JOHANNES KEPLER UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche Fakultät

Dependency Schemes and Search-Based QBF Solving: Theory and Practice

DISSERTATION

zur Erlangung des akademischen Grades

Doktor

im Doktoratsstudium der

Technischen Wissenschaften

Eingereicht von:

Dipl.-Ing. Florian Lonsing

Angefertigt am:

Institut für Formale Modelle und Verifikation
Johannes Kepler Universität
Altenbergerstr. 69
4040 Linz
Österreich

Beurteilung:

Univ.-Prof. Dr. Armin Biere (Betreuung)
Ao. Univ.-Prof. Dr. Uwe Egly

Mitwirkung:

Assist.-Prof. Dr. Martina Seidl

Linz, März 2012

Abstract

The logic of *quantified Boolean formulae (QBF)* extends propositional logic with universal quantification over propositional variables. The presence of universal quantifiers in QBF does not add expressiveness, but often allows for more compact encodings of problems. From a theoretical point of view, the decision problems of propositional logic (SAT) and QBF are NP-complete and PSPACE-complete, respectively. Compared to SAT, which successfully has been used for practical applications in model checking or formal verification, for example, empirical studies show that current approaches to QBF solving do not scale well in practice.

The quantifier prefix of QBFs in *prenex conjunctive normal form (PCNF)* imposes a linear ordering on the variables. In general, the ordering of the prefix gives rise to *dependencies* between variables which are differently quantified. Variable dependencies restrict the freedom of QBF solvers and must be respected during semantical evaluation to avoid incorrect results.

We consider *dependency schemes*, which were introduced in related work, to overcome the drawbacks of quantifier prefixes in PCNFs. A dependency scheme is a binary relation over the set of variables of a PCNF which expresses independence between variables. If two variables are independent then a search-based QBF solver can safely assign them in arbitrary order. Thus independence increases the freedom for QBF solvers.

We analyze theoretical properties of different dependency schemes which can be computed by analyzing the syntactic structure of a PCNF. We show that the common approach of *mini-scoping* is not optimal among syntactic methods of dependency analysis. As an alternative, we introduce specific approaches to compute and represent the *standard dependency scheme* efficiently. As a byproduct, we obtain *compact dependency graphs* as a representation of arbitrary dependency schemes. A main contribution of this work is the *combination of arbitrary dependency schemes and search-based QBF solvers* relying on the QDPLL algorithm. This way, QDPLL can profit from independence of variables which otherwise is hidden by the quantifier prefix. We implemented the solver DepQBF which tightly integrates dependency schemes. Experimental results confirm the potential benefits for practical QBF solving in contrast to quantifier prefixes. Our results motivate further research on dependency schemes for applications in QBF solving.

Zusammenfassung

Die Logik *quantifizierter Boolescher Formeln (QBF)* stellt eine Erweiterung der klassischen Aussagenlogik dar, bei der die in der Formel auftretenden, aussagenlogischen Variablen existentiell oder universell quantifiziert sind. Auch wenn der Einsatz von Quantoren in QBF nicht zu einer höheren Ausdrucksstärke dieser Sprache führt, so lassen sich dadurch Kodierungen von Problemstellungen meist kompakter darstellen. Hinsichtlich der Komplexität ist das Entscheidungsproblem der Aussagenlogik (SAT) NP-vollständig während jenes für QBF PSPACE-vollständig ist. In der Praxis kommen heute im Bereich der formalen Verifikation oder Modellprüfung unterschiedliche, auf SAT basierende Verfahren zum Einsatz, deren Anwendung durch effiziente Entscheidungsverfahren für SAT erst ermöglicht wurde. Im Gegensatz dazu verhindert die in Fallstudien zu beobachtende mangelnde Effizienz aktueller Entscheidungsverfahren für QBF eine umfassende praktische Anwendung.

Hinsichtlich QBF betrachten wir Formeln in *PKNF*, also Formeln, die aus einem quantorenfreien Teil in *konjunktiver Normalform (KNF)* und einem separaten *Quantorenpräfix* bestehen. Die lineare Anordnung der quantifizierten Variablen im Präfix führt zu Abhängigkeiten zwischen Variablen unterschiedlichen Quantorentyps in einer PKNF. Variablenabhängigkeiten schränken die Freiheit von suchbasierten Entscheidungsverfahren für QBF insofern ein, als die Bewertungsreihenfolge der Variablen der Präfixordnung genügen muss und eine Nichtbeachtung dieser Bedingung falsche Auswertungsergebnisse zur Folge haben kann.

Wir versuchen, die durch das Quantorenpräfix einer PKNF hervorgerufenen Einschränkungen anhand sogenannter *Abhängigkeitsschemata* (engl. *dependency schemes*), welche in verwandten Arbeiten eingeführt wurden, zu überwinden. Ein Abhängigkeitsschema ist eine binäre Relation über der Variablenmenge einer gegebenen PKNF, welche Unabhängigkeit von Variablen ausdrückt. Zwei voneinander unabhängige Variablen können in einer suchbasierten semantischen Auswertung der PKNF in beliebiger Reihenfolge bewertet werden. Somit erhöht die Unabhängigkeit von Variablen in einer PKNF also die Freiheiten von Entscheidungsverfahren.

Wir untersuchen die theoretischen Eigenschaften verschiedener Abhängigkeitsschemata, welche mittels einer Analyse der syntaktischen Struktur einer PKNF berechnet werden können. Wir zeigen, dass bekannte Verfahren zur Antipränexierung (engl. *mini-scoping* oder *anti-prenexing*), wenn diese zur Analyse von Variablenabhängigkeiten eingesetzt werden, nicht in der Lage sind, jene volle Information über Unabhängigkeit von Variablen zu ermitteln, welche durch syntaktische Analyse im Grunde gewonnen werden kann. Stattdessen schlagen wir vor, das sogenannte *Standardabhängigkeitsschema* (engl. *standard dependency scheme*) zur Abhängigkeitsanalyse zu verwenden und führen Algorithmen und Datenstrukturen zu dessen effizienter Berechnung und Repräsentation ein. Damit verbunden erhalten wir *kompakte Abhängigkeitsgraphen*, welche als Repräsentation für beliebige Abhängigkeitsschemata geeignet sind. Als einen zentralen Beitrag dieser Arbeit *kombinieren wir Abhängigkeitsschemata mit Entscheidungsverfahren, welche auf dem suchbasierten QDPLL-Algorithmus beruhen.* Auf diese Weise kann QDPLL von der Unabhängigkeit von Variablen profitieren, welche durch das jeweilige Abhängigkeitsschema gegeben ist. Um die Kombination von Abhängigkeitsschemata und QDPLL experimentell zu evaluieren, haben wir diesen Ansatz in DepQBF implementiert. Die Ergebnisse unserer Experimente zeigen die potentiellen Vorteile auf, welche Entscheidungsverfahren für QBF aus Abhängigkeitsschemata ziehen können und motivieren gleichzeitig weiterführende Untersuchungen.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Acknowledgements

I am grateful for the support of my advisor Armin Biere. In 2008, he offered me a position as an assistant in his group, the Institute of Formal Models and Verification (FMV) at Johannes Kepler University in Linz. I highly appreciate Armin's informal and cooperative style of working together with colleagues and students in his group. Sharing his ideas and, in particular, his practical experience in SAT solving was a great benefit for my work on QBF. He also encouraged me to attend conferences and workshops. Armin always granted full independence to me regarding topics and working style which allowed me to develop my own ideas. Personally, I consider this attitude as most important for conducting research.

My colleague Martina Seidl helped me a lot during my time at the FMV and she was always open for valuable discussions. I gratefully took her comments on early versions of this work for improvements. I want to thank Uwe Egly for proof reading, giving me comprehensive feedback and for the opportunity of future collaboration. I had many discussions, mostly by e-mail, with Allen Van Gelder related to my research which gave me new insights.

I owe very much to the work of Marko Samer. Without his papers where he prepared the theoretical foundation of dependency schemes, I would not have been able to work out the practical aspects of dependency schemes and QBF solving presented in this work.

Many thanks to Reiner Hähnle and his colleagues for hosting me during my Short Term Scientific Mission (STSM) at Chalmers University of Technology in Gothenburg, Sweden, in September 2010.[1] In particular, I want to thank Richard Bubel for giving me tutorials on the KeY System.

I am indebted to my parents Irmgard and Wolfgang, my brother Michael and my dear friends for the unconditional support with the same strong commitment throughout the years. Thank you so much!

[1]Funded by COST Action IC0901: `http://richmodels.epfl.ch/`.

Contents

Chapter 1

Introduction

Propositional logic is a formalism which first has been of theoretical interest only since the 1970s. The *satisfiability problem of propositional logic (SAT)* was the first problem to be proved NP-complete [34], a result which entailed further research on related combinatorial problems [49] in the complexity class NP. Assuming that P $\neq$ NP, there is no polynomial-time algorithm to solve the decision problem of propositional logic.

It was not until the 1990s that impressive progress in SAT solving enabled practical applications. Despite exponential run time in the worst case, modern SAT solvers can efficiently tackle encodings of problems from real-world applications like bounded model checking [19], for example. Successful applications in practice motivate researchers to get involved in SAT, which in turn speeds up the overall progress in SAT research. As a result, modern and robust SAT solvers are widely applied in industry and academia.

Since SAT is NP-complete, it is natural to use SAT for encodings of problems from that complexity class. However, in practice SAT is also used to encode problems which are presumed to be in higher complexity classes. For example, the model checking problem for linear temporal logic (LTL) is PSPACE-complete [120], whereas its bounded variant is in NP [19]. To make bounded model checking (BMC) complete, the diameter of a system has to be computed, a problem which is in PSPACE [19, 92].

The question arises why not to tackle problems from classes like PSPACE using specific solvers in practice. Just as SAT is the prototypical problem for NP, the decision problem of *quantified Boolean formulae (QBF)* is PSPACE-complete [124]. QBF can be regarded as a generalization of SAT where propositional variables are existentially ($\exists$) or universally ($\forall$) quantified. Whereas explicit quantifiers do not add to expressiveness, problems like BMC can often be encoded more compactly in QBF than in SAT [14, 73].

The success of SAT solving originated from the classical DPLL algorithm [37].[1] DPLL is a backtracking algorithm which systematically enu-

[1]Modern SAT solvers differ substantially from basic DPLL due to several optimizations.

merates assignments to the variables in a given propositional formula. In the late 1990ies, DPLL was extended to QBF by handling universal quantifiers accordingly, which brought up the QDPLL algorithm [30].[2]

In contrast to SAT, neither QDPLL-based QBF solvers nor alternative approaches based on variable elimination have been found competitive for applications. In Section 2.6 of [14], references to a series of negative results are given, along with the following statement:

> *Unfortunately, the SAT arena has turned out to be quite unfavorable to QBF. All the experimental comparisons carried out recently yield (extremely) negative results (...)*

Solving a QBF seems to be inherently more difficult than solving a formula in propositional logic. It is only universal quantification which makes the difference between QBF and SAT at the syntactic level. A common syntactic structure of QBFs is *prenex conjunctive normal form (PCNF).* A PCNF consists of a propositional formula with a quantifier prefix. The quantifier prefix specifies whether variables are existentially or universally quantified and introduces an ordering on the variables. For example, the formula

$$\forall x \exists y.\ \phi \tag{1.1}$$

in general expresses something different than

$$\exists y \forall x.\ \phi \tag{1.2}$$

To see the difference, let us consider a high-level example from a first-year undergraduate course on mathematics which the author of this work attended at JKU Linz.

Variable Dependencies

Assume that we are given a set of locks and keys. Let ϕ in Formulae 1.1 and 1.2 represent the following proposition:

> Key y unlocks lock x.

Formula 1.1 amounts to the proposition:

> For all locks x there exists a key y such that y unlocks x,

whereas Formula 1.2 amounts to:

> There exists a key y for all locks x such that y unlocks x.

[2]The (Q)DPLL algorithm is also called (Q)DLL in some publications. We stick to (Q)DPLL in this work.

According to the first proposition, there is a key for every lock but the keys can be different. For example, key k_1 unlocks lock x_1, key k_2 unlocks lock x_2 but $k_1 \neq k_2$. That is, key k_1 does not unlock lock x_2. Hence the choice of the key might depend on the given lock.

According to the second proposition there exists *one particular* key k which unlocks *all* locks. We can think of k as the master key for all the locks. If the second proposition is true then so is the first one but not necessarily vice versa. The fact that for every lock there is a key does not necessarily imply that there is a master key for all locks.

There is strong indication in QBF research that the hardness of QBF observed in practice is due to dependencies between variables as pointed out in our example. If a QBF solver is given Formula 1.1 then in general it must not assign variable y before x because otherwise y can no longer take values with respect to the value of x. Hence the value of y might depend on the value of x. Neglecting dependencies can cause a QBF solver to return incorrect results. For example, if Formula 1.1 is satisfiable, it might erroneously be found unsatisfiable if y is assigned before x. In general, variables have to assigned in the ordering of the quantifier prefix during semantical evaluation of a QBF using QDPLL. Therefore the prefix ordering imposes restrictions on the set of assignments that QDPLL can enumerate.

Quantifier Prefixes vs. Dependency Schemes

In this work, we consider approaches to overcome the restrictions resulting from quantifier prefixes in PCNFs. The prefix ordering might be too strict in the sense that relaxations are possible without affecting the result determined by QBF solvers. Any relaxation of the prefix ordering grants additional freedom to a QBF solver. For example, in SAT solving where all variables are (implicitly) existentially quantified, a solver is free to assign variables in arbitrary order. By relaxing the prefix ordering of PCNFs, it is possible to bring QDPLL-based QBF solvers closer to the freedom of SAT solvers, as far as the selection of variables is concerned.

As a means for relaxing the prefix ordering in PCNFs, we consider the framework of *dependency schemes*. Dependency schemes were introduced by Samer and Szeider, first for QBF [112, 113] and and later for quantified constraint satisfaction problems (CSPs) [111]. A dependency scheme D is a binary relation over the set of variables of a given PCNF which expresses information on independence of variables. If $(x, y) \notin D$ then variable y does not depend on variable x. Otherwise, if $(x, y) \in D$ then we conservatively *regard* y to depend on x. If y does not depend on x then a QBF solver is free to assign x before y or vice versa. The relative order of these assignments does not cause the solver to produce incorrect result. Once D has been computed for a given PCNF ψ, independence represented by D can be used to relax the ordering of the quantifier prefix of ψ.

Informally, the quality of a dependency scheme D can be expressed in terms of the amount of independence that is represented by D for a given PCNF. If $(x, y) \in D$ then y either indeed depends on x or it is found independent with respect to some other, "better" dependency scheme D'.

It is possible to compute precise and optimal information on independence in a given PCNF using the framework of dependency schemes. That is, for every PCNF there exists an optimal dependency scheme. However, obtaining such optimal information is at least as hard as QBF solving. Hence in practice we have to trade optimality for efficiency of computation.

We focus on tractable dependency schemes which can be computed in polynomial time by analyzing the syntactic structure of a given PCNF. The cost of tractability comes at a reduced amount of independence that can be identified. It is well known that the dependency scheme given by the ordering of quantifier prefixes in PCNFs is not the best we can get. For example, the approach of *mini-scoping* (also called *anti-prenexing*), which is common in first-order theorem proving and QBF solving, allows to shift quantifiers in a PCNF from the prefix into its quantifier-free part. This way, the prefix ordering can be relaxed in terms of tree-like quantifier structure. That is, mini-scoping produces a quantifier tree which corresponds to a partial ordering of quantifiers. This is in contrast to the linear ordering given by the quantifier prefix.

We can think of the binary relation D given by a dependency scheme as a general form of non-linear quantifier structure. Dependency schemes represent a partial ordering on the variables, just as quantifier trees. However, it turns out that tree-like quantifier structure given by mini-scoping is not the best we can get by syntactic analysis. There is the tractable *standard dependency scheme* D^{std} [113] which improves upon mini-scoping in the sense that it identifies at least the same (and possibly more) amount of independence in a given PCNF. In contrast to mini-scoping, the standard dependency scheme can be computed deterministically.

We attempt to present a uniform view on dependency schemes for QBF which allows for applications in QBF solvers. Dependency schemes are relevant for QBF solvers based on QDPLL as well as on variable elimination. Our goal is to combine QDPLL with dependency schemes such that information on independence can be exploited during the evaluation of a PCNF. We describe how to integrate dependency schemes seamlessly into QDPLL. Actually, dependency schemes are intrinsic to QBF semantics. Our integrated view generalizes the classical QDPLL algorithm [30] from quantifier prefixes to arbitrary dependency schemes. This generalization directly enables applications of advanced dependency schemes such as the triangle [113] or quadrangle dependency scheme [51].

We evaluated the performance of QDPLL relying on quantifier prefixes, mini-scoping and the standard dependency scheme D^{std}. For experiments we implemented the solver DepQBF [84] based on QDPLL with conflict-directed clause learning (CDCL) and solution-directed cube learning (SDCL). DepQBF tightly integrates dependency schemes as compact dependency graphs. We present compact dependency graphs as a representation of *arbitrary* dependency schemes. Similarly, our solver DepQBF can be combined with arbitrary dependency schemes. Experimental results in our setting show that QDPLL performs best when combined with the standard dependency scheme D^{std} in terms of solved instances and average run time. Compared to quantifier trees obtained by mini-scoping and the standard dependency scheme, we observed larger numbers of backtracks for classical QDPLL with quantifier prefixes. Hence our experimental results illustrate the potential improvements that can be drawn from dependency schemes in QBF solving in general. These observations apply to approaches alternative to QDPLL like variable elimination as well.

Outline and Contributions

The structure and main contributions of this work are outlined as follows.

- In Chapter 2 we introduce basic terminology and concepts related to syntax and semantics of QBF. Different from most publications in QBF literature, our semantical definition relies on *assignment trees*, which we adopt from related work. In contrast to common, recursive definitions of QBF semantics, the concept of assignment trees allows to describe dependency schemes naturally.

- We introduce the theory of *dependency schemes* in Chapter 3. The theoretical framework is due to Samer and Szeider [111, 112, 113]. We attempt to present a uniform view which allows for applications in practical QBF solving. As an important result, we prove that quantifier trees obtained by mini-scoping never identify more independence than the standard dependency scheme. We favour the standard dependency for practical applications because it can be computed deterministically in contrast to quantifier trees.

- Chapter 4 considers efficient representations of dependency schemes. We introduce *directed acyclic dependency graphs (DAGs)* in order to represent the binary relation given by a dependency scheme D. We point out how to obtain *compressed dependency DAGs* by defining equivalence relations over the set of variables of a PCNF. Equivalences between variables are given with respect to dependency information in D. Compressed dependency DAGs can be used to represent arbitrary dependency schemes in general. Taking the standard dependency scheme D^{std} as a concrete example, we present an *algorithm to compute a compressed dependency DAG for D^{std}*. Related *experimental results* illustrate the efficiency of our approach in practice.

