*. Free Software Foundation, 2007. Version 3. Available also from: https://www.gnu.org/licenses/lgpl-3.0.html.

40. *QDIMACS standard*. 2005. Version 1.1. Available also from: http://www.qbflib.org/qdimacs.html.

A Attached Files

As a part of this thesis, these attachments, which can be found in the Information System of Masaryk University as well as linked github repositories, are included:

- `dqbdd.zip`[1] – the source code of DQBDD,
- `benchmarks.zip`[2] – the used benchmark sets, benchmarks definitions for BenchExec, and raw results,
- `thesis.zip`[3] – the LaTeX source of this thesis.

1. `https://github.com/jurajsic/DQBDD`
2. `https://github.com/jurajsic/DQBFbenchmarks`
3. `https://github.com/jurajsic/mastersthesis`

B Documentation for DQBDD

DQBDD is a dependency quantified Boolean formula (DQBF) solver that uses binary decision diagrams (BDDs) as an underlying representation of formulas. It is written in C++ and it reads DQBFs encoded in `DQDIMACS` format [6] for which it checks their satisfiability using quantifier elimination. The source codes with binaries can be found at `https://github.com/jurajsic/DQBDD`.

B.1 Dependencies

There is no need to install any dependency, all of them are compiled with DQBDD. They are these:

- antom[1] – SAT solver used in HQSpre,
- CUDD v3.0.0[2] [32] – BDD library,
- cxxopts v2.2.0[3] – argument parser
- Easylogging++ v9.96.7[4] – C++ logger used in HQSpre,
- HQSpre[5] [29] – DQBF preprocessor,
- PicoSAT[6] [38] – SAT solver used in HQSpre.

B.2 Installation

To compile DQBDD, a C++ compiler supporting C++14 standard and CMake[7] is needed.

---

1. `https://projects.informatik.uni-freiburg.de/projects/antom`
2. `https://github.com/ivmai/cudd`
3. `https://github.com/jarro2783/cxxopts`
4. `https://github.com/zuhd-org/easyloggingpp`
5. `https://abs.informatik.uni-freiburg.de/src/projects_view.php?projectID=21`
6. `http://fmv.jku.at/picosat/`
7. `https://cmake.org/`

Execute

```
mkdir Release
cd Release
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

to build DQBDD which will be located in `Release/src/`.

B.3 Usage

Run DQBDD with

```
DQBDD [OPTION...] <input file>
```

where `<input file>` should be formula to solve in DQDIMACS format. The possible options are

```
-h, --help                    Print usage
-v, --version                 Print the version number
-l, --localise arg            Use quantifier tree with
                              localisation of
                              quantifiers (default: 1)
-p, --preprocess arg          Use preprocessing
                              (default: 1)
-e, --elimination-choice arg  Decide what to eliminate on
                              each level of quantifier
                              tree during transformation
                              to formula (default: 1)
-u, --uvar-choice             The heuristics by which the
                              next universal variable for
                              elimination is chosen
                              (default: 0)
-d, --dyn-reordering          Allow dynamic reordering of
                              variables in BDDs
                              (default: 1)
```

where for `--localise`, `--preprocess` and `--dyn-reordering` the arguments are 0 or 1 to turn off or on the corresponding option. The option `--elimination-choice` is used only if we localise and then we have these possible arguments:

- 0 – nothing is eliminated (**None**),
- 1 – simple quantifiers are eliminated (**Simple**),
- 2 – all possible quantifiers are eliminated (**All**).

For the option `--uvar-choice` we have:

- 0 – the order of the universal variables is set at beginning (**AtBeginning**),
- 1 – the variable with the currently lowest number of dependencies is chosen (**CurrentLowest**),
- 2 – the order is based on the number of variables in conjuncts of universal expansion (**VarsInConjuncts**).

After finishing, DQBDD outputs `SAT` with exitcode 10 for satisfiable formula and `UNSAT` with exitcode 20 for unsatisfiable one.

B.4 Examples

Running DQBDD as

- `DQBDD file.dqdimacs`

solves the formula in `file.dqdimacs` with the default settings,

- `DQBDD --preprocess 0 --dyn-reordering 0 file.dqdimacs`

solves the formula in `file.dqdimacs` without running the preprocessor HQSpre first and without using dynamic reordering of variables in BDDs as implemented in CUDD,

- `DQBDD --localise 0 --uvar-choice 1 file.dqdimacs`

solves the formula in `file.dqdimacs` without localising quantifiers (or creating quantifier tree) where the next universal variable for universal expansion is always the one that has the minimal number of dependent existential variables,

- `DQBDD --localise 1 --elimination-choice 2 file.dqdimacs`

solves the formula in `file.dqdimacs` with localising quantifiers where it eliminates all universal and possible existential variables while creating the final BDD from the quantifier tree.

B.5 Licence

DQBDD is licensed under version 3 of GNU Lesser General Public License [39].

C DQDIMACS Format

In this appendix, we define the DQDIMACS format [6], which can represent DQBFs in PCNF.

C.1 Syntax

The format is defined by this BNF grammar (based on the one for QDIMACS format [40]):

```
<input> ::= <preamble> <prefix> <matrix> EOF

<preamble>      ::= [<comment_lines>] <problem_line>
<comment_lines> ::= <comment_line> <comment_lines>
                  | <comment_line>
<comment_line>  ::= c <text> EOL
<problem_line>  ::= p cnf <pnum> <pnum> EOL