- In Chapter 5, we consider applications of dependency schemes in search-based QBF solving. We show how to *combine arbitrary dependency schemes and QDPLL*. Compressed dependency graphs introduced in Chapter 4 are the key to efficient combinations. Further, we present QDPLL with conflict-directed clause learning (CDCL) and solution-directed cube learning (SDCL) as implemented in our solver DepQBF. By analyzing the parts of QDPLL, we point out how to profit from dependency schemes in practice. Comprehensive empirical results confirm the potential benefits compared to classical QDPLL in practice.

Chapter 2

Preliminaries

We introduce syntax and semantics of QBF. Our focus is on specific syntactic forms and semantic concepts that are relevant in the context of dependency schemes. This yields definitions related to semantics which can be considered non-standard compared to variants prevalent in QBF literature. Below we clearly motivate deviations from the de facto standard by applications of dependency schemes as illustrated in Chapter 3.

2.1 Syntax

The syntax of some logic specifies how formulae are structured. Formulae which do not comply with syntactic rules are not part of the language of the logic. We first introduce syntactic definitions of QBF which are standard in literature. Then, for the purpose of dependency schemes, we focus on syntactically restricted formulae. We define the syntax of QBF on top of propositional logic and hence regard QBF as an extension thereof. For a general introduction to propositional logic we refer to [28], for example. The following syntax definitions in this section are common in QBF literature.

2.1.1 Propositional Logic

The basic building blocks of propositional logic are *propositional variables*, also called atoms, and *truth constants*. We write "variables" instead of "propositional variables". Such variables represent propositions which can be either true or false. Truth constants represent propositions which are always false ($\bot$) or always true ($\top$).

Definition 2.1.1. Given a set of variables V, a *formula of propositional logic* is built from variables in V and the propositional operators *negation* $\neg$, *disjunction* $\lor$ and *conjunction* $\land$ according to the following rules:

1. $\bot$ and $\top$ are propositional formulae.

2. If $x \in V$ then x is a propositional formula.

3. If ϕ is a propositional formula then $\neg(\phi)$ is also a propositional formula.

4. If ϕ and ϕ' are propositional formulae then $(\phi \otimes \phi')$, where $\otimes \in \{\wedge, \vee\}$, is also a propositional formula.

Given formula ϕ, $V(\phi)$ is the set of variables occurring in ϕ. For brevity, we write V if ϕ is clear from the context. Apart from negation, disjunction and conjunction as introduced above, there are further propositional operators such as *exclusive disjunction* $\oplus$, *implication* $\rightarrow$, or *biconditional* $\leftrightarrow$. It is well known that the latter operators can be expressed in terms of $\neg$ combined with either $\vee$ or $\wedge$ under a potential size increase of a formula.

The set of rules defined above allows to build propositional formulae with arbitrary structure. There is no restriction on how propositional operators are nested. For practical applications, it is often convenient to allow only formulae which have a particular uniform structure, called *normal form.*

Example 2.1.1. A propositional formula in *negation normal form (NNF)* is built from the same rules as in Definition 2.1.1, except that for rule 3 ϕ must be a truth constant or a propositional variable.

In the following, we introduce a popular normal form based on NNF which is widely used in the domain of automated reasoning. We also consider dependency schemes and QBF solving entirely in the context of that normal form in the forthcoming chapters.

2.1.2 Conjunctive Normal Form

The following definitions are standard in propositional logic. For a variable x, a *literal* is either x or its negation $\neg x$ where $v(x) = x$ and $v(\neg x) = x$ denotes the variable of the literal. A literal l is *positive* if $l = x$ and *negative* if $l = \neg x$. A *clause* is a disjunction $C_i := (l_1 \vee \ldots \vee l_{k_i})$ over literals. For clauses C_i, a propositional formula $\phi := C_1 \wedge \ldots \wedge C_n$ is in *conjunctive normal form (CNF).*

Definition 2.1.2. For a CNF ϕ and a literal l, $O(l) := \{C \mid C \in \phi, l \in C\}$ is the set of *literal occurrences* of l, that is the set of all clauses in ϕ which contain literal l. For a variable x, note that $O(x) \nsubseteq O(\neg x)$.

Definition 2.1.3. Given a variable x, the set of *variable occurrences* of a variable x is $O(x) \cup O(\neg x)$.

The *empty clause* and the *empty formula* are empty disjunctions and conjunctions, respectively, and are denoted by the empty set $\emptyset$.

In general, we assume that a clause neither contains multiple nor complementary literals of one and the same variable. A clause containing complementary literals is redundant and can be eliminated from the CNF. Further, we require that a clause does not contain truth constants $\top$ and $\bot$.

2.1.3 Quantified Boolean Formulae

It is appropriate to regard the logic of quantified Boolean formulae (QBF) as an extension of propositional logic. In QBF variables can explicitly be associated with universal ($\forall$) or existential ($\exists$) *quantifiers.* We rely on definitions from [26].

Definition 2.1.4. A *quantified Boolean formula* is built from propositional formulae and quantifiers according to the following rules:

1. Every propositional formula according to Definition 2.1.1 is a QBF.
2. If ϕ is a QBF then $\neg(\phi)$ is also a QBF.
3. If ϕ and ϕ' are QBFs then $(\phi \otimes \phi')$, where $\otimes \in \{\wedge, \vee\}$, is also a QBF.
4. If ϕ is a QBF, $x \in V(\phi)$ and expression Qx does not occur in ϕ then $Q'x.\ (\phi)$, where $Q, Q' \in \{\forall, \exists\}$, is also a QBF.

Like above for CNFs, $V(\phi)$ the set of variables occurring in a QBF ϕ. Propositional operators and quantifiers can be arbitrarily nested in QBFs according to the building rules in Definition 2.1.4. We abbreviate sequences $Qx_1Qx_2 \dots Qx_n.\ (\phi)$ of equally quantified variables by $Qx_1, x_2, \dots, x_n.\ (\phi)$, where $Q \in \{\forall, \exists\}$. It is common to visualize the syntactic structure of QBFs by parse trees.

Definition 2.1.5 (taken from [28])**.** A *parse tree* $T(\phi)$ of a QBF ϕ is defined recursively based on the syntactic structure of ϕ:

1. If ϕ is a truth constant or a variable then $T(\phi)$ consists of only one node representing ϕ.
2. If $\phi = \neg\phi'$ or $\phi = \phi' \otimes \phi''$ where $\otimes \in \{\wedge, \vee\}$, or $\phi = Qx.\ (\phi')$ where $Q \in \{\forall, \exists\}$, then parse trees $T(\phi)$ look as shown below.

$$\begin{tikzcd}
\neg \arrow[d, no head] & & \otimes \arrow[dl, no head] \arrow[dr, no head] & & Qx \arrow[d, no head] \\
\phi' & \phi' & & \phi'' & \phi'
\end{tikzcd}$$

Given a QBF of the form $Qx.\ (\phi)$ where ϕ is a QBF, $x \in V(\phi)$ and $Q \in \{\forall, \exists\}$. The *scope* of the quantified variable Qx is the QBF ϕ. The following definitions were adopted from [26]. For QBF $Qx.\ \phi$, where $Q \in \{\forall, \exists\}$, the occurrence of x in expression Qx is a *quantified occurrence*. Any other occurrence of x is non-quantified. An occurrence of a variable x is *bound* if the occurrence is in the scope of Qx. All the non-quantified occurrences of a variable x which are not in the scope of Qx are *free occurrences*. A variable is *free* in a QBF ϕ if there is a free occurrence of x in ϕ. Otherwise, x is *bound* in ϕ. A QBF is *closed* if it does not contain free variables. In general, we consider only closed QBFs. We omit parentheses in $Qx.\ (\phi)$ and write $Qx.\ \phi$, where $Q \in \{\forall, \exists\}$, if the scope of Qx is clear from the context.

2.1.4 Prenex Conjunctive Normal Form

A quantified Boolean formula (QBF) $\psi := Q_1B_1 \dots Q_nB_n.\, \phi$ in *prenex conjunctive normal form (PCNF)* consists of a propositional formula ϕ in CNF over a set of variables V and a *quantifier prefix* $Q_1B_1 \dots Q_nB_n$, where $Q_i \in \{\forall, \exists\}$. The quantifier prefix is a linearly ordered set of *quantifier blocks* B_i, where $B_1 < \dots < B_n$, which forms a *partition* on the set of variables: $V = \bigcup B_i$ where $B_i \neq \emptyset$ and $B_i \cap B_j = \emptyset$ for $1 \leq i, j \leq n$ and $i \neq j$. For PCNF ψ, $V(\psi)$ is the set of variables occurring in ψ.

A quantifier block B_i is *existential* ($Q_i = \exists$) if it is associated with an existential quantifier and *universal* ($Q_i = \forall$) otherwise. The set of existential and universal variables is denoted by $V_\exists = \bigcup\{B_i \mid Q_i = \exists\}$ and $V_\forall = \bigcup\{B_i \mid Q_i = \forall\}$, respectively. For a literal l with $v(l) \in B_i$, $b(l) = B_i$ is the quantifier block of (the variable of) l and $q(l) := q(v(l)) := Q_i$ is the *quantifier type* of (the variable of) l. Two adjacent quantifier blocks B_i and B_{i+1} in the prefix are always differently quantified, that is $Q_i \neq Q_{i+1}$ for $1 \leq i < n$. Given a QBF with n quantifier blocks, there are $n-1$ *quantifier alternations*. For a quantifier block B_i and literal l, $\delta(B_i) = i$ and $\delta(l) = \delta(b(v(l)))$ denote the *level* of B_i and of l, respectively. Blocks B_1 and B_n are the *outermost* and *innermost* quantifier blocks.

For quantifier blocks B_i and B_j, B_j is *larger* than B_i, written as $B_i < B_j$, if $i < j$. The linear ordering of quantifier blocks is extended to literals as follows. For each quantifier block B_i, let $\lessdot_i$ be an arbitrary but fixed linear ordering on the variables of B_i. Given literals l and l', l' is *larger* than l, written as $l < l'$ if and only if either $\delta(l) < \delta(l')$, that is l and l' are from different quantifier blocks, or $l \lessdot_i l'$, that is $v(l)$ and $v(l')$ are from the same block B_i and $\lessdot_i$ is the corresponding linear ordering on variables in B_i.

For convenience, we only consider PCNFs where, for all clauses $C_i := (l_1 \vee \dots \vee l_{k_i})$, $l_j < l_{j'}$ for $1 \leq j < j' \leq k_i$ and $q(v(l_{k_i})) = \exists$. That is, all literals are sorted ascendingly and the largest literal is existential. Literals l_{k_i} where $q(v(l_{k_i})) = \forall$ can always be eliminated by *universal reduction* as described in Section 5.3.1 on page 96.

Additionally, we assume that, for all variables $x \in V$, at least $O(x) \neq \emptyset$ or $O(\neg x) \neq \emptyset$. That is, there occurs at least one literal for each quantified variable in the formula. Note that $V = \bigcup B_i$ as defined above. Further, if there is a literal l in some clause C_i with $v(l) = x$, then by definition of literals, also $x \in V$. Thus all variables which occur in the formula are quantified and hence all formulae are closed.

QDIMACS Format

Our syntax definition of PCNF from Section 2.1.4 is close to the QDIMACS format [106] which is a standardized format of QBFs in PCNF. The format does not prohibit free variables but treats them in a special way. Let a PCNF $Q_1B_1, \ldots, Q_nB_n.\ \phi$ be given. If the outermost quantifier block B_1 is existential then any free variable x is quantified in B_1. Otherwise, if B_1 is universal then an additional existential quantifier block B_0 is added where all free variables are quantified, thus obtaining $\exists B_0 Q_1B_1, \ldots, Q_nB_n.\ \phi$. Given a CNF ϕ where all variables are free, this way a PCNF can be obtained for ϕ where all variables are existentially quantified.

2.2 Semantics

Now that we have defined the syntactic structure of QBFs and PCNFs in particular, we address semantical evaluation. Semantics provide a set of rules to assign meaning to formulae. We are concerned with propositional logic and a generalization thereof. Therefore, a semantical evaluation amounts to assign truth values *true* ($\top$) or *false* ($\bot$) to variables. For simplicity, we use the symbols $\top$ and $\bot$ both for the syntactic truth constants as in Definition 2.1.1 as well as for truth values in semantics. The truth value of a formula can be determined based on its syntactic structure.

In the following, we introduce the semantics of PCNFs based on tree-like representations of truth assignments. Although not being new [111, 114], this particular semantic definition deviates from recursive semantics which is established in QBF literature. Our approach has advantages when it comes to dependency schemes, which is pointed out in Section 2.2.2 below.

2.2.1 Assignments and Assignment Trees

We adopt definitions from [111, 114].

Definition 2.2.1. Given PCNF ψ, an *assignment* A is a function $A : V \to \{\top, \bot\}$ which maps variables V in ψ to truth values true ($\top$) and false ($\bot$). An assignment A is *complete* if function A is total and *partial* otherwise.

We represent an assignment A as a set of literals $\{l_1, \ldots, l_n\}$ such that, for a variable $x \in V$, $l_i = x$ if $A(x) = \top$ and $l_i = \neg x$ if $A(x) = \bot$. Hence literals in A represent truth assignments to variables.

Given an assignment A and PCNF ψ, $\psi[A]$ is the *formula under assignment A*. Let $A := \{l\}$ be an assignment with $\delta(v(l)) = i$ and $\psi :=$

$Q_1B_1 \dots Q_i(B_i \cup \{v(l)\}) \dots Q_nB_n.\, \phi$. The formula $\psi[\{l\}]$ is obtained from ψ by substituting the occurrences of $v(l)$ by truth constants and by deleting $v(l)$ from the prefix. That is $\psi[\{l\}] := Q_1B_1 \dots Q_iB_i \dots Q_nB_n.\, (\phi[\{l\}])$ where $\phi[\{l\}]$ is obtained from ϕ as follows. Given the assignment $\{l\}$, for each variable occurrence l' of $v(l)$ by Definition 2.1.3, where $l' = v(l)$ or $l' = \neg v(l)$, $v(l)$ in l' is replaced by $\top$ if $l = v(l)$ and by $\bot$ if $l = \neg v(l)$.

Further, $\phi[\{l\}]$ is *simplified* by applying the following well-known equivalences of Boolean algebra as rewrite rules until saturation:

$$\begin{array}{ccc} \neg\top \rightsquigarrow \bot & \neg\bot \rightsquigarrow \top & \top \wedge \phi \rightsquigarrow \phi \\ \\ \bot \vee \phi \rightsquigarrow \phi & \top \vee \phi \rightsquigarrow \top & \bot \wedge \phi \rightsquigarrow \bot \end{array}$$

Additionally, quantifiers of variables which do no longer occur in $\psi[A]$ are removed from the prefix of $\psi[A]$. Note that notation $\psi[A]$ is applicable to CNFs as well. The result of simplifying a formula under a complete assignment is always either $\top$ or $\bot$. We define $\psi[\emptyset] := \psi$ for the *empty assignment* $\emptyset$ and $\psi[\{l_1, l_2, \dots, l_n\}] := (\psi[l_1])[\{l_2, \dots, l_n\}]$ for *compound assignments*. For simplicity, we omit parentheses in $\psi[\{l_1, \dots, l_n\}]$ and write $\psi[l_1, \dots, l_n]$.

Example 2.2.1. Given the PCNF $\psi := \forall x \exists y.\, (x \vee \neg y) \wedge (\neg x \vee y)$. We have $\psi[\emptyset] = \psi$, $\psi[y] = \forall x.\, (x)$, $\psi[\neg x] = \exists y.\, (\neg y)$, $\psi[x, y] = \top$ and $\psi[x, \neg y] = \bot$.

Definition 2.2.2. A (complete or partial) assignment m is a *CNF-model* of a CNF ϕ, written as $m \models \phi$, if $\phi[m] = \top$.

Definition 2.2.3. Given a PCNF $\psi := Q_1B_1 \dots Q_nB_n.\, \phi$. An *assignment tree* T is a tree of complete assignments according to the following restrictions. Every *node* N in T except the *root* r represents a truth assignment to a variable v in V. Node N assigns literal v ($\neg v$) if variable v is assigned to $\top$ ($\bot$). A node has exactly one *sibling* if and only if it assigns a truth value to a universal variable. Nodes which assign existential variables do not have siblings. Two siblings altogether denote assignments $\top$ and $\bot$ to universal variables. In this case the left (right) sibling assigns $\bot$ ($\top$) to the respective variable. Every *path* P from the root to a leaf of T corresponds to a complete assignment A for variables in ψ. A node N for variable v is an *ancestor* of another node N' for variable v' in P if and only if $v < v'$. That is, assignments along every path P respect the variable ordering as defined in Section 2.1.4.

Example 2.2.2. Figure 2.1 shows three assignment trees for the PCNF $\psi := \forall x \exists y.\, (x \vee \neg y) \wedge (\neg x \vee y)$. Note that the assignments along the paths are ordered with respect to the quantifier prefix. Further, the left sibling of universal nodes assigns always $\bot$ to x.

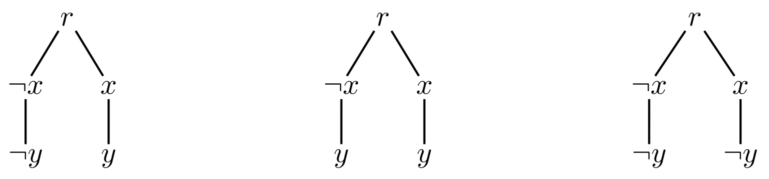


Figure 2.1: Three assignment trees for the PCNF $\psi := \forall x \exists y.\ (x \vee \neg y) \wedge (\neg x \vee y)$ from Examples 2.2.2 and 2.2.3. The leftmost tree is a PCNF-model of ψ whereas the other two are not.

Definition 2.2.4. An assignment tree m is a *PCNF-model* of the PCNF $\psi := Q_1 B_1 \dots Q_n B_n.\, \phi$, written as $m \models \psi$, if every path P in m is a CNF-model of ϕ. A PCNF-model is also called a *satisfying assignment tree.*

A CNF is *satisfiable* if it has a CNF-model. Two CNFs ϕ and ϕ' are *model-equivalent*, written as $\phi \equiv_m \phi'$, if and only if for all assignments m, $m \models \phi$ if and only if $m \models \phi'$. Two CNFs ϕ and ϕ' are *satisfiability-equivalent*, written as $\phi \equiv_s \phi'$, if and only if ϕ is satisfiable then ϕ' is satisfiable and vice versa. Satisfiability, model-equivalence and satisfiability-equivalence with respect to PCNFs are defined accordingly with respect to PCNF-models.

Example 2.2.3. Given the PCNF $\psi := \forall x \exists y.\ (x \vee \neg y) \wedge (\neg x \vee y)$ from Example 2.2.1 and the three assignment trees shown in Figure 2.1. The leftmost assignment tree is a PCNF-model of ψ because the assignments $\{\neg x, \neg y\}$ and $\{x, y\}$ along the two paths from the root r to the leaves are CNF-models of the CNF-part of ψ: $\psi[\neg x, \neg y] = \top$ and $\psi[x, y] = \top$. This is not the case for the other two assignment trees on the right since the assignments $\{\neg x, y\}$ and $\{x, \neg y\}$, respectively, are no CNF-models: $\psi[\neg x, y] = \bot$ and $\psi[x, \neg y] = \bot$.

An alternative definition of models for QBFs in PCNF was used in [26]. Given a PCNF ψ, each existential variable x_i is replaced by a Boolean function $f_{x_i}(y_1, \dots, y_{j_i})$ which depends on exactly those universal variables y_k where $y_k < x_i$ for $1 \leq k \leq j_i$. The possibility that x_i takes different values in different paths in an assignment tree of ψ is reflected by f_{x_i} which depends on universal variables smaller than x_i. Replacing existential variables by functions corresponds to *skolemization* [121].

2.2.2 Recursive Semantical Evaluation

Our semantical definition based on assignment trees and tree-like models deviates from what can be considered "standard semantics" of QBFs. For the vast majority of QBF-related publications, semantics are defined recursively with respect to the syntactic structure of a formula. Semantically evaluating a formula breaks down to evaluating subformulae in its parse tree and combining the results in order to finally obtain the truth value of the formula. We call this kind of semantics “standard” because it prevails in QBF literature.

In the following we introduce the well-known alternative semantical definition of QBF based on *recursive evaluation.* The purpose is to show that our original definition relying on assignment trees and tree-like models is more suitable to act as a framework for the theory of dependency schemes. We argue that in the context of standard recursive semantics like [26, 28], it is not possible in a straightforward way to distinguish multiple QBF models. We are concerned with situations where existential variables are assigned differently with respect to the values of universal variables.

In order to apply the following definition to PCNFs, we temporarily assume that quantifier blocks contain exactly one variable and that adjacent blocks are not necessarily differently quantified. For example $\forall x{,}y \exists a{,}b.\ \phi$ is treated as $\forall x \forall y \exists a \exists b.\ \phi$.

Definition 2.2.5. Given a *closed* QBF ψ, satisfiability is determined recursively based on the syntactic structure of ψ as follows:

1. If $\psi := \bot$ then ψ is unsatisfiable.
2. If $\psi := \top$ then ψ is satisfiable.
3. If $\psi := \phi \vee \phi'$ then ψ is satisfiable if and only if ϕ or ϕ' is satisfiable. Otherwise ψ is unsatisfiable.
4. If $\psi := \phi \wedge \phi'$ then ψ is satisfiable if and only if both ϕ and ϕ' are satisfiable. Otherwise ψ is unsatisfiable.
5. If $\psi := \exists x.\ \phi$ then ψ is satisfiable if and only if $\phi[x]$ or $\phi[\neg x]$ is satisfiable. Otherwise ψ is unsatisfiable.
6. If $\psi := \forall x.\ \phi$ then ψ is satisfiable if and only if both $\phi[x]$ and $\phi[\neg x]$ are satisfiable. Otherwise ψ is unsatisfiable.

Note that in Definition 2.2.5 there is no case for $\psi := x$ where x is a variable. Since we consider only closed QBFs ψ, this case cannot occur. All variables are assigned in the subcases of $\psi := \exists x.\ \phi$ and $\psi := \forall x.\ \phi$, which amounts to evaluating $\psi := \bot$ and $\psi := \top$ in the end. Further, semantics based on assignment trees and recursive evaluation from Definitions 2.2.3 and 2.2.5, respectively, are compatible. That is, a PCNF has a PCNF-model if and only if it is satisfiable by Definition 2.2.5. The ordering of assignments along paths in assignment trees corresponds to recursive applications by rules 5 and 6 in Definition 2.2.5.

Example 2.2.4. Given the PCNF $\psi := \forall x \exists y.\ (x \vee \neg y) \wedge (\neg x \vee y)$ from Example 2.2.1. Due to $\forall x$ we have to check if both $\psi[x] = \exists y.\ (\phi[x])$ and $\psi[\neg x] = \exists y.\ (\phi[\neg x])$ are satisfiable. For $\psi[x] = \exists y.\ (\phi[x])$, $\phi[x, y]$ or $\phi[x, \neg y]$

must be satisfiable. Similarly for $\psi[\neg x] = \exists y.\ (\phi[\neg x])$, $\phi[\neg x, y]$ or $\phi[\neg x, \neg y]$ must be satisfiable. As pointed out in Example 2.2.3, we have $\psi[x, y] = \top$ and $\psi[\neg x, \neg y] = \top$.

For the PCNF ψ from Example 2.2.4, we find out that $\phi[x, y]$ and $\phi[\neg x, \neg y]$ are satisfiable but neither $\phi[x, \neg y]$ nor $\phi[\neg x, y]$ are. This is sufficient to justify satisfiability of ψ. In fact we could come to that conclusion without evaluating all four base cases but just the two satisfiable ones. Assume we are given $\psi[x]$. Once we know that $\phi[x, y]$ is satisfiable, there is no need to consider $\phi[x, \neg y]$, and similarly for $\psi[\neg x]$. This is due to the rule for evaluating $\exists y$ in $\psi[x]$ and $\psi[\neg x]$. There is no predefined ordering for considering subcases, hence the choice is arbitrary and non-deterministic.

The construction of dependency schemes, which is part of Chapter 3, requires to analyze dependencies between variables in a PCNF. From a theoretical point of view, dependency analysis in a PCNF ψ reduces to checking whether a variable takes different values in different subtrees of a satisfying assignment tree ψ. Referring to Example 2.2.4, we need to know if changing the value of x forces the value of y to change as well. That is, are both $\phi[x, y]$ and $\phi[\neg x, y]$ or both $\phi[x, \neg y]$ and $\phi[\neg x, \neg y]$ satisfiable? If one of these is true then x and y are independent because changing the value of x does *not* force a change of the value of y to satisfy the formula. However, regarding PCNF ψ from Example 2.2.4, x and y are *not* independent because none of the two possible assignment trees (see the two trees on the right in Figure 2.1) where the value of y is the same in the two subtrees is a PCNF-model of ψ. Hence changing the value of x *forces* a change of the value of y to satisfy ψ.

From that perspective, recursive semantics as defined above seems to be too coarse. The relation of values between different variables is not reflected explicitly as in assignment trees. This is the reason why we prefer assignment trees and tree-like models in the context of dependency schemes.

2.2.3 Complexity

SAT, the satisfiability problem of propositional logic, is the classical *NP-complete* problem [34]. Unrestricted occurrences of different quantifiers in QBFs render the corresponding decision problem *PSPACE-complete* [49, 124]. The number of quantifier alternations of a PCNF is related to the *polynomial-time hierarchy* [90, 123] as follows.

Definition 2.2.6. (copied from Definition 23.3.1 in [26]) The *prefix type* of a QBF[1] is defined as follows. Every propositional formula by Definition 2.1.1 has prefix type $\Sigma_0 = \Pi_0$. Given a PCNF ψ with prefix type Σ_n (Π_n), the PCNF $\forall x_1, \ldots, x_m.\ \psi$ ($\exists x_1, \ldots, x_m.\ \psi$) has prefix type Π_{n+1} (Σ_{n+1}).

[1]We consider only PCNFs, but Definition 2.2.6 can be adapted to non-CNF formulae with quantifier prefixes as well.

Definition 2.2.7. (copied from Section 23.3 in [26]) The *polynomial-time hierarchy* is defined as follows for $k \geq 0$:

$$\Delta_0^P := \Sigma_0^P := \Pi_0^P := P,$$

$$\Sigma_{k+1}^P := NP^{\Sigma_k^P},\ \Pi_{k+1}^P := co\Sigma_{k+1}^P,\ \Delta_{k+1}^P := P^{\Sigma_k^P}$$

where Δ_{k+1}^P (Σ_{k+1}^P) is the class of all problems which can be decided deterministically (non-deterministically) in polynomial-time with the help of an *oracle* for a problem in Σ_k^P. An oracle for a problem in Σ_k^P is a subroutine which solves a problem in Σ_k^P in constant time. The class Π_{k+1}^P contains every problem whose complement is in Σ_{k+1}^P. We have $\Sigma_1^P = NP$, $\Pi_1^P = coNP$, $\Delta_1^P = P$ and, for $k \geq 1$, $\Delta_k^P \subseteq (\Sigma_k^P \cap \Pi_k^P)$.

Theorem 2.2.1 (copied from Theorem 23.3.2 in [26], respectively [123, 129])**.** *For $k \geq 1$, the satisfiability problem for QBFs with prefix type Σ_k (Π_k) is Σ_k^P-complete (Π_k^P-complete).*

It is unknown how the complexity classes P, NP and PSPACE are exactly related to each other. So far no polynomial time algorithms are known for SAT or QBF. There are subclasses of QBF based on syntactic restrictions which can be decided in polynomial time, for example a specific form of quantified Horn formulae or QBFs with at most two literals per clause [2]. We refer to [24, 26, 28] for further details.

2.3 Decision Procedures: An Overview

We briefly give an overview of the two main approaches for checking satisfiability of a QBF, that is variable elimination and backtracking search. There is an abundance of literature on SAT and QBF solving, and a comprehensive treatment is out of scope of this work. In Chapter 5 we deal with the practice of backtracking search in detail. This section provides only an overview sufficient to understand the motivation of dependency schemes in the context of QBF solving in Chapter 3.

2.3.1 Backtracking Search

Semantics by Definitions 2.2.4 and 2.2.5 naturally form a framework for search-based decision procedures for QBF. Definition 2.2.5 can be turned into a recursive algorithm in a straightforward way. Rules for evaluating $\psi := \forall x.\ \phi$ and $\psi := \exists x.\ \phi$ correspond to case splits into subgoals $\phi[x]$ and $\phi[\neg x]$, respectively. Splitting the proof into subgoals is also called *branching* or *decision making*. Depending on the result of checking subgoals, the algorithm backtracks to the most recent unsolved subgoal and continues. In the following, we focus on solving PCNFs.

Definition 2.2.4 gives rise to a simple yet infeasible algorithm. Given PCNF ψ, we can try to construct a PCNF-model m by generating assignments A iteratively. If A satisfies the CNF-part of a PCNF then A can be turned into a path in m. Paths have to be added for siblings of universal nodes in m. If ψ is satisfiable then finally a complete PCNF-model m is obtained. Otherwise, there is at least one universal node for which no satisfying path starting at its sibling can be found, and so ψ is unsatisfiable. In the worst case the explicit representation of assignment trees requires space which is exponential in the number of variables of the QBF.

Given a QBF, the previously described algorithm searches for assignments which satisfy the CNF-part and checks whether these CNF-models can be combined to form a PCNF-model. The problem of exponential space is avoided in algorithms relying on the classical *DPLL* approach [37]. Originating from [38], DPLL[2] generates assignments recursively like in Definition 2.2.5. Assignments are represented implicitly by the structure of recursive applications of the semantical rules. Hence classical DPLL requires only space which is linear in the number of variables.

A first description of a DPLL-based algorithm for QBF was given in [30, 31]. We refer to this QBF-specific variant as *QDPLL*[3] and present a detailed description in Chapter 5. Modern implementations of SAT and QBF solvers highly differ from the original formulation of DPLL. The algorithm is typically not implemented recursively but iteratively. Splitting the proof into subcases is deferred as much as possible by applying rules like *boolean constraint propagation (BCP)*. We consider a QBF-specific variant of BCP in Chapter 5. Parallel variants of QDPLL like [48, 79, 80], for example, might benefit from modern multicore architectures and distributed computing environments.

Clause Learning

The method of *clause learning*, also called *conflict-directed clause learning (CDCL)* [117, 118], aims at guiding the search process out of regions of the search space which do not contain solutions. The theoretical foundation of clause learning for SAT and QBF is *resolution* and *Q-resolution*, respectively [27, 108], which we consider in Chapter 5. Using (Q-)resolution, clauses derived from the original formula and can be added thereto. Such learnt clauses are logically implied by the formula and hence are also satisfied by all CNF-models. Learnt clauses rule out certain assignments which are not satisfying anyway. Thus the solver does not have to consider such assignments. Learning methods for QBF were developed independently in [58, 78, 133, 134] under the terms *lemma and model caching*, or *clause and*

[2]Although Hilary Putnam is not co-author of [37], it is common to refer to the algorithm as *DPLL*, *DLL* or *CDCL* [118], if clause learning is applied (see next section).

[3]We also find other names such as *QSAT* and *Q-DLL* in literature.

cube learning. A cube is a conjunction of literals. Different from clauses, cubes are used to rule out assignments which satisfy the CNF-part of a CNF. We introduce clause and cube learning for QBF in Chapter 5.

Adding learnt clauses increases both the space and time requirements of a solver. The latter is due to rules like BCP which involve inspection of learnt clauses as well. Clause learning might increase the space complexity of DPLL from linear to exponential, because in the worst case exponentially many learnt clauses can be derived. To prevent this behaviour, learnt clauses are periodically discarded according to some heuristic strategy. The goal is to keep those learnt clauses where the solver effectively benefits from. Related approaches from SAT solving like [3, 5, 42] could also be adapted to QBF. We describe a trivial variant in Section 5.6.2 on page 121.

Branching Heuristics

The ordering in which variables are assigned might substantially influence the overall performance of a QBF solver (see also Example 3.3.6 on page 34). *Branching heuristics* determine the selection and ordering of variables which are used for case splits in the proof. Dynamic variants allow the solver to adapt the search with respect to those parts of the search space that have been visited recently. As with clause learning, work from the SAT domain related to branching heuristics [40, 42, 64, 93, 116, 118] might be applied to QBF as well but, to the best of our knowledge, there is no comprehensive empirical study in the context of QBF.

Non-Normal Form Solving

In the description of QDPLL, we focused on QBFs in PCNF. Definition 2.2.5 provides rules to evaluate arbitrary QBFs, and a corresponding recursive algorithm can be obtained in a straightforward way. Although original DPLL operates on CNF only, it is common to use the term QDPLL for generalizations to arbitrary QBFs as well. This approach has been implemented along with extensions like learning in [45, 46, 66, 67, 68], for example.

2.3.2 Variable Elimination

Backtracking search as described above in Section 2.3.1 is closely related to semantical definitions. In either of Definitions 2.2.4 and 2.2.5 variables are assigned in the ordering given by the quantifier prefix or parse tree. For assignment trees we explicitly require that assignments along paths fulfill this requirement. In recursive semantics, such ordering is implicitly given by nested evaluation. Due to this property, we call backtracking search a *top-down* approach, since variables are assigned from left (top) to right (bottom) in the prefix or parse tree, respectively. Example 3.1.1 on page 24 shows that this condition cannot be relaxed in general.

Variable elimination, the second major approach to solve QBFs, does not fit into that top-down framework. Differently from backtracking search, variable elimination does not explicitly try to generate a PCNF-model. Instead, the goal is to successively get rid of quantified variables until finally the formula reduces to $\top$ or $\bot$. Our focus is on PCNF but the approaches can be generalized to arbitrary QBFs. We briefly outline common approaches below and list related work at the end of this chapter with respect to data structures used in practice. Eliminating a variable typically increases the size of the formula. Practical applicability of variable elimination is determined by the amount of that size increase. In the worst case, the size of the formula doubles each time and hence induces an exponential growth over a sequence of elimination steps.

Shannon Expansion

In the following, we introduce *Shannon expansion* [38, 41, 115] for variable elimination. We first consider QBFs in PCNF where all variables are existentially quantified and then address expansion for general QBFs below.

Lemma 2.3.1. *The PCNF*

$$\psi := \exists x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n.\ (\phi)$$

is satisfiability-equivalent to

$$\exists x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n.\ (\phi[x_i] \vee \phi[\neg x_i]).$$

Note that $\phi[x_i]$ and $\phi[\neg x_i]$ is the formula obtained from ϕ by assigning x_i as defined in Section 2.2.1. Variable x_i is permanently assigned to true in one, and to false in another copy of CNF ϕ, respectively. The effects of assigning x to either truth value are simultaneously and directly encoded into the two copies of ϕ. This is different from backtracking search where variable assignments are made tentatively and retracted if no solution was found. Expansion by Lemma 2.3.1 eliminates one variable at the cost of doubling the size of the formula. In practice the variable to be expanded often occurs only in a small part of ϕ, which is different from the general pattern. In this case, expansion is much cheaper in terms of size increase because only the relevant parts of ϕ have to be copied. The approach of *mini-scoping*, which we consider in Section 3.2.2, can be used to find out parts which are relevant for expansion.

Elimination Ordering

We consider expansion for arbitrary PCNFs. The presence of both universal and existential variables in PCNFs complicates variable elimination in general. This is in contrast to the following straightforward adaption of Lemma 2.3.1 to arbitrary PCNFs.

Lemma 2.3.2. *The PCNF*

$$\psi := Q_1B_1, \dots, Q_n(B_n \cup \{x\}).\ \phi$$

is satisfiability-equivalent to

$$Q_1B_1, \dots, Q_nB_n.\ (\phi[x] \otimes \phi[\neg x])$$

where $\otimes := \wedge$ *if* $Q_n = \forall$ *and* $\otimes := \vee$ *if* $Q_n = \exists$.

The two copies of ϕ are conjoined by conjunction or disjunction with respect to the quantifier type of x. Note that the expanded variable x in Lemma 2.3.2 is from the *innermost* quantifier block. Variables must be eliminated from *right to left* in general, which differs from backtracking search. Therefore, we call methods of variable elimination *bottom-up* approaches. In Chapter 3, we show by Examples 3.1.1 and 3.1.2 that respecting the ordering is crucial both for backtracking search and variable elimination.

Universal Expansion

Different from Lemma 2.3.2, universal variables from the innermost quantifier block of a PCNF do not have to be expanded but can be eliminated right away by universal reduction (see also Section 5.3.1 on page 96) without incurring any size increase. This approach works only on PCNF and it is not clear, to the best of our knowledge, how to extend it to arbitrarily structured QBFs.

It is possible to extend Lemma 2.3.2 to expansion of universal variables from arbitrary quantifier blocks. This was done in [25] which built upon ideas from [16]. The potential drawback of this approach is a larger size increase. We introduce expansion of universal variables from the first non-innermost quantifier block B_{n-1} and refer to Lemma 3.4.4 on page 53.