<prefix>     ::= [<quant_sets>]
<quant_sets> ::= <quant_set> <quant_sets> | <quant_set>
<quant_set>  ::= <quantifier> <atom_set> 0 EOL
<quantifier> ::= e | a | d
<atom_set>   ::= <pnum> <atom_set> | <pnum>

<matrix>      ::= <clause_list>
<clause_list> ::= <clause> <clause_list> | <clause>
<clause>      ::= <literal> <clause> | <literal> 0 EOL
<literal>     ::= <num>

<text>  ::= {A sequence of non-special ASCII characters}
<num>   ::= {A 32-bit signed integer different from 0}
<pnum>  ::= {A 32-bit signed integer greater than 0}
```

where the terminal symbols EOL and EOF stand for the end-of-line and end-of-file markers respectively, [<expr>] denotes that <expr> is an optional expression, and concatenation of expressions has prece-

dence over choice between expressions. Furthermore, we have these additional constraints:

- The first <pnum> in <problem_line> corresponds to the maximal possible number of distinct atoms appearing in the prefix and the matrix, i.e., all the members of the quantified sets and the absolute value of each literal in all the clauses is less than or equal to this number.
- The second <pnum> in <problem_line> corresponds to the number of clauses appearing in the matrix, i.e., the number of <clause> expressions which <matrix> is comprised of.
- Each atom in the prefix must appear in the matrix.
- It is not possible to have two <quant_set> after each other, where both have a or both have e as <quantifier>.
- The same atom cannot appear in the prefix more than once except for atoms occurring in the second and further place of atom_set with d as <quantifier>. These atoms have to appear somewhere before in the prefix in <quant_set> with a quantifier.

C.2 Semantics

The prefix represents the DQBF prefix where atoms appearing in <atom_set> following <quantifier> equal to

- a are universally quantified variables,
- e are existentially quantified variables where each of them is dependent on all universally quantified variables which already appeared before and
- d are an existential variable with universal variables in its dependency set, that is the first atom is the existential variable y and all others are universal variables in the dependency set D_y.

The matrix represents the BF in CNF with clauses in <clause_list> where the number n in <literal> represents

either the variable x_n if it is positive or $\neg x_n$ if it is negative. Furthermore, all variables which appear in the matrix but do not appear in the prefix are existentially quantified with empty dependency sets.

Note, that `DQDIMACS` is, by removing `d` quantifiers, backwards compatible with `QDIMACS` (used for QBFs) format and, by removing the whole prefix, with `DIMACS` (used for BFs) format.

C.3 An Example

As an example, we have that

```
c Simple example
p cnf 5 2
a 1 2 0
e 3 0
d 4 1 0
-1 2 3 0
-2 -4 5 0
```

represents DQBF

$$\forall x_1 \forall x_2 \exists x_3(x_1, x_2) \exists x_4(x_1) \exists x_5(\emptyset)\ ((\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_4 \vee x_5)).$$

D DQBF Wrongly Solved by dCAQE

The DQBF

$$\begin{aligned}\forall x_1 \forall x_2 \forall x_3 \forall x_6 \exists y_4(x_1, x_2, x_3) \exists y_5(x_1, x_2, x_3) \exists y_7(x_6) \exists y_8(x_6) \\ ((\neg x_1 \vee x_2 \vee x_3 \vee y_4 \vee y_5) \\ \wedge (\neg x_1 \vee x_2 \vee x_3 \vee \neg y_4) \\ \wedge (\neg x_1 \vee x_2 \vee x_3 \vee \neg y_5 \vee y_7) \\ \wedge (x_1 \vee \neg x_2 \vee x_3 \vee y_4 \vee y_5) \\ \wedge (x_1 \vee \neg x_2 \vee x_3 \vee \neg y_4 \vee y_8) \\ \wedge (x_1 \vee \neg x_2 \vee x_3 \vee \neg y_5) \\ \wedge (x_1 \vee x_2 \vee x_3 \vee y_4 \vee y_5) \\ \wedge (x_1 \vee x_2 \vee x_3 \vee \neg y_4 \vee \neg x_6 \vee \neg y_7) \\ \wedge (x_1 \vee x_2 \vee x_3 \vee \neg y_5 \vee x_6 \vee \neg y_8))\end{aligned}$$

which in `DQDIMACS` format has the form

```
p cnf 8 9
a 1 2 3 0
e 4 5 0
a 6 0
d 7 6 0
d 8 6 0
-1  2 3  4  5           0
-1  2 3 -4              0
-1  2 3    -5     7     0
 1 -2 3  4  5           0
 1 -2 3 -4            8 0
 1 -2 3    -5          0
 1  2 3  4  5           0
 1  2 3 -4    -6  -7    0
 1  2 3    -5  6     -8 0
```

is deemed as satisfiable by dCAQE. However, it is not satisfiable, as

- the first three clauses force x_7 to be true no matter what x_6 is evaluated to,

- the fourth to sixth clauses force x_8 to be true too, and
- the last three clauses then say that x_4 and x_5 would need to be dependent on x_6, which is not the case.

Other solvers (DQBDD, HQS, iDQ, iProver, and also preprocessor HQSpre) correctly decide its unsatisfiability.