Lemma 2.3.3. *The PCNF*

$$\psi := Q_1B_1, \dots, \forall(B_{n-1} \cup \{x\})\exists B_n.\ \phi$$

is satisfiability-equivalent to

$$\psi := Q_1B_1, \dots, \forall B_{n-1}\exists(B_n \cup B'_n).\ (\phi[x] \wedge \phi'[\neg x]).$$

Set B'_n *consists of fresh variables obtained from* duplicating B_n*, and in* ϕ' *occurrences of variables in* B_n *are replaced by duplicated ones in* B'_n*.*

In all preceding definitions of expansion we copied the entire formula ϕ which is conservative but usually pessimistic. In practice, it suffices to copy only those parts of ϕ where the expanded variable occurs. Additionally when applying Lemma 2.3.3, parts with occurrences of duplicated variables must be copied. Therefore, the number of duplicated variables in the set B'_n usually has an impact on the size of that part of ϕ that must be copied.

Example 2.3.1. Given the satisfiable PCNF $\psi := \forall x \exists y.\ (x \vee \neg y) \wedge (\neg x \vee y)$. By Lemma 2.3.3 formula ψ is satisfiability-equivalent to

$$\forall x \exists y,y'.\ ((x \vee \neg y) \wedge (\neg x \vee y))[x] \wedge ((x \vee \neg y') \wedge (\neg x \vee y'))[\neg x],$$

which further reduces to the satisfiable formula $\exists y,y'.\ (y) \wedge (\neg y')$.

We point out by Example 3.1.2 on page 24 that the requirement of duplicating variables cannot be relaxed *in general*. However, as we point out in Chapter 3, dependency schemes might be able to reduce the number of duplicated variables in B'_n in practice and hence limit the cost of variable elimination (see also Example 3.3.7 on page 36).

Data Structures and Formula Representation

The rules for expansion of variables work for arbitrary QBFs although in definitions above we considered only PCNFs. This gives rise to non-normal form solvers relying on variable elimination and either prenex non-CNF or non-prenex non-CNF formulae. For the latter, in general the elimination ordering has to follow the sequence of quantified occurrences $\forall x$ and $\exists x$ in the parse tree in bottom-up fashion. In contrast to PCNF, data structures like *and-inverter graphs (AIGs)* often allow for a more compact representation of a QBF [101, 102, 107]. *Negation normal form (NNF)* is close to the original parse tree of a QBF and was used in [6, 81].

The classical *Davis-Putnam (DP) algorithm* [38] can be used to eliminate existential variables in PCNFs by resolution. The resolution operation was originally introduced for first-order logic [108]. Variable elimination by resolution typically generates many redundant clauses. Sophisticated approaches like subsumption removal [16, 131] or methods of preprocessing [21, 41, 55] can be applied to reduce the size of the formula.

Binary decision diagrams (BDDs) [23] were used for QBF solving in several ways. Search-based approaches were coupled with representations of CNF-models by BDDs [4]. The implementation of standard Boolean operators enables the use of BDDs for variable elimination by expansion [75, 97, 99]. BDDs can also be used to encode sets [91] of clauses in combination with the Davis-Putnam algorithm [32, 33, 99].

Skolemization [121] is relevant for theoretical concepts of QBF models [26], but was also applied in practice to eliminate existential variables from QBFs [13, 74]. The QBF solver presented in [13] integrates skolemization with BDDs and SAT solving techniques and can therefore be considered a hybrid approach.

Chapter 3

Dependency Schemes

3.1 Introduction

In Section 2.3, we briefly introduced the two approaches of variable elimination and backtracking search for SAT and QBF solving. Backtracking search by means of DPLL [37] and QDPLL [30, 31][1] is the core of many state-of-the-art SAT and QBF solvers, respectively. There have been continuous improvements of the basic algorithm since the early 1990ies. Apart from technical issues such as efficient data structures and implementation details, DPLL was enhanced, for example, with clause learning and activity-based decision heuristics.

The latter approach is a strategy to dynamically change the assignment ordering of variables in paths of the decision tree constructed by DPLL-based algorithms. The SAT solver keeps track of how important a particular variable was during the search by maintaining a heuristic per-variable activity score. This way the solver can adapt the search process on the current branch with respect to information that was learnt from previous branches in the decision tree. The goal is to steer the solver out of parts of the search space which seem to be irrelevant to the solution of the problem. The use of activity scores for dynamic assignment orderings in SAT solvers is justified by semantics since, in contrast to general PCNFs, DPLL can assign variables in arbitrary order to decide the formula.

3.1.1 Variable Orderings by Prefixes in PCNFs

When it comes to QBF solving using QDPLL, variables must not be assigned in arbitrary order in general. This is due to the quantifier prefix. Occurrences of universal quantifiers which are interleaved with existential ones in the quantifier prefix might introduce *dependencies* between differently quan-

[1]We write *DPLL* to denote the algorithm for propositional logic and *QDPLL* for the QBF-specific variant of DPLL.

$$\begin{tikzcd}
& r \arrow[d] & \\
& \neg y \arrow[dl] \arrow[dr] & \\
\neg x & & x
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
& r \arrow[d] & \\
& y \arrow[dl] \arrow[dr] & \\
\neg x & & x
\end{tikzcd}$$

Figure 3.1: Two possible assignment trees of the PCNF $\psi := \forall x \exists y.\ (x \vee \neg y) \wedge (\neg x \vee y)$ from Example 3.1.1 where y is *erroneously* assigned before x.

tified variables. We address variable dependencies formally in Section 3.4 below. Actually, it is more important to know which variables do *not* depend on each other. For now, we confine our presentation to illustrative examples and to a informal notion of the term "dependency". Informally, dependencies require to choose the value of variables with respect to the value of some other variables when a PCNF is semantically evaluated. Some variable y may only be assigned by QDPLL as soon as all variables x where y depends on have been assigned already. Although activity scores for variables could be applied in QDPLL for QBF as well like in DPLL for SAT, dependencies in QBFs limit their potential positive effects. Given the quantifier prefix of a PCNF, dependencies are respected if variables are assigned "from left to right“. Neglecting this condition might yield unsound results.

Example 3.1.1. Given the satisfiable PCNF $\psi := \forall x \exists y.\ (x \vee \neg y) \wedge (\neg x \vee y)$. We assign variables from left to right using recursive semantics from Section 2.2.2. Formula ψ is satisfiable if and only if both $\psi[x]$ and $\psi[\neg x]$ are satisfiable, which is the case: both $\psi[x, y]$ and $\psi[\neg x, \neg y]$ are satisfiable. The left assignment tree in Figure 2.1 on page 13 is the corresponding PCNF-model. The value of y depends on the value of x: y must take the same value as x. Thus neither $\psi[x, \neg y]$ nor $\psi[\neg x, y]$ is satisfiable. On the contrary, assume we assign y before x, thus breaking the prefix ordering and also the dependency between x and y. Then neither $\psi[y]$ nor $\psi[\neg y]$ is satisfiable because y does not take values with respect to x since it was assigned before x. Figure 3.1 shows the two corresponding assignment trees which are *no* PCNF-models. Consequently we *cannot* conclude that ψ is satisfiable.

Example 3.1.1 points out that the prefix ordering matters if QBFs are solved using QDPLL. We now show a similar result for QBF solving by variable elimination. Expanding universal variables by Lemma 2.3.3 on page 20 *without* duplicating larger existential variables might be unsound.

Example 3.1.2. Given the satisfiable PCNF $\psi := \forall x \exists y.\ (x \vee \neg y) \wedge (\neg x \vee y)$ from Example 3.1.1 and Example 2.3.1 on page 21. Expanding x, which is not from the innermost quantifier block, by Lemma 2.3.3 is unsound. Formula ψ is not satisfiability-equivalent to

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

$$\begin{tikzcd}
 & r \arrow[d] & \\
 & \neg y \arrow[dl] \arrow[dr] & \\
\neg x & & x
\end{tikzcd}$$

Figure 3.2: PCNF-model for the PCNF $\psi' := \exists y \forall x. (x \vee \neg y) \wedge (\neg x \vee \neg y)$ from Example 3.1.3 which was obtained from the original PCNF ψ by shifting y to the front of the quantifier prefix.

because this formula further reduces to $\exists y. (y) \wedge (\neg y)$ which is unsatisfiable. This is due to the same reason as in Example 3.1.1. Variable y must take values with respect to values of x but this is not possible because we did not duplicate y. Note the difference to Example 2.3.1 on page 21.

Given Examples 3.1.1 and 3.1.2, we observe that respecting the quantifier ordering is crucial for both backtracking search and variable elimination. However, there might be situations in practice where the ordering can safely be relaxed. This would allow to take into account variables other than just the leftmost ones in the prefix for activity-based decision heuristics in QDPLL. Thus QDPLL could profit from dynamic assignment orderings in the same way as DPLL. Finally, we might be able to decide a QBF more quickly than if we had relied strictly on the ordering imposed by the quantifier prefix. For expansion, similar observations were made [16]. We point out this situation in Example 3.3.6 on page 34 below. First we need to elaborate more on the notion of dependencies.

In Example 3.1.1, breaking the prefix ordering of PCNF $\psi := \forall x \exists y. (x \vee \neg y) \wedge (\neg x \vee y)$ during semantical evaluation by assigning y before x actually corresponds to modifications of the quantifier prefix. We shift variable y to obtain PCNF $\psi' := \exists y \forall x. (x \vee \neg y) \wedge (\neg x \vee y)$ and then assign variables according to the new prefix. We will see in Section 3.4 below that shifting variables plays a central role for a precise formal definition of dependence and independence. For now, we observe that breaking the prefix ordering in a PCNF (like ψ above) might change satisfiability (like ψ' above) due to violations of variable dependencies. However, such change of satisfiability need not happen necessarily, as the following examples show.

Example 3.1.3. Given PCNF $\psi := \forall x \exists y. (x \vee \neg y) \wedge (\neg x \vee \neg y)$ which is satisfiable since both $\psi[x]$ and $\psi[\neg x]$ are satisfiable. Breaking the prefix ordering corresponds to shifting variable y to the front: $\psi' := \exists y \forall x. (x \vee \neg y) \wedge (\neg x \vee \neg y)$. Formula ψ' is satisfiable as well since $\psi'[\neg y]$ is satisfiable. Figure 3.2 shows the corresponding PCNF-model of ψ'. In fact, the value of y in ψ does *not* depend on the value of x. For both of $\psi[x]$ and $\psi[\neg x]$, assigning y to *false* satisfies the formula.[2]

Example 3.1.4. Given PCNF $\psi := \forall x \exists y.\ (x \vee \neg y) \wedge (\neg x \vee \neg y)$ from Example 3.1.3. When expanding universal variable x *without* duplicating y like in Example 3.1.2 then expansion is sound in this case. The expanded formula $\forall x \exists y.\ ((x \vee \neg y) \wedge (\neg x \vee \neg y))[x] \wedge ((x \vee \neg y) \wedge (\neg x \vee \neg y))[\neg x]$ reduces to $\exists y.\ (\neg y) \wedge (\neg y)$ which is satisfiable.

3.1.2 The Need for Dependency Analysis

Examples 3.1.1 to 3.1.4 point out two different situations in the context of backtracking search and variable elimination. First, breaking the prefix ordering, that is not assigning variables from left-to-right, could violate variable dependencies and change satisfiability. This must be avoided in practice when evaluating PCNFs and QBFs in general. However, as in Examples 3.1.3 and 3.1.4, we could benefit from situations where shifting variables in the prefix does not change satisfiability. Formula ψ from Example 3.1.3 can be decided by showing that either the original PCNF ψ or the modified ψ' is satisfiable. Here we can first assign x and then y or vice versa, and the result is sound in any case. This corresponds to increased freedom during semantical evaluation. Given ψ from Example 3.1.3, first we may only assign x unless we know that shifting y to front as in ψ' does not change satisfiability. This property of ψ' allows us to assign y before x. In order to decide ψ we actually have two possible choices of assigning variables although there is only one choice with respect to the original quantifier prefix of ψ.

Increased freedom during semantical evaluation as pointed out above allows to overcome the effects of restricted assignment orderings that follow from the quantifier prefix. The effects of such restricted orderings can be negative for QDPLL-based QBF solvers just as was reported for DPLL-based SAT solvers [72]. Assigning variables in restricted order might cause the QBF solver to spend overly much time in parts of the search space of a formula which are irrelevant to its truth value. We consider an example related to these observations in Section 3.3.3 below. Altogether, the quantifier prefix of PCNFs and the resulting kind of dependencies severely limit the freedom of QBF solvers. This does not only apply to search-based QBF solving using QDPLL but also to variable elimination. Therefore, it is crucial from a practical perspective to analyze whether variables are independent in a given PCNF since this increases the freedom for QBF solvers. For variable elimination, independence amounts to situations where the size of the set of duplicated variables B'_n by Lemma 2.3.3 on page 20 can be reduced. This in turn might reduce the size of the expanded formula (see also Example 3.3.7 on page 36). We now focus on QDPLL.

[2]See also Example 3.4.3 on page 41 for the dual case of unsatisfiable formulae.

3.2 Methods of Dependency Analysis

In the previous section we observed that the quantifier prefix of PCNFs might introduce a left-to-right ordering of variables which could be too strict. In practice it might often be possible to relax such ordering, thereby obtaining more freedom for QBF solving. For that purpose, it is crucial to know precisely which variables do *not* depend on other ones. In terms of prefix patterns like $\ldots\forall x\ldots\exists y\ldots$ or $\ldots\exists y\ldots\forall x\ldots$, this amounts to situations where x and y are independent and hence can be assigned independently by QDPLL. We refer to the act of finding out whether x is independent from y, for all variables x and y in a given PCNF, as *dependency analysis*.

In the following section, we review a well-known approach of dependency analysis for PCNFs based on scope minimization of quantifiers, called *mini-scoping*. We focus on QBFs in PCNF because it is a common format which is widely used. We also rely on PCNF when we introduce a formal framework of variable dependencies and dependency analysis in Section 3.4, called *dependency schemes*. Dependency schemes are relations over variables which represent independence. We point out severe drawbacks of mini-scoping which can be overcome by dependency schemes. Further, dependency schemes allow to assess the quality of dependency analysis. At this point, by quality we informally refer to the amount of independence identified by some approach (see also Definition 3.4.18 on page 50). Dependency analysis in PCNFs actually involves a tradeoff between quality and computational effort. Although optimal dependency analysis is infeasible, certain dependency schemes can be computed efficiently. At the same time they provide considerable information about independence between variables.

3.2.1 Maximizing Quantifier Scopes: Prenexing

PCNF is a widely used input format for QBF solvers just as CNF for SAT solving. QBF encodings of problems are typically not in PCNF right from the beginning but in arbitrary syntactic form. Given a QBF ρ with non-prenex non-CNF syntactic structure, the conversion of ρ into PCNF consists of two steps:

1. Converting ρ into a QBF ρ' which is in prenex normal form, that is all quantifiers occur in the quantifier prefix.

2. Converting ρ' into PCNF ψ.

The first step is commonly referred to as prenexing. Given QBF ρ, quantifiers are successively shifted towards the top of the parse-tree of ρ (see Definition 2.1.5 on page 9), starting at topmost quantifiers, until finally ρ' in prenex normal form is obtained. This maximizes the scopes of quantifiers, hence this approach is also called *maxi-scoping* [6]. The following well-known schema of shifting rules is used for prenexing [43, 44, 47, 96]:[3]

$$(Qx.\ \phi) \otimes \phi' \equiv Qx.\ (\phi \otimes \phi') \text{ where } Q \in \{\forall, \exists\}, \otimes \in \{\wedge, \vee\} \text{ and } x \notin V(\phi').$$

After generating ρ' in prenex normal form in the first step, its quantifier-free part is converted into CNF. In principle, this can be done by applying laws of distributivity of Boolean operators $\wedge$ and $\vee$. However, this simple approach might yield a CNF which is exponentially larger than the original formula ρ'. There are alternative approaches to obtain a CNF which is linear with respect to ρ' [39, 87, 104, 127]. The idea is to encode the output function of Boolean operators in ρ' by clauses involving fresh existential variables. These variables can then be put into the innermost quantifier block B_n, or if B_n is not existential, into a new innermost existential block [53].

Prenexing is Non-Deterministic

Quantifier shifting for QBF is a well-known and established technique for prenexing, although it has a severe drawback. There are formulae where, at a certain step during prenexing, more than one shifting rule is applicable.

Example 3.2.1 (taken from [62])**.** Given non-prenex non-CNF formula $\rho := (\exists x.\ \phi) \wedge (\forall y.\ \phi')$. When prenexing ρ, there is a non-deterministic choice of whether first shifting $\exists x$ and then $\forall y$ to the prefix or vice versa. The resulting formulae are $\rho' := \exists x \forall y.\ \phi \wedge \phi'$ and $\rho'' := \forall y \exists x.\ \phi \wedge \phi'$, respectively.

Prenexing by non-deterministic applications of quantifier shifting rules might produce quantifier prefixes with different shape. This applies to the number quantifier alternations as well as to the final position of variables in the prefix. Overall, the possibility of having multiple different prefixes for one and the same formula can be a serious problem in practice.

First, it is unnatural to have multiple PCNFs with different prefixes, and thus potentially different dependencies, for one and the same non-prenex non-CNF formula ρ. By the term "dependencies" we refer to informal prefix-induced dependencies as in Examples 3.1.1 and 3.1.3 above. It is unknown in advance which prefix might be suited for one particular solver to show best performance. By Theorem 2.2.1 on page 16, the number of quantifier alternations in a PCNF has an impact on the theoretical complexity of the decision problem with respect to the polynomial-time hierarchy by Definition 2.2.7. As argued in [47], the position of variables in the prefix could also have an influence on solver performance in practice.

Second, different prefixes might not only introduce different dependencies, but dependencies might also be spurious. The following situation was reported in [47, 62] as well. In the original formula ρ from Example 3.2.1, neither y depends on x nor vice versa. The two variables are unrelated since

[3]For simplicity, we assume that all variables in a QBF are named differently.

none of them occurs in the quantifier scope of the other. In contrast to that, y depends on x in ρ' and x depends on y in ρ'' when taking dependencies by prefixes. Following the quantifier ordering, a QDPLL-based solver can start with assigning x (but not y) in ρ' and with assigning y (but not x) in ρ''. However, dependencies in these prenex formulae are spurious in the sense that they were introduced only by prenexing but are not inherent to the original non-prenex non-CNF formula ρ.

3.2.2 Minimizing Quantifier Scopes: Anti-Prenexing

There has been research on how to alleviate the drawbacks of prenexing. The approach of *anti-prenexing* or *mini-scoping* aims at reversing the effects of prenexing for a given PCNF. Mini-scoping is common in first-order logic and QBF solving [6, 12, 43, 96, 101]. The idea is to apply the same set of quantifier shifting rules as used for prenexing. Quantifiers are shifted from the prefix back into the CNF-part of the PCNF, starting at innermost quantifiers. This way, the linear quantifier structure of the prefix is converted into a tree-like one, and so are dependencies derived therefrom. By "tree-like" we refer to information on quantifier structure which is present in the parse-tree of a formula (see Definition 2.1.5 on page 9). If Qx is a predecessor of $Q'y$ in the parse-tree, where $Q, Q' \in \{\forall, \exists\}$ and $Q \neq Q'$, that is x and y are differently quantified, then we consider y to depend on x. Otherwise, like in formula ρ from Example 3.2.1, x and y are independent. Note that this notion of dependence is still informal. We call such tree-like quantifier structure a *quantifier tree* [12] and consider examples below.

Anti-Prenexing is Non-Deterministic

Anti-prenexing converts a PCNF into a QBF in non-prenex CNF. The resulting quantifier structure is tree-like and we can extract dependency information from the quantifier tree of the formula as described above. Since this approach applies the same set of quantifier shifting rules as prenexing, it suffers from the same drawback of non-determinism.

Example 3.2.2 (taken from [85])**.** Consider the PCNF

$$\exists a,b\forall x,y\exists c,d.\ (a \vee b) \wedge (a \vee x \vee c) \wedge (b \vee c) \wedge (b \vee y \vee d).$$

Minimizing the scopes of $\exists c, \exists d, \forall x$ and $\forall y$ is deterministic and yields

$$\exists a,b.\ (a \vee b) \wedge (\forall x\exists c.\ (a \vee x \vee c) \wedge (b \vee c)) \wedge (\forall y\exists d.\ (b \vee y \vee d)).$$

Now there is the non-deterministic choice of whether to first minimize $\exists a$ and then $\exists b$ (right tree in Figure 3.3) or vice versa (left tree in Figure 3.3). Note that the left tree induces a dependency between a and y which is not the case in the right tree. Further, the left tree can be transformed into the

$$\begin{tikzcd}
 & \exists a,b \arrow[dl] \arrow[dr] & \\
\forall x \arrow[d] & & \forall y \arrow[d] \\
\exists c & & \exists d
\end{tikzcd}$$

$$\begin{tikzcd}
 & \exists b \arrow[dl] \arrow[dr] & \\
\exists a \arrow[d] & & \forall y \arrow[d] \\
\forall x \arrow[d] & & \exists d \\
\exists c & &
\end{tikzcd}$$

Figure 3.3: Quantifier trees for the PCNF $\exists a{,}b\forall x{,}y\exists c{,}d.\ (a \vee b) \wedge (a \vee x \vee c) \wedge (b \vee c) \wedge (b \vee y \vee d)$ from Example 3.2.2. Directed edges indicate successor-predecessor relationship of quantifiers in the parse tree of the mini-scoped formula. Mini-scoping of $\exists a$ in the left tree yields the tree on the right.

tree on the right by first swapping $\exists a$ and $\exists b$, since $\exists a \exists b.\phi$ is equivalent to $\exists b \exists a.\phi$, and then minimizing $\exists a$.

Anti-prenexing as in Example 3.2.2 produces two different quantifier trees for one and the same formula. From a practical perspective, we prefer the tree on the right in Figure 3.3 because it allows more freedom. A QDPLL-based solver can assign y as soon as b was assigned, since y does not depend on a according to tree-like quantifier structure of that tree. This is not possible with the tree on the left where both a and b have to be assigned before y. In this example, the worse quantifier tree on the left can be transformed into the better one on the right by another anti-prenexing step. However, in general there are formulae where none of the possible trees is best with respect to resulting dependencies. We point out that situation in Example 3.3.4 below.

3.3 Quantifier Trees are not Optimal

Summarizing our observations related to quantifier prefixes, quantifier trees, generation of PCNFs by prenexing and recovering tree-like dependencies by anti-prenexing, there seems to be a waste of work. We first convert a non-prenex non-CNF formula to PCNF which possibly involves non-deterministic applications of quantifier shifting rules. Then we undo prenexing afterwards to get back tree-like dependency information. Due to non-determinism in anti-prenexing there is no guarantee that we get back the tree-like quantifier structure that was present in the original formula.

So why not just take tree-like quantifier structure present before prenexing and use it directly for QBF solving? This approach was combined with QDPLL already [62], and it is implicitly applied in the context of search-based non-PCNF solving, where prenexing is omitted [45, 46, 66, 67, 68]. Although taking present quantifier structure allows to avoid the effects of non-determinism during prenexing and anti-prenexing, quantifier trees in general are not optimal among syntactic methods for dependency analysis.

In the following, we point out that there are more sophisticated methods for dependency analysis in PCNFs than quantifier trees. In particular, these methods have the potential to improve upon both quantifier structure present in non-PCNFs and quantifier trees by anti-prenexing. The methods we consider can all be described in the framework of *dependency schemes*. We introduce the theory of dependency schemes in Section 3.4 below. For now we focus on an informal notion of the so called *standard dependency scheme* (see also Definition 3.4.17 on page 49). We consider the standard dependency scheme to be more sophisticated than quantifier trees because it can be computed deterministically. Further, it might prove certain tree-like dependencies spurious. We will see that the standard dependency scheme is never worse than quantifier trees with respect to dependencies. It is often able to grant more freedom to a QBF solver than quantifier trees. These observations motivate the use of dependency schemes instead of quantifier trees for dependency analysis in the context of QBF solving.

3.3.1 Dependency Schemes: An Informal View

Dependency schemes provide a formal framework to express the notion of variable dependencies [111, 113]. Given some PCNF ψ, a *dependency scheme* for ψ is a binary relation D on the set of variables V of ψ where $(x, y) \in D$ if y is *assumed* to depend on x. We point out in Section 3.4 that a dependency scheme D must be constructed such that no actual dependencies are missed. However, D might contain spurious ones. If $(x, y) \notin D$ then it is guaranteed that y is independent from x, and there is no actual dependency between x and y. Note that, as above, we use the term "dependencies" rather informally in this section. Independence can be exploited by a QBF solver.

Example 3.3.1. Given PCNF $\psi := \forall x \exists y. (x \vee \neg y) \wedge (\neg x \vee y)$ from Example 3.1.1 and relations $D := \{(x, y)\}$ and $D' := \emptyset$. Relation D is a correct dependency scheme for ψ since, as pointed out in Example 3.1.1, y depends on x. A QDPLL-based solver relying on D will have to assign x before y. In contrast to that, D' is no dependency scheme since $(x, y) \notin D'$ but there is an actual dependency between x and y. Assigning y before x as suggested by D' is unsound.

Example 3.3.2. Given PCNF $\psi := \forall x \exists y. (x \vee \neg y) \wedge (\neg x \vee \neg y)$ from Example 3.1.3 and relations $D := \{(x, y)\}$ and $D' := \emptyset$. Both relations D and D' are correct dependency schemes. Variables x and y are independent as argued in Example 3.1.3 and so y can be assigned before x as suggested by D'. A QDPLL-based solver relying on D would not be able to do so due to dependency $(x, y) \in D$, which is actually spurious.

In practice, a dependency scheme D must be computed according to some strategy which influences the quality of D in terms of spurious dependencies. For example, we could analyze the syntactic structure of a PCNF to figure out dependencies. Trivially, D could be defined to correspond to the quantifier prefix: $(x, y) \in D$ if y occurs to the right of x in the prefix and is quantified differently. Such trivial dependency scheme is usually too restrictive and contains spurious dependencies. The goal is to minimize the size of a dependency scheme to allow more freedom during QBF solving.

Example 3.3.3. Given formula $\rho := (\exists x.\ \phi) \wedge (\forall y.\ \phi')$ from Example 3.2.1. We construct a dependency scheme D^{tree} for ρ from tree-like quantifier structure by adding pair (x, y) to D^{tree} if x and y are differently quantified and Qx is a predecessor $Q'y$ in the parse tree, where $Q, Q' \in \{\forall, \exists\}$ and $Q \neq Q'$. However, in this case $D^{\text{tree}} = \emptyset$ since x and y are unrelated. If we take the prenex formulae $\rho' := \exists x \forall y.\ \phi \wedge \phi'$ or $\rho'' := \forall y \exists x.\ \phi \wedge \phi'$ from Example 3.2.1, then the trivial dependency scheme D^{triv} based on the prefix contains either (x, y) or (y, x), respectively. In this case we consider D^{tree} better than either of D^{triv} because it allows more freedom due to fewer dependencies.

In Example 3.3.3 we constructed dependency schemes D^{tree} and D^{triv} from tree-like quantifier structure and quantifier prefixes, respectively. Hence these two approaches for dependency analysis fit into the general framework of dependency schemes. We observed that D^{tree} was better than D^{triv} in the sense of fewer dependencies. Actually, as shown in Section 3.4.4 below, we can compare dependency schemes in that respect.

In the following, we show by examples that D^{tree} is not optimal among syntactic approaches for dependency analysis in PCNFs. There are more sophisticated dependency schemes which can be computed from the syntactic structure. We point out that the so called standard dependency scheme D^{std} has advantages over D^{tree}. Construction is entirely deterministic and thus D^{std} is unique for every PCNF. It is superior to D^{tree} in the sense that it will never identify more dependencies than D^{tree}. We elaborate more on D^{triv}, D^{tree} and D^{std} in Section 3.4.3 below. Now we consider a concrete example illustrating the advantages of D^{std} on a given PCNF.

3.3.2 The Standard Dependency Scheme vs. Quantifier Trees

Example 3.3.4. Given PCNF

$$\exists a,b \forall x,y \exists c,d.\ (a \vee x \vee c) \wedge (a \vee b) \wedge (b \vee y \vee d).$$

Minimizing the scopes of $\exists c$, $\exists d$, $\forall x$ and $\forall y$ yields

$$\exists a,b.\ (\forall x \exists c.\ (a \vee x \vee c)) \wedge (a \vee b) \wedge (\forall y \exists d.\ (b \vee y \vee d)).$$

Now there is the non-deterministic choice whether to first minimize $\exists b$ and then $\exists a$ or vice versa. Figure 3.4 shows the quantifier trees (left and middle) for the two alternatives. Dependency schemes resulting from the trees are

$$D := \{(a, x), (x, c), (a, y), (b, y), (y, d)\}$$

$$\begin{tikzcd}
 & \exists a \arrow[dl] \arrow[dr] & & & \exists b \arrow[dl] \arrow[dr] & & & & \\
\forall x \arrow[d] & & \exists b \arrow[d] & \exists a \arrow[d] & & \forall y \arrow[d] & \exists a \arrow[d] & \exists b \arrow[d] \\
\exists c & & \forall y \arrow[d] & \forall x \arrow[d] & & \exists d & \forall x \arrow[d] & \forall y \arrow[d] \\
 & & \exists d & \exists c & & & \exists c & \exists d
\end{tikzcd}$$

Figure 3.4: Two possible quantifier trees for the PCNF $\exists a{,}b\forall x{,}y\exists c{,}d.\ (a \vee x \vee c) \wedge (a \vee b) \wedge (b \vee y \vee d)$ from Example 3.3.4 obtained by anti-prenexing (left and middle) and dependencies by the standard dependency scheme D^{std} from Example 3.3.5 (right).

and

$$D' := \{(b, x), (a, x), (x, c), (b, y), (y, d)\}$$

for the tree on the left and in the middle of Figure 3.4, respectively.

The standard dependency scheme D^{std} was introduced in [113] and is based on ideas from expansion-based solvers [16, 25]. Dependencies are identified by analyzing connections between variables in a PCNF over sequences of clauses. We present an algorithm for constructing D^{std} and related theoretical properties in Chapter 4. The following example illustrates the construction informally (see also Definition 3.4.17 on page 49).

Example 3.3.5. Given the PCNF $\exists a{,}b\forall x{,}y\exists c{,}d.\ (a\vee x\vee c)\wedge(a\vee b)\wedge(b\vee y\vee d)$ from Example 3.3.4. There is a connection between a, x and c because they occur in the common clause $(a \vee x \vee c)$. Similarly there is a connection between b, y and d by clause $(b\vee y\vee d)$. However, there is a connection neither between a and y nor between b and x. We could establish a connection between a and y, for example, by clause $(a \vee b)$ and then continue with b in clause $(b \vee y \vee d)$ but this is ruled out by definition of D^{std}. When starting from a or b, clause $(a\vee b)$ is no connection point because the two variables are in the same quantifier block. Dependencies follow from connections between differently quantified variables: $D^{\text{std}} := \{(a, x), (x, c), (b, y), (y, d)\}$.

Note that in Example 3.3.5 $(a, y) \notin D^{\text{std}}$ and $(b, x) \notin D^{\text{std}}$, hence y does not depend on a and x not on b by D^{std}. Comparing dependencies in D^{tree} from Examples 3.3.4 and 3.3.5 shows a crucial difference between tree-like quantifier structure and D^{std}. Dependencies by D^{std} can be strictly less restrictive. No matter which of the two non-deterministically constructed quantifier trees in Figure 3.4 is taken for dependency computation, either (a, y) or (b, x) is included in the resulting dependency scheme but neither in D^{std}. For the PCNF from Example 3.3.4, D^{std} is clearly superior than D^{triv} and D^{tree}. Different sets of dependencies in dependency schemes might have severe impacts on the performance of QBF solvers.

3.3.3 The Benefits of More Powerful Dependency Schemes

Above we pointed out that the standard dependency scheme D^{std} for a given PCNF might be superior to dependency analysis based on tree-like quantifier structure. Now we consider a more realistic, practical example related to QBF solving. In the following, we introduce a family of PCNFs containing specific subformulae which are provably hard to decide for QDPLL-based solvers. On these PCNFs, QDPLL relying on D^{std} can avoid to tackle the hard subformulae. Instead, solving other, easier parts of the formula yield a quick answer for the entire PCNF. As we are going to show, such quick answer cannot be obtained using QDPLL with D^{tree} in general.

Similar observations were made with respect to D^{tree} and the prefix-based trivial dependency scheme D^{triv} we computed in Example 3.3.3 [62]. Different from that we do not restrict ourselves to D^{tree}. We argue that QBF solvers in general can benefit from more powerful dependency analysis relying on dependency schemes. This also applies to schemes that have not yet been applied in practical QBF solving, like the *triangle dependency scheme* [111, 113], or the *quadrangle dependency scheme* and *resolution path dependencies* [51]. Although our example points out that D^{std} is favourable compared to D^{tree}, this also holds with respect to D^{triv} since D^{tree} is never worse than D^{triv} (see also Example 3.3.3 and Section 3.4.4 below). In addition to search-based QBF solving, we argue that QBF solvers based on variable elimination can benefit from dependency schemes as well.

Search-Based QBF Solving

Example 3.3.6. Given the unsatisfiable PCNF

$$\psi := \exists a,b,b_1,\ldots,b_n \forall x,y \exists c,d.\ (a \lor x \lor c) \land (a \lor b \lor b_1 \lor \ldots \lor b_n) \land (b \lor y \lor d) \land \phi' \land \phi'',$$

where $\phi' := (x \lor c) \land (x \lor \neg c)$ is a CNF which is unsatisfiable under the quantifier prefix of ψ. Let ϕ'' be a *pigeon hole formula* [35, 69] in CNF over n variables $\{b_1, \ldots, b_n\}$, which is unsatisfiable as well. Figure 3.5 shows quantifier trees for ψ obtained by anti-prenexing like in Figure 3.4. Again there is the non-deterministic choice whether to first minimize a and then b and all b_i or vice versa like in Example 3.3.4. The right part of Figure 3.5 shows dependencies in ψ by the standard dependency scheme D^{std} similar to Figure 3.4.

The pigeon hole formula ϕ'' of ψ from Example 3.3.6 is propositional since it does not contain universal variables. This class of unsatisfiable formulae is hard for DPLL-based solvers [69].[4] For a sufficiently large number of

[4]It was proved in [69] that *any* resolution refutation of a pigeon hole formula involves an exponential number of resolvents. Due to related results [9, 10, 22], *general resolution* is exponentially stronger than classical DPLL [37], which corresponds to *tree-like resolution*. Therefore, DPLL-based algorithms including QDPLL require exponential time to solve a pigeon hole formula as in Example 3.3.6.

$$\begin{tikzcd}
 & \exists a \arrow[dl] \arrow[dr] & \\
\forall x \arrow[d] & & \exists b, b_1, \ldots, b_n \arrow[d] \\
\exists c & & \forall y \arrow[d] \\
 & & \exists d
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \exists b, b_1, \ldots, b_n \arrow[dl] \arrow[dr] & \\
\exists a \arrow[d] & & \forall y \arrow[d] \\
\forall x \arrow[d] & & \exists d \\
\exists c & &
\end{tikzcd}
\qquad
\begin{tikzcd}
\exists a \arrow[d] & \exists b, b_1, \ldots, b_n \arrow[d] \\
\forall x \arrow[d] & \forall y \arrow[d] \\
\exists c & \exists d
\end{tikzcd}$$

Figure 3.5: Two possible quantifier trees for the PCNF ψ from Example 3.3.6 shown above on the left have similar shape as the ones in Figure 3.4. The graph on the right shows dependencies by the standard dependency scheme.

variables $b_1, \ldots, b_n$, any such solver will spend exponential time to decide that the pigeon hole formula is unsatisfiable.

Although formula ψ contains the hard subfomula ϕ'', it can be decided quickly using a QDPLL-based solver that relies on D^{std} and clause learning that we briefly addressed in Section 2.3.1. Once variable a is assigned, subformula ϕ' can be found unsatisfiable by assigning x and c. Note that x is universally quantified under the prefix of ψ and that x does not depend on variables b and $b_1, \ldots, b_n$ as shown on the right in Figure 3.5. Starting from the particular clause of ϕ' which is falsified under the current assignment, that is either $(x \vee c)$ or $(x \vee \neg c)$, the clause learning mechanism derives the empty clause from clauses $(x \vee c)$ and $(x \vee \neg c)$ by Q-resolution. This immediately shows unsatisfiability of the entire PCNF ψ. We deal with Q-resolution and the practice of clause learning in detail in Chapter 5.

The quick answer by QDPLL based on D^{std} outlined above in general cannot be obtained when using D^{tree} from the quantifier tree in the middle of Figure 3.5. Apart from a, variable b and all of $b_1, \ldots, b_n$ have to be assigned before x and c due to dependencies by the quantifier tree, which are spurious in contrast to D^{std}. This causes the solver to tackle the hard pigeon hole formula ϕ'' where it spends an exponential amount of time. Clauses in ϕ' cannot be falsified by assigning $b, b_1, \ldots, b_n$ because those variables do not occur in ϕ'. Hence the clause learning heuristics will not be able to infer the empty clause from ϕ' as before when D^{std} was used.

It is important to note that *general* Q-resolution without QDPLL is of course able to derive the empty clause from clauses in ϕ' independently from dependency schemes. In contrast to that, here we focus on a combination of QDPLL and *heuristic* clause learning which is common in search-based QBF solving [58, 78, 133, 134]. Example 3.3.6 shows that more powerful dependency schemes might uncover additional *heuristic* Q-resolution derivations, which could yield exponential gaps in solving times as shown above.

However, Example 3.3.6 does *not* show that more powerful dependency schemes always guarantee better performance of QDPLL-based solvers. Instead, such schemes might grant solvers the freedom to enter promising parts of the search space, but they do not prevent worst-case behaviour. After variable a was assigned in the example above, according to D^{std} QDPLL is free to assign x or $b, b_1, \dots, b_n$. In the latter case, the solver just as well attempts to solve the pigeon hole formula, thus running into exponential time behaviour. However, it *could have* assigned x according to D^{std}, which is impossible if D^{tree} is used. Consequently, decision heuristics also influence the benefits resulting from more powerful dependency schemes in practice.

It is straightforward to adapt the formula from Example 3.3.6 such that QDPLL with D^{tree} from the first quantifier tree in Figure 3.5 exhibits exponential behaviour as well. Hence this example shows that QDPLL with D^{tree} obtained by mini-scoping might be strictly worse than QDPLL with D^{std}. In Section 3.4.4 below we argue that D^{std} never contains more spurious dependencies than D^{tree}. If we combine QDPLL with D^{std} then we will have at least the same, and possibly more, amount of freedom for assigning variables than with D^{tree}. Together with observations related to non-determinism of mini-scoping in Section 3.2.2, this property clearly motivates to use of dependency schemes like D^{std} which are superior to D^{tree} by mini-scoping.

Variable Elimination

Apart from search-based QBF solvers, more powerful dependency schemes have the potential to improve QBF solving by variable elimination as well. According to Lemma 2.3.3 on page 20, expansion of universal variables involves duplication of existential variables which are larger by the prefix ordering. The prefix ordering actually corresponds to the trivial dependency scheme D^{triv}. The number of duplicated variables influences the size of the subformula that has to be copied. Consequently, our goal is to minimize that size, which can be achieved by dependency schemes other than D^{triv}.

Example 3.3.7 (taken from [113])**.** Given PCNF

$$\psi := \forall x, y \exists a_1, \dots, a_n. \ (x \lor \neg a_1) \land (a_1 \lor \neg a_2) \land \dots \land (a_{n-1} \lor \neg a_n) \land (y \lor a_n).$$

Expansion of x by Lemma 2.3.3 and D^{triv} has to duplicate $a_1, \dots, a_n$ and hence copy all clauses of the formula although x itself occurs only in the first clause. This doubles the size of the formula. Neither D^{std} nor any non-deterministically constructed D^{tree} can improve this situation since $D^{\text{triv}} = D^{\text{tree}} = D^{\text{std}}$ in this case. On the contrary, the *triangle dependency scheme* and generalizations thereof [51, 111, 113] find out that none of $a_1, \dots, a_n$ depends on x. The triangle dependency scheme is similar to D^{std}, but it is based on a more refined notion of connections between variables. We do not introduce it formally and only argue that it is superior to D^{std}. Thus the prefix of ψ can be modified to $\forall y \exists a_1, \dots, a_n \forall x$ by shifting x to the right over all a_i. Now x is innermost and can be expanded by Lemma 2.3.2 on page 20 where duplication of variables is not required. Actually, x can also be eliminated by universal reduction (see Section 5.3.1 on page 96). The result is similar to expansion by Lemma 2.3.2 and does not increase the size of the formula at all. This was observed in [113] already.

3.4 The Theory of Dependency Schemes

In the previous sections, we introduced the notion of variable dependencies and dependency schemes for PCNFs informally. We observed that independence between variables can be exploited by QBF solvers to relax the linear ordering in the quantifier prefix of PCNFs. This might enable solvers to tackle subproblems which can be decided quickly, which in turn could have positive effects on the overall solving process. Related observations were made in Examples 3.3.6 and 3.3.7 above. In the best case, exponential improvements with respect to solving time and space requirements can be achieved. At the same time, it is crucial to be aware of which variables depend on each other. Neglecting such dependencies in QBF solving can yield unsound results. This was pointed out in Examples 3.1.1 and 3.1.2 for solvers based on search and on variable elimination, respectively.

In this section we introduce the theoretical framework of dependency schemes, which was first presented in the context of QBF [112, 113]. Later this concept was extended to *quantified constraint satisfaction problems (CSP)* [111]. QBFs can be regarded as a special type of quantified CSPs where variables have Boolean domain with only two values *true* and *false*. Therefore, dependency schemes related to CSPs can also be applied to QBF. One of our goals is to apply the theoretical concept of dependency schemes in practical QBF solving. In Chapter 5 we point out that dependency schemes in general are inherent to QBF semantics. This is true even for work from the early days of QBF solving when dependency schemes were still unknown, like for the original QDPLL algorithm [30], for example.

We adopt the theoretical framework of dependency schemes introduced in this section from related work [111, 113], particularly from the field of quantified CSPs. As observed in [111], dependency schemes for quantified CSPs are based on a more precise notion of independence compared to the original definitions made in the context of QBF [113]. This is the reason why we rely on definitions related to quantified CSPs. Additionally, in Section 3.4.5 we attempt to summarize and extend observations made in [111, 113] with respect to practical applications of dependency schemes.

3.4.1 Variable Independence

We define independence between variables in a given PCNF ψ with respect to PCNF-models of ψ [111]. Variable independence is the foundation of dependency schemes which are formally introduced in Section 3.4.2 below. We already considered dependency schemes informally in Section 3.3.1. Examples 3.1.1 and 3.1.3 point out that changing the quantifier ordering of the prefix by shifting variables *might* change satisfiability of a PCNF. If shifting preserves satisfiability, then this is due to independence between variables the relative order of which was changed by shifting. This idea led to the first theoretical framework of independence and dependency schemes [112, 113]. However, it turned out that quantifier shifting is not adequate. There are formulae where changing the relative ordering of two variables by shifting changes satisfiability [111], although these variables are independent according to the model-based definition we focus on (see also Example 3.4.5 below).

Independence of Existential Variables

Differently from the original definition [111], we present independence between variables separately with respect to quantifier types. That is, we are interested whether two variables x and y in PCNFs with prefix patterns $\ldots\forall x\ldots\exists y\ldots$ or $\ldots\exists y\ldots\forall x\ldots$ are independent. Variables with same quantifier types are always independent. We refer to Sections 1 and 3 of the original work [111]. Example 3.4.1 below illustrates the following definitions.

Definition 3.4.1. Given an assignment tree T and a node N in T, the *depth* of N is $d(N) := 0$ if N is the root of T, and $d(N) := d(p(N)) + 1$ otherwise, where $p(N)$ is the unique parent node N' such that N is a child of N'.

Definition 3.4.2 (adapted from [111])**.** Given an assignment tree T and a variable x, $T[x]$ is the sequence of all literals $l_1, \ldots, l_m$ with $v(l_i) = x$ which are assigned by nodes $N_1, \ldots, N_m$ in T.

Note that all nodes N_i in Definition 3.4.2 have the same depth since assignments along paths in assignment trees are complete. Further, sequence $T[x]$ is uniquely determined for universal variables because siblings of nodes which assign universal variables are ordered by Definition 2.2.3 on page 12.

Definition 3.4.3. Given a PCNF $\psi := Q_1(\{x\} \cup B_1) \ldots Q_n B_n.\ \phi$ and an assignment tree T with root node r. Then T_x ($T_{\neg x}$) with root r' denotes the *immediate assignment subtree* of T such that r' assigns x to *true* (*false*) and r is parent of r'.

Example 3.4.1. Let T be the left assignment tree in Figure 3.6 on page 40. Nodes which assign variable y_1 all have depth two. Given variables x_2 and y_2, $T[x_2] = \neg x_2, x_2, \neg x_2, x_2$ and $T[y_2] = \neg y_2, \neg y_2, y_2, y_2$ are the sequences of literals assigned by nodes in T, respectively. For the two siblings of universal nodes in T, always the left (right) sibling assigns $\bot$ ($\top$) to a variable. The left (right) child of the root r of T is the immediate subtree $T_{\neg x_1}$ (T_{x_1}) of T. Note that notation T_{x_2}, for example, is undefined.

In the following, we formally define the independence of existential variables from smaller universal ones in the quantifier prefix of a PCNF. Example 3.4.2 and Figure 3.4.2 below illustrate the definition.

Definition 3.4.4 (adapted from [111])**.** Given the PCNF

$$\psi := Q_1 B_1 \dots \exists B_{i-1} \forall (\{x\} \cup B_i) \dots \exists (\{y\} \cup B_j) \dots Q_n B_n.\ \phi,$$

where x is universal, y is existential and $x < y$. Let

$$\psi' := \forall \{x\} Q_1 B_1 \dots \exists B_{i-1} \forall B_i \dots \exists (\{y\} \cup B_j) \dots Q_n B_n.\ \phi$$

be the PCNF obtained from ψ by shifting x to the front of the prefix. Then x and y are *independent* in ψ if the following condition holds:

> If the original PCNF ψ has a PCNF-model T then ψ' has a PCNF-model T' where $T'_{\neg x}[z] = T'_x[z]$ for all $z \in (\{y\} \cup B_1 \cup \dots \cup B_{i-1})$.

Variable x in Definition 3.4.4 is shifted to the front of the prefix of ψ to allow for applications of Definition 3.4.2 and 3.4.3. The intuition behind independence by Definition 3.4.4 is based on observations made in Example 3.1.3 on page 25. Since x is smaller than y in the original PCNF ψ by prefix ordering, x is assigned before y along all paths in a PCNF-model of ψ. Thus y might take different values in different subtrees with respect to x. If it is possible to construct a specific PCNF-model T for ψ where the values of y are the *same* in all subtrees with respect to x then apparently the choice of values for y does not depend on the current values of x. In this case, changing the value of the universal variable x does not force changing the value of y to obtain the PCNF-model T for ψ.

We express the existence of the specific PCNF-model T of ψ indirectly by the modified PCNF ψ', where x is outermost, and its PCNF-model T' with $T'_{\neg x}[y] = T'_x[y]$ as required in Definition 3.4.4 above. The idea of using PCNF ψ' is to explicitly compare the values of y with respect to current values of x. Variable x was shifted to the front of the prefix in ψ' to allow for such comparison relying on immediate assignment subtrees by Definition 3.4.3. The *sequences* of values assigned to y in different subtrees of T' with respect to x must be equal, that is $T'_{\neg x}[y] = T'_x[y]$. Note that y still can take different values as pointed out in Example 3.4.2 below. The same condition must hold in T' for variables which are smaller than x in the prefix of the original PCNF ψ, that is $T'_{\neg x}[z] = T'_x[z]$ for all $z \in (B_1 \cup \dots \cup B_{i-1})$. This requirement is necessary, since x was shifted to obtain ψ'. Otherwise, the variables which are smaller than x in ψ could take different values with respect to x in T', which is not possible in the original PCNF ψ. Note that T' is a PCNF-model of ψ', which was obtained from ψ by shifting x. However, due to the specific properties of T' ensured by the conditions in Definition 3.4.4, we conclude that the original PCNF ψ has some other PCNF-model T where the values of y are chosen independently from the current values of x. Consequently, x and y are independent in the original PCNF ψ.

$$\begin{tikzcd}
 & & & r \arrow[dll] \arrow[drr] & & & \\
 & \neg x_1 \arrow[d] & & & & x_1 \arrow[d] & \\
 & y_1 \arrow[dl] \arrow[dr] & & & & y_1 \arrow[dl] \arrow[dr] & \\
\neg x_2 \arrow[d] & & x_2 \arrow[d] & & \neg x_2 \arrow[d] & & x_2 \arrow[d] \\
\neg y_2 & & \neg y_2 & & y_2 & & y_2
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & & r \arrow[dll] \arrow[drr] & & & \\
 & \neg x_2 \arrow[dl] \arrow[dr] & & & & x_2 \arrow[dl] \arrow[dr] & \\
\neg x_1 \arrow[d] & & x_1 \arrow[d] & & \neg x_1 \arrow[d] & & x_1 \arrow[d] \\
y_1 \arrow[d] & & y_1 \arrow[d] & & y_1 \arrow[d] & & y_1 \arrow[d] \\
\neg y_2 & & y_2 & & \neg y_2 & & y_2
\end{tikzcd}$$

Figure 3.6: Given a satisfiable PCNF $\psi := \forall x_1 \exists y_1 \forall x_2 \exists y_2.\ \phi$. Variable y_2 is independent of x_2 by Definition 3.4.4. Assume that the assignment tree T on the left is a PCNF-model of ψ. Based on the PCNF-model T' of $\psi' := \forall x_2, x_1 \exists y_1, y_2.\ \phi$ shown on the right, sequences of values of y_2 can be compared. See also Example 3.4.2.

Example 3.4.2. Figure 3.6 illustrates independence by Definition 3.4.4. Given the satisfiable PCNF $\psi := \forall x_1 \exists y_1 \forall x_2 \exists y_2.\ \phi$. Assume that the assignment tree T on the left in Figure 3.6 is a PCNF-model of the original PCNF ψ. Variable y_2 is independent of x_2 in ψ. This is due to PCNF-model T' of $\psi' := \forall x_2, x_1 \exists y_1, y_2.\ \phi$ on the right in Figure 3.6 where x_2 was shifted to the front of the prefix. The sequence $T'_{\neg x_2}[y_2] = \neg y_2, y_2$ in subtree $T'_{\neg x_2}$ is equal to the sequence $T'_{x_2}[y_2] = \neg y_2, y_2$ in subtree T'_{x_2}. Additionally, as required by Definition 3.4.4, we have $T'_{\neg x_2}[y_1] = T'_{x_2}[y_1]$ and $T'_{\neg x_2}[x_1] = T'_{x_2}[x_1]$ for variables y_1 and x_1 which are smaller than x_2 with respect to ψ.

Related to Example 3.4.2, it is important to note that the value of the existential variable y in Definition 3.4.4 does *not* have to be either only $\top$ or only $\bot$ in both immediate assignment subtrees. The value of y can change but it is crucial whether such change is due to x or some other universal variable. In Example 3.4.2, the universal variable x_1 forces the change of the value of y_2 but not x_2, as can be seen from the PCNF-models in Figure 3.6.

Further, note that by Definition 3.4.4 any existential variable y larger than x in the original PCNF ψ is trivially independent from x if ψ is unsatisfiable. In this case, ψ does not have a PCNF-model and hence the condition of independence is vacuously true.

The additional criterion for restricting the value of variables $(B_1 \cup \ldots \cup B_{i-1})$ which are smaller than x in ψ takes into account universal variables as well. Actually, this requirement is crucial for existential variables only. Nodes assigning universal variables always have exactly one sibling. By Definition 2.2.3 on page 12, the siblings of universal nodes in assignment trees are ordered. Therefore, the sequences of values of universal variables by Definition 3.4.2 are always unique.

Independence of Universal Variables

Now we consider PCNFs with prefix pattern $\ldots \exists y \ldots \forall x \ldots$ and check if x is independent from y. This situation is dual to the one pointed out in Definition 3.4.4 and is illustrated by the following example.

Example 3.4.3. Given the unsatisfiable PCNF $\psi := \exists y \forall x.\ (y \vee x) \wedge (\neg y \vee x)$. We swap the variables in the prefix to obtain $\psi' := \forall x \exists y.\ (y \vee x) \wedge (\neg y \vee x)$. The prefix order of ψ' allows different values of y with respect to x in assignment trees of ψ'. As ψ' is also unsatisfiable, we conclude that x is independent from y in ψ since the formula remains unsatisfiable under additional freedom to select different values for y according to the prefix of ψ'.[5]

Given Example 3.4.3, interesting cases of independence arise from *unsatisfiable* PCNFs with prefix pattern $\ldots \exists y \ldots \forall x \ldots$ only. For satisfiable ones, independence of x and y is trivial, which is explained informally as follows. If a PCNFs ψ with prefix pattern $\ldots \exists y \ldots \forall x \ldots$ is satisfiable, then y is assigned before x in every path of every PCNF-model of ψ which fixes the value of y for all values of x. We could assign the same, fixed value to y after swapping x and y in the prefix of ψ. Hence x and y are independent if ψ is satisfiable.

If a PCNF ψ with prefix pattern $\ldots \exists y \ldots \forall x \ldots$ is unsatisfiable then we want to know whether the formula remains unsatisfiable even if we allow different values for the existential variable y with respect to universal variable x. If so, then we regard x and y as independent. However, since $y < x$ in the prefix ordering we cannot express the choice of different values for y based on assignment trees of ψ as we did in Definition 3.4.4. Along the paths of an assignment tree of ψ, y is always assigned before x and hence the values of y are fixed for all values of x.

Different from Definition 3.4.4, we argue indirectly as follows. Given an unsatisfiable PCNF ψ with prefix pattern $\ldots \exists y \ldots \forall x \ldots$. We want to check whether x and y are independent. Instead of the original statement

> "If PCNF ψ, where the value of y is assigned before x in every assignment tree, is *unsatisfiable*, then ψ, where we allow different values for y with respect to x, is *unsatisfiable*."

we consider the *contrapositive* statement

> "If PCNF ψ, where we allow different values for y with respect to x, is *satisfiable*, then PCNF ψ, where the value of y is assigned before x in every assignment tree, is *satisfiable*."

To express the contrapositive statement above in terms of assignment trees like in Definition 3.4.4, we obtain the PCNF ψ' from ψ by shifting

[5]Compare to Example 3.1.3 on page 25.

x to the front of the quantifier prefix of ψ. That is, ψ' has prefix pattern $\forall x \dots \exists y \dots$. In PCNF ψ' we have $x < y$ by prefix ordering.

We use ψ' to express the antecedent

> "If PCNF ψ, where we allow different values for y with respect to x, is *satisfiable*,..."

of the contrapositive statement by searching for a PCNF-model T' of ψ' such that variables $z \neq y$ where $z < x$ in the *original* PCNF ψ are assigned the *same* values with respect to x in T'. The restriction on the values of variables z is necessary because we want to allow only variable y to get different values with respect to x in the PCNF-model T' of ψ'. Actually, we can think of that restriction as a "simulation" of the *original* prefix ordering of PCNF ψ where we had $z < x$. Thus, every variable z is assigned before x in a PCNF-model of the original PCNF ψ. We enforce that original assignment ordering indirectly by imposing the restriction on variables z in the PCNF-model T' of the modified PCNF ψ', where x is outermost in the prefix and therefore is assigned before z.

In order to express the consequent

> "..., then PCNF ψ, where the value of y is assigned before x in every assignment tree, is *satisfiable*."

of the contrapositive statement, we search for another PCNF-model T'' of ψ' where, in addition to the above restriction of values for variables z with $z < x$ in the original PCNF ψ, we require that y is assigned the same value with respect to x.

Given the contrapositive statement from above and the modified PCNF ψ' where x is outermost in the prefix, we formally define independence of x and y as follows.

Definition 3.4.5 (adapted from [111]). Given a PCNF

$$\psi := Q_1B_1 \dots \forall B_{i-1} \exists(\{y\} \cup B_i) \dots \forall(\{x\} \cup B_j) \dots Q_nB_n.\ \phi,$$

where y is existential, x is universal and $y < x$. Let

$$\psi' := \forall\{x\} Q_1B_1 \dots \forall B_{i-1} \exists(\{y\} \cup B_i) \dots \forall(B_j) \dots Q_nB_n.\ \phi$$

be the PCNF obtained from ψ by shifting x to the front of the prefix. Then x and y are *independent* in ψ if the following condition holds:

If ψ' has a PCNF-model T' such that

$$T'_{\neg x}[z] = T'_x[z] \text{ for all } z \in ((B_1 \cup \dots \cup B_{j-1}) \setminus \{y\})$$

then ψ' has a PCNF-model T'' such that

$$T''_{\neg x}[z] = T''_x[z] \text{ for all } z \in (B_1 \cup \dots \cup B_{j-1} \cup \{y\}).$$

$$\begin{tikzcd}
 & & & r \arrow[d] & & & \\
 & & & u \arrow[dll] \arrow[drr] & & & \\
 & \neg v \arrow[d] & & & & v \arrow[d] & \\
 & y \arrow[dl] \arrow[dr] & & & & y \arrow[dl] \arrow[dr] & \\
\neg x & & x & & \neg x & & x
\end{tikzcd}$$

$$\begin{tikzcd}
 & & & r \arrow[dll] \arrow[drr] & & & \\
 & \neg x \arrow[d] & & & & x \arrow[d] & \\
 & u \arrow[dl] \arrow[dr] & & & & u \arrow[dl] \arrow[dr] & \\
\neg v \arrow[d] & & v \arrow[d] & & \neg v \arrow[d] & & v \arrow[d] \\
\neg y & & \neg y & & y & & y
\end{tikzcd}$$

$$\begin{tikzcd}
 & & & r \arrow[dll] \arrow[drr] & & & \\
 & \neg x \arrow[d] & & & & x \arrow[d] & \\
 & u \arrow[dl] \arrow[dr] & & & & u \arrow[dl] \arrow[dr] & \\
\neg v \arrow[d] & & v \arrow[d] & & \neg v \arrow[d] & & v \arrow[d] \\
y & & \neg y & & y & & \neg y
\end{tikzcd}$$

Figure 3.7: Given the PCNF $\psi := \exists u \forall v \exists y \forall x.\ \phi$ from Example 3.4.4. An assignment tree of ψ (no PCNF-model) is shown on the left. The other two trees are PCNF-models T' and T'' of the modified PCNF $\psi' := \forall x \exists u \forall v \exists y.\ \phi$ as used in Definition 3.4.5 to explain independence of x and y.

Example 3.4.4 (from [111])**.** Assume that the PCNF $\psi := \exists u \forall v \exists y \forall x.\ \phi$ is unsatisfiable. We argue that x and y are independent by Definition 3.4.5. An arbitrary assignment tree of ψ is shown on the left of Figure 3.7. Note that y is assigned before x in that tree. We obtain the PCNF $\psi' := \forall x \exists u \forall v \exists y.\ \phi$ from ψ by shifting x to the front of the prefix. The PCNF-model T' of ψ' is shown in the middle of Figure 3.7. Note that $T'_{\neg x}[u] = T'_x[u] = u$ as required for T' by Definition 3.4.5 because $u < x$ in the prefix of the original PCNF ψ. Further, $T'_{\neg x}[y] \neq T'_x[y]$ hence y takes different values in T'. Finally, the PCNF-model T'' of ψ' is shown in the right of Figure 3.7. Like with T', we have $T''_{\neg x}[u] = T''_x[u] = u$ and additionally $T''_{\neg x}[y] = T''_x[y] = y, \neg y$.

The two cases of independence of two differently quantified variables by Definitions 3.4.4 and 3.4.5 are sufficient. Two equally quantified variables x and y are always considered to be independent [111]. The question of whether x and y in a PCNF with prefix pattern $\ldots \exists x \ldots \forall z \ldots \exists y \ldots$ are independent, for example, can be dealt with either by Definition 3.4.4 or 3.4.5 with respect to x and z or z and y, respectively.

Expressing Independence by Shifting Variables

Our informal notion of independence used in earlier parts of this chapter relies on modifications of the quantifier prefix by shifting variables like in Examples 3.1.1, 3.1.3 and 3.4.3. If swapping two variables x and y in the prefix of a PCNF preserves (un)satisfiability then we consider them as independent. Independence was explained by shifting in the first systematic approach of dependency schemes [112, 113]. However, independence based on assignment trees by Definitions 3.4.4 and 3.4.5 is strictly more refined, as observed in [111, 113]. Problems occur if variables x and y are not from quantifier blocks which are adjacent to each other in the prefix. Swapping them also changes the relative order of x, y and other variables between x

and y. If swapping changes (un)satisfiability then this might also be due to changes of that relative order. Although the notion of independence based on assignment trees applies shifting as well to generate PCNFs ψ' as in Definitions 3.4.4 and 3.4.5, it is more fine-grain. It explicitly rules out the effects of changes of the relative order of x, y and other variables. This is achieved by restricting the values of variables smaller than the shifted one in assignment trees of ψ' as done in Definitions 3.4.4 and 3.4.5.

Example 3.4.5 (taken from [111])**.** Given a PCNF $\psi := \forall x \exists u \forall y \exists v.\ \phi$. By applying Definitions 3.4.4 and 3.4.5, assume that we find out that x and u, u and y, y and v are not independent, respectively. However, we still could have that x and v are independent by Definition 3.4.4. In this case, variables x and v cannot be swapped without changing the (un)satisfiability of ψ. Doing so would also change the relative order of y and v, which are not independent.

It is important to note that independence by Definitions 3.4.4 and 3.4.5 is precise. If variables are *not* independent then swapping them in the prefix *will* change (un)satisfiability of the PCNF. Further, the other direction of the situation illustrated in Example 3.4.5 holds: if two variables can be swapped in the prefix of a PCNF then they are also independent by Definitions 3.4.4 and 3.4.5. This correlation was pointed out in [111].

3.4.2 Dependency Schemes

Given the precise notion of independence from the previous section, we now introduce dependency schemes that we already considered informally in Section 3.3.1 on page 31. We argued that QBF solvers can profit from independence identified by some dependency scheme D. Dependencies according to D given by $(x, y) \in D$ must be respected in QBF solving to guarantee sound results. However, we observed that dependencies can also be spurious in the sense that they do not correspond to actual dependencies in a PCNF. This was pointed out in Examples 3.3.1 and 3.3.2.

We first define dependency schemes relying on Definitions 3.4.4 and 3.4.5. This gives rise to optimal dependency schemes where full and exact information of independence is represented. That is, such optimal dependency schemes do not contain any spurious dependencies. Unfortunately, this optimal approach is infeasible as pointed out by Proposition 3.4.3 on page 47. Therefore, we introduce dependency schemes which can be computed efficiently at the cost of optimality and comment on related properties. The goal is to apply dependency schemes to relax the linear ordering of the variables as given by the quantifier prefix of PCNFs. Chapters 4 and 5 present related applications of dependency schemes in search-based QBF solving.

Definition 3.4.6. Given a PCNF ψ over a set of variables V, $V_\times := \{(x, y) \in ((V_\forall \times V_\exists) \cup (V_\exists \times V_\forall)) \mid x < y\}$ is the set of all pairs (x, y) of differently quantified variables such that x is smaller than y in the quantifier prefix of ψ.

Definition 3.4.7 (adapted from [111])**.** Given a PCNF ψ over variables V. A binary relation $D \subseteq V_\times$ is a *dependency scheme* for ψ if for all pairs $(x, y) \in V_\times$ the following holds: if $(x, y) \notin D$ then x and y are independent in ψ. We write $x \prec_D y$ if $(x, y) \in D$, or simply $x \prec y$ if D is arbitrary or clear from the context.

Definition 3.4.8. Given a PCNF ψ and a dependency scheme D for ψ. A pair $(x, y) \in D$ is a *dependency with respect to* D. The set of dependencies of some variable x with respect to D is $D(x) := \{y \mid (x, y) \in D\}$. Variables in $D(x)$ *depend* on x with respect to D. A dependency $(x, y) \in D$ is *spurious* if x and y are independent in ψ.

We say that "variable x depends on y" in a PCNF if the dependency scheme D with $(x, y) \in D$ is arbitrary or clear from the context.

Definition 3.4.9. Given a PCNF ψ over variables V and a dependency scheme D for ψ. The *transitive closure* D^* of D is defined as follows:

- $D_0 := D$.
- For $i \geq 0$, $D_{i+1} := D_i \cup \{(a, y) \in V_\times \mid \exists x, b \in V : (a, x) \in D_i, (x, b) \in D_i \text{ and } (b, y) \in D_i\}$.
- $D^* := D_i$ where i is the smallest natural number such that $D_i = D_{i+1}$.

Definition 3.4.10. Given a PCNF ψ, a dependency scheme D for ψ is *transitive* if and only if $D = D^*$.

Due to set $V_\times$ by Definition 3.4.6, dependency schemes never contain pairs of equally quantified variables. Similarly, such pairs are excluded from sets D_i in Definition 3.4.9. Given variables x and y, if $q(x) = q(y)$ then, as noted above, such variables are always regarded as independent and hence $(x, y) \notin D$. Similarly, by Definition 3.4.6 pairs (x, y) where $x > y$ are excluded from any dependency scheme D. This is no limitation because the symmetric pair (y, x) can be added to D if necessary.

Given a PCNF ψ and a dependency scheme D for ψ. If $(x, y) \notin D$ then search-based QBF solvers are free to assign x and y in arbitrary order, provided that other dependencies in D are respected. For example, if ψ is satisfiable, $q(x) = \forall$ and $q(y) = \exists$, then by Definition 3.4.4 there are PCNF-models similar to T and T' of ψ where the value of y either might change with respect to the value of x or where that value is fixed for all values of x. This two possibilities cover the cases when the solver assigns x before y and y before x, respectively. Satisfiability of ψ is not affected by the different assignment orderings of x and y. Dependency schemes allow to generalize semantics based on assignment trees by Definition 2.2.3.

Proposition 3.4.1. *Given a PCNF ψ and a dependency scheme D for ψ, ψ is satisfiable if and only if it has a PCNF-model T such that if $(x, y) \in D$ then x is predecessor of y on every path in T.*

Proof. Due to Proposition 3.4.5 on page 51 below, the trivial dependency scheme D^{triv} given by the prefix of ψ is the largest possible dependency scheme for ψ. Therefore, we have $D \subseteq D^{\text{triv}}$. If $(x, y) \notin D$ then x and y can be assigned in arbitrary order along the paths in an assignment tree. In this case, independence by Definitions 3.4.4 and 3.4.5 guarantees that a PCNF-model can (not) be constructed. Thus (un)satisfiability is preserved. □

Satisfiability of PCNFs by Proposition 3.4.1 in general imposes weaker constraints on PCNF-models with respect to the ordering of assignments along paths. The prefix-based linear ordering of paths in assignment trees by Definition 2.2.3 can be relaxed by any arbitrary dependency scheme D (see also Example 3.4.8 and Figure 3.9 on page 51). Spurious dependencies in a dependency scheme restrict the freedom of QBF solvers as pointed out in Example 3.3.2. If the dependency $(x, y) \in D$ is spurious then the solver *never* assigns y before x although the two variables are independent.

Intractability of Optimal Dependency Schemes

Definition 3.4.7 does not prevent spurious dependencies. The fewer dependencies in D are spurious, the more freedom is granted by D to QBF solvers. Based on that, we define a criterion of optimality of dependency schemes.

Definition 3.4.11. A dependency scheme D^{opt} for PCNF ψ is *optimal* if and only if the following condition holds for all pairs $(x, y) \in V_\times$:

$$(x, y) \notin D^{\text{opt}} \text{ if and only if } x \text{ and } y \text{ are independent in } \psi.$$

Proposition 3.4.2. *The optimal dependency scheme D^{opt} for a PCNF ψ is unique.*

Proof. Given PCNF ψ, assume that dependency schemes D and D' for ψ are both optimal and that $D \neq D'$ due to, for example, $(x, y) \notin D$ but $(x, y) \in D'$. Since D is optimal and $(x, y) \notin D$, x and y are independent by Definition 3.4.11. As $(x, y) \in D'$ and D' is optimal, x and y are not independent by Definition 3.4.11, which is a contradiction. Hence $D = D'$. □

Definition 3.4.11 improves upon Definition 3.4.7 by including the other direction of the implication: if x and y are independent then $(x, y) \notin D$. This way, spurious dependencies are avoided at all. Consequently, there is no dependency scheme D with $D \subset D^{\text{opt}}$. If any pair $(x, y) \in D^{\text{opt}}$ is removed, then D^{opt} is no longer a dependency scheme. Given a PCNF ψ, we can compute the optimal dependency scheme D^{opt} by checking if x and y are independent for all pairs $(x, y) \in V_\times$. While D^{opt} grants full freedom to QBF solvers, it typically cannot be applied in practice.

Proposition 3.4.3 (related to Proposition 2 in [113])**.** *Computing the optimal dependency scheme D^{opt} for a PCNF ψ is at least as hard as solving ψ.*

Proof. The conditions in Definitions 3.4.4 and 3.4.5 involve searching for PCNF-models. Thus the satisfiability of the PCNFs ψ and ψ' must be decided, a problem which is PSPACE-complete. To compute D^{opt}, this must be carried out $\mathcal{O}(|V_{\times}|)$ times in the worst case. □

3.4.3 Tractable Dependency Schemes

Following from Proposition 3.4.3, optimal dependency schemes cannot be applied for QBF solving in practice. Therefore, we have to trade optimality for efficiency. Different from optimal dependency schemes, we want to allow spurious dependencies in dependency schemes. Under that relaxation, it turns out that there are several dependency schemes which can be computed efficiently. At the same time, these schemes are able to grant considerable freedom to QBF solvers to improve their performance in practice. We show related experimental results in Chapter 5. In this section, we introduce dependency schemes that we considered informally in Section 3.3.1 on page 31. All of these schemes can be computed in polynomial-time with respect to the size of a PCNF ψ. For convenience, we also use the term "dependency scheme" to denote an approach to construct a relation D which then has the properties of a dependency scheme for some PCNF ψ by Definition 3.4.7.

Definition 3.4.12 ([113])**.** A dependency scheme D is *tractable* if, for all PCNFs ψ and for all pairs $(x, y) \in V_{\times}$, it can be decided in polynomial-time whether $(x, y) \in D$ or $(x, y) \notin D$.

Tractability of dependency schemes is a rather theoretical concept. In the worst case, $\mathcal{O}(|V|^2)$ pairs have to be considered to compute D which yields at least a quadratic overhead in the end. Although being polynomial-time, this could be too expensive for practical applications if the number of variables is large. In Chapter 4 we present an algorithm to compute the standard dependency scheme D^{std} which was briefly illustrated in Example 3.3.5. It turns out that D^{std} can be computed and represented efficiently in practice.

Tractable dependency schemes by Definition 3.4.12 do not prevent spurious dependencies. Depending on the concrete algorithm that is used to compute D, a dependency $(x, y) \in D$ *might* be spurious. If $(x, y) \notin D$ then variables x and y must be independent because otherwise D is not a dependency scheme by Definition 3.4.7.

In the following, we define tractable dependency schemes which can be computed by analyzing the syntactic structure of a PCNF. In Section 3.4.4 below, we compare these schemes by the number of spurious dependencies, which gives rise to a hierarchy of dependency schemes.

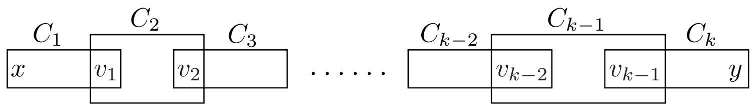


Figure 3.8: Variables x and y are connected to each other by an X-path with clauses $C_1, \dots, C_k$ where $X := \{v_1, \dots, v_{k-1}\}$. See also Definition 3.4.15 and Example 3.4.6.

Definition 3.4.13 (taken from [113]). Given a PCNF ψ over variables V, $D^{\text{triv}} := V_\times$ is the *trivial dependency scheme* for ψ.

Dependencies in the trivial dependency scheme D^{triv} for a PCNF ψ correspond exactly to the linear ordering of the quantifier prefix of ψ. This causes variables to be assigned "from left to right" in classical QDPLL [30], for example. In Chapter 5 we point out the benefits of applying dependency schemes other than D^{triv} in QDPLL. Due to Proposition 3.4.5 below, D^{triv} is the largest possible dependency scheme for a PCNF as it is equal to $V_\times$.

Definition 3.4.14. Given a PCNF ψ, let ψ' be a non-deterministically constructed non-prenex CNF formula obtained from mini-scoping by quantifier shifting rules from Section 3.2.1 on page 27. The parse tree of ψ' induces the dependency scheme $D^{\text{tree}} \subseteq V_\times$ for ψ: $(x, y) \in D^{\text{tree}}$ if x is predecessor of y in the parse tree of ψ' and $q(x) \neq q(y)$.

Dependency analysis by quantifier trees [12] is closely related to Definition 3.4.14, but different from the former we do not take into account duplication of universal variables by distributivity: $\forall x.\ (\phi \wedge \phi') \equiv (\forall x.\ \phi) \wedge (\forall x'.\ \phi')$. We illustrated the construction of D^{tree} in Examples 3.3.3 and 3.3.4.

Next, we introduce the standard dependency scheme D^{std} we informally discussed in Sections 3.3.1 and 3.3.2 already. Construction of D^{std} is based on checking if variables are connected to each other by particular sequences of clauses, which was illustrated in Example 3.3.5 on page 33.

Definition 3.4.15 ([82, 113]). Given PCNF ψ over variables V, set $X \subseteq V$ and variables $x, y \in V$. An *X-path* between x and y is a sequence $C_1, \dots, C_k$ of clauses in ψ such that $x \in C_1$, $y \in C_k$ and $C_i \cap C_{i+1} \cap X \neq \emptyset$ for $1 \leq i < k$.

An X-path by Definition 3.4.15 represents a connection between two variables by a sequence of clauses such that adjacent clauses have variables in X in common (see also Figure 3.8).

Example 3.4.6. For the PCNF from Example 3.3.4, there are X-paths between b and y for $X = \emptyset$ and clause $(b \vee y \vee d)$ and between a and y for $X = \{b\}$ and clauses $(a \vee b)$ and $(b \vee y \vee d)$.

Definition 3.4.16. Given PCNF over variables V and $q \in \{\exists, \forall\}$, $V_{q,i} := \{x \in V_q \mid i \leq \delta(x)\}$ is the set of all variable whose quantifier type equals q and which are from quantifier block B_i or any larger block.

Definition 3.4.17 ([82, 113])**.** Given PCNF ψ over variables V, the *standard dependency scheme* for ψ is $D^{\text{std}} := \{(x, y) \in V_\times \mid$ there is an X-path between x and y for $X := V_{\exists,i}$ where $i := \delta(x) + 1\}$.

There is a dependency $x \prec y$ by the standard dependency scheme if x is connected to y by existential variables larger than x. This requirement is expressed by setting $i := \delta(x) + 1$ and $X := V_{\exists,i}$. Alternatively, D^{std} can be computed with $i := \delta(x)$ [113] but the variant from Definition 3.4.17 is more refined because it allows fewer variables to be used for X-paths.

Example 3.4.7. For the PCNF $\exists a,b\forall x,y\exists c,d.\ (a \vee x \vee c) \wedge (a \vee b) \wedge (b \vee y \vee d)$ from Example 3.3.4 on page 32, we have $D^{\text{triv}} := \{(a, x), (a, y), (b, x), (b, y), (x, c), (x, d), (y, c), (y, d)\}$. For D^{std} and D^{tree}, we refer to Examples 3.3.4 and 3.3.5.

Lemma 3.4.1. *D^{triv}, D^{tree} and D^{std} are dependency schemes for a PCNF ψ by Definition 3.4.7.*

Proof. Our definition of independence is based on assignment trees [111] whereas the one in [113] relies on shifting variables in the quantifier prefix of a PCNF. As argued in Example 3.4.5 above, our definition is strictly more refined. Therefore, the proof in [113] that D^{std} is a dependency scheme applies to our context as well. Both D^{triv} and D^{tree} are dependency schemes since $D^{\text{std}} \subseteq D^{\text{tree}} \subseteq D^{\text{triv}}$ by Proposition 3.4.5 below. □

Lemma 3.4.2. *D^{triv}, D^{tree} and D^{std} are transitive for all PCNFs ψ.*

Proof. Let ψ be an arbitrary but fixed PCNF and D^{triv}, D^{tree} and D^{std} be the respective dependency schemes for ψ. Since $D^{\text{triv}} = V_\times$, transitivity follows immediately because $V_\times$ is a transitive relation. For D^{tree}, assume $(a, x) \in D^{\text{tree}}$, $(x, b) \in D^{\text{tree}}$ and $(b, y) \in D^{\text{tree}}$. Then by Definition 3.4.14, a is predecessor of x, which is predecessor of b, which is predecessor of y in the parse tree of the mini-scoped formula ψ'. Thus a is also predecessor of y and $q(a) \neq q(y)$, hence $(a, y) \in D^{\text{tree}}$.

For D^{std}, assume that $(a, x) \in D^{\text{std}}$, $(x, b) \in D^{\text{std}}$, $(b, y) \in D^{\text{std}}$ and $q(a) = \forall$ (the proof works analogously for $q(a) = \exists$). Then $q(x) = \exists$, $q(b) = \forall$ and $q(y) = \exists$. By Definition 3.4.17 $a < x < b < y$ in the prefix of PCNF ψ. Therefore, the X-paths from a to x, from x to b and from b to y can be chained to obtain an X-path from a to y. Such connection can be established with existential variables only, which fulfills the requirement of Definition 3.4.17. Particularly, the clause containing b in the path from x to b must contain at least one existential variable which is larger than x and, due to $a < x$, also larger than a. Hence $(a, y) \in D^{\text{std}}$. □

Proposition 3.4.4. *D^{triv}, D^{tree} and D^{std} are tractable.*

Proof. Given a PCNF ψ over variables V, D^{triv} can be computed in $\mathcal{O}(|V|^2)$ time by traversing the quantifier prefix of ψ and collecting pairs in D^{triv}. For D^{tree} and D^{std}, polynomial-time algorithms were suggested in [12] and [113], respectively. □

We consider efficient computation and representation of D^{std} in Chapter 4. As argued in Example 4.1.1 on page 57, a direct application of Definition 3.4.17 might be too costly for practical applications.

3.4.4 Comparing Dependency Schemes

Tractable dependency schemes by Definition 3.4.12 do not prevent spurious dependencies. Actually, tractability comes at the cost of a loss of optimality. In this section we compare dependency schemes by the number of spurious dependencies. This way, we introduce a refinement relation on dependency schemes which also affects practical applications.

Definition 3.4.18 (corresponds to Definition 2 in [85], related to Proposition 6 in [113])**.** Given two dependency schemes D and D', D *refines* D' if $D \subseteq D'$ for all PCNFs ψ. D *strictly refines* D' if additionally $D \neq D'$ for at least one PCNF ψ.

Equivalently, if D refines D' then we can regard D as *more refined* than D'. See also [51] for observations related to refinements of dependency schemes. If a dependency scheme D (strictly) refines some other scheme D', then this is always due to spurious dependencies which occur in D' but not in D. Therefore, we favour D over D' for practical applications because it grants more freedom to QBF solvers. In general, we consider all dependency schemes D for a PCNF ψ other than the optimal dependency scheme D^{opt} by Definition 3.4.11 as *overapproximations* thereof, since D^{opt} refines all other dependency schemes for ψ (see also Proposition 3.4.5 below).

Given such overapproximation D, we can think of pairs in D as pairs of variables for which independence as required by Definition 3.4.7 could not be proved by the particular algorithm A used to compute D. In practice, dependencies have to be added to D in conservative fashion. If independence between variables x and y cannot be proved by algorithm A then still the dependency (x, y) must be added to D in order to fulfill the formal requirements of a dependency scheme. Otherwise, if $(x, y) \notin D$ and the dependency (x, y) is *not spurious*, then D is in fact not a dependency scheme and using D for QBF solving *will* yield unsound results if the solver assigns y before x. This is due to the precise notion of independence by Definitions 3.4.4 and 3.4.5. If $(x, y) \in D$ and the dependency (x, y) is spurious, then D is a dependency scheme and thus guarantees sound results, yet it misses independence of x and y at the cost of additional freedom in QBF solving.

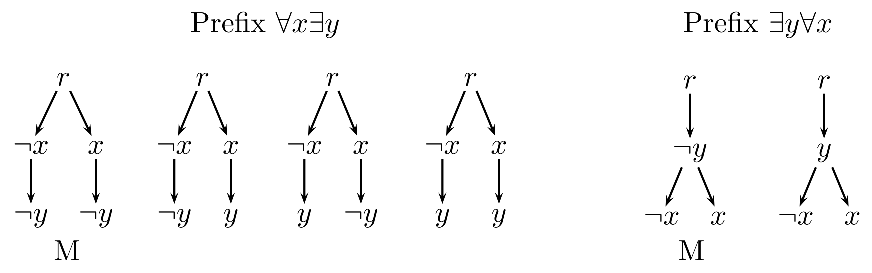


Figure 3.9: All four possible assignment trees for the PCNF $\psi := \forall x \exists y.\ (x \vee \neg y) \wedge (\neg x \vee \neg y)$ from Example 3.4.8 are shown on the left. Since x and y are independent, these variables can be swapped in the prefix. The two possible assignment trees for the modified PCNF $\psi' := \exists y \forall x.\ (x \vee \neg y) \wedge (\neg x \vee \neg y)$ are shown on the right. Assignment trees marked with "M" are PCNF-models.

Example 3.4.8. Given the satisfiable PCNF $\psi := \forall x \exists y.\ (x \vee \neg y) \wedge (\neg x \vee \neg y)$ from Example 3.1.3 on page 25. We have $D^{\text{opt}} = \emptyset$ and $D^{\text{triv}} = \{(x, y)\}$ for ψ. A search-based solver relying on D^{triv} must assign x before y. Possible assignment trees that can be constructed by such solver are shown on the left in Figure 3.9. If D^{opt} is applied instead, then *additionally* the solver can assign y before x, which corresponds to swapping x and y in the prefix of ψ. Additional assignment trees are shown on the right in Figure 3.9. Given D^{opt}, the solver can construct six assignment trees, where two are PCNF-models ($\frac{2}{6} = 33\%$). In contrast to that, a solver with D^{triv} can construct four assignment trees where only one is a PCNF-model ($\frac{1}{4} = 25\%$). Hence in this example, chances to discover a PCNF-model are higher if a dependency scheme is used which refines D^{triv}.

As illustrated by Example 3.4.8, dependency schemes which strictly refine D^{triv} grant additional freedom to QBF solvers. This increases the number of assignment trees and PCNF-models that can be constructed. Our goal is to apply dependency schemes other than D^{triv}, which prevails in QBF literature, in QBF solving. However, the actual proof-theoretic effects related to dependency schemes are still unknown and thus considered future work.

Partial Order on Dependency Schemes

Given the refinement relation from Definition 3.4.18, we consider dependency schemes D^{opt} and D^{triv} to be best and worst, respectively, for practical applications in QBF solving. For a given PCNF ψ, the former is free of spurious dependencies by definition whereas the latter contains the largest possible number of spurious ones.

Proposition 3.4.5 (see also Figure 1 in [51]). *Given a PCNF ψ. The refinement relation from Definition 3.4.18 based on subset properties induces*

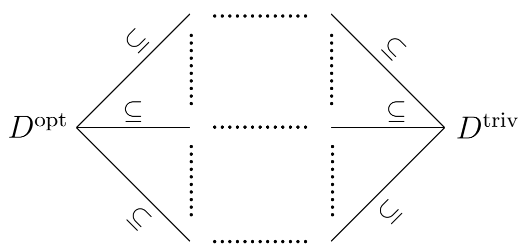


Figure 3.10: Partial order on dependency schemes by Proposition 3.4.5.

a partial order on all dependency schemes for ψ with D^{triv} and D^{opt} as least and greatest elements, respectively (see Figure 3.10).

Lemma 3.4.3. *For every PCNF ψ, $D^{\text{std}} \subseteq D^{\text{tree}} \subseteq D^{\text{triv}}$.*

Proof. We show $D^{\text{std}} \subseteq D^{\text{tree}}$ and $D^{\text{tree}} \subseteq D^{\text{triv}}$. The claim then follows by transitivity of $\subseteq$.

$D^{\text{tree}} \subseteq D^{\text{triv}}$: if $(x, y) \in D^{\text{tree}}$ then x is predecessor of y in the parse tree of the mini-scoped formula ψ' by Definition 3.4.14. Then y was shifted before x during mini-scoping. Therefore, $x < y$ by the quantifier prefix of ψ and hence $(x, y) \in D^{\text{triv}}$.

$D^{\text{std}} \subseteq D^{\text{tree}}$: if $(x, y) \in D^{\text{std}}$ then there is an X-path between x and y for clauses $C_1, \ldots, C_k$ and $X = V_{\exists,i}$ where $i = \delta(x)+1$. By Definition 3.4.15, x occurs in C_1 and y in C_k. See also Figure 3.8. When constructing D^{tree}, then y and variables in X are shifted before x since $x < y$ and $x < y_i$ for all $y_i \in X$. Clauses C_1 and C_k are in the minimized scopes of x and y in the mini-scoped formula ψ'. Since variables in X are common to at least two clauses in the X-path, finally x is a predecessor of y and all y_i in X in the parse tree of ψ'. Hence $(x, y) \in D^{\text{tree}}$. $\square$

Example 3.3.5 on page 33 shows a concrete PCNF where D^{std} contains fewer spurious dependencies than any non-deterministically constructed variant of D^{tree}. Hence $D^{\text{std}} \subset D^{\text{tree}}$ and also $D^{\text{tree}} \subset D^{\text{triv}}$ in that example. Actually, by Lemma 3.4.3, D^{std} is *never worse* than D^{tree} for any PCNF with respect to the number of spurious dependencies. For QBF solving, D^{std} grants at least the same amount of freedom as D^{tree} and possibly more, which could improve the performance as pointed out in Example 3.3.6 on page 34. Together with arguments from Section 3.3, these observations are our motivation to apply D^{std} instead of D^{tree} in practice.

Note that D^{tree} by Definition 3.4.14 is based on given PCNFs. In fact, it is unknown whether Lemma 3.4.3 also holds even if D^{tree} is computed right from the parse tree of a non-prenex non-CNF formulae. In this case, D^{tree} and D^{std} cannot easily be compared. The problem is that CNF conversion introduces clauses and new variables if Tseitin transformation [127] is applied. Thus two variables x and y might be connected by Definition 3.4.15 in the resulting PCNF which were unrelated in the parse tree of the original formula because neither was predecessor of the other one. That is, we could have $(x, y) \in D^{\text{std}}$ but $(x, y) \notin D^{\text{tree}}$, which contradicts Lemma 3.4.3. A similar effect might occur if a CNF is obtained by applying the laws of distributivity which, different from Tseitin transformation, avoids introduction of new variables. Given all that, we are not aware of a proof of Lemma 3.4.3 under the assumption that D^{tree} is computed from quantifier structure of the parse tree of non-prenex non-CNF formula. Such proof would finally allow to prefer D^{std} instead of D^{tree} in QBF solving *in general*. However, as we observed already, D^{tree} obtained by mini-scoping is strictly worse than D^{std} due to non-deterministic construction and spurious dependencies.

3.4.5 Dependency Schemes in Practice

Classical approaches of QBF solving like QDPLL [30] were implicitly introduced with respect to the trivial dependency scheme D^{triv}, even if at the time of publication the notion of dependency schemes was still unknown. The left-to-right ordering of variables in the quantifier prefix of a PCNF corresponds exactly to D^{triv}. Natural exceptions are approaches of search-based QBF solving for non-PCNFs which we sketched in Section 2.3 on page 16, or QDPLL where tree-like dependency information is extracted from the parse tree of a non-PCNF [62] or from quantifier trees obtained by mini-scoping [12]. In Chapter 5, we present a novel view on search-based QBF solving which extends QDPLL from D^{triv} to arbitrary dependency schemes. This directly enables the potential benefits from dependency schemes more refined than D^{triv}. Note that D^{triv} is the worst of all dependency schemes by Definition 3.4.13 and Proposition 3.4.5. Modifications of basic QDPLL are not required and hence dependency schemes can be integrated seamlessly into search-based QBF solving. From the tractable dependency schemes presented in Section 3.4.3, we prefer D^{std} since it strictly refines both D^{triv} and D^{tree}. Different from D^{tree}, it can be constructed deterministically.

Variable Elimination

Dependency schemes can also be used to improve the performance of QBF solvers based on variable elimination. As an example, we extend universal expansion for variables from the first non-innermost quantifier block by Lemma 2.3.3 on page 20 to arbitrary universal variables. Additionally we comment on related observations made in [113].

Lemma 3.4.4 ([16, 25])**.** *The PCNF*

$$\psi := Q_1 B_1, \ldots, \forall (B_i \cup \{x\}) \exists B_{i+1} \forall B_{i+2} \exists B_{i+3} \ldots \forall B_{n-1} \exists B_n. \, \phi$$

is satisfiability-equivalent to

$$\psi' := Q_1 B_1, \ldots, \forall B_i \exists (B_{i+1} \cup B'_{i+1}) \forall B_{i+2} \exists (B_{i+3} \cup B'_{i+3}) \ldots$$
$$\ldots \forall B_{n-1} \exists (B_n \cup B'_n).\ (\phi[x] \wedge \phi'[\neg x]).$$

Sets $B'_{i+1}, B'_{i+3}, \ldots, B'_n$ consist of fresh variables obtained from duplicating $B_{i+1}, B_{i+3}, \ldots, B_n$, and in ϕ' occurrences of variables in $B_{i+1}, B_{i+3}, \ldots, B_n$ are replaced by occurrences of duplicated ones in $B'_{i+1}, B'_{i+3}, \ldots, B'_n$.

As noted with respect to Lemma 2.3.3 in Section 2.3.2, not necessarily all variables in $B_{i+1}, B_{i+3}, \ldots, B_n$ have to be duplicated. Actually, in Lemmata 2.3.3 and 3.4.4 we implicitly rely on D^{triv}. If we apply a dependency scheme D which refines D^{triv} then we might be able to find out that some variables in $B_{i+1}, B_{i+3}, \ldots, B_n$ are independent from x. This might reduce the number of duplicated variables in $B'_{i+1}, B'_{i+3}, \ldots, B'_n$ and hence also the size of the copied part ϕ'. In the best case, all variables $B_{i+1}, B_{i+3}, \ldots, B_n$ are independent and x can be eliminated without increasing the size of the formula at all. This was pointed out by Example 3.3.7 on page 36. However, in general all existential variables in the transitive closure $D^*(x)$ of a dependency scheme D have to be duplicated.

Proposition 3.4.6 (related to Remarks 1 and 2 in [113])**.** *Given a PCNF ψ over variables V and some $x \in V$ where $q(x) = \forall$. Let D be a dependency scheme for ψ. Expanding variable x by Lemma 3.4.4 produces a satisfiability-equivalent formula ψ' if all existential variables in $D^*(x)$ are duplicated and respective parts of ψ are copied.*

Example 3.4.9 (taken from Remark 2 in [113])**.** Given the satisfiable PCNF

$$\psi := \forall x \exists u \forall y \exists v.\ (x \vee y \vee \neg v) \wedge (\neg x \vee \neg y \vee \neg v) \wedge (u \vee y \vee v) \wedge$$
$$(\neg u \vee y \vee \neg v) \wedge (\neg u \vee \neg y \vee v) \wedge (u \vee \neg y \vee \neg v).$$

The *triangle dependency scheme* $D^{\triangle}$ was introduced in [113] and refines D^{std}. Similarly to D^{std}, it is based on X-paths by Definition 3.4.15 but it takes the polarities of literals into account. For the given PCNF ψ, we have $D^{\triangle} := \{(x, u), (u, y), (y, v)\}$. None of the dependencies in $D^{\triangle}$ is spurious in this case, that is $D^{\triangle} = D^{\text{opt}}$. Assume that we want to expand x by Lemma 3.4.4 with respect to $D^{\triangle}$ and not $D^{\triangle *}$, which contradicts the requirement in Proposition 3.4.6. Hence we duplicate variables and clauses with respect to $D^{\triangle}(x) = \{u\}$ only. However, the expanded formula is unsatisfiable. Instead, as noted in Remark 2 in [113], we must duplicate variables and clauses with respect to the transitive closure $D^{\triangle *}(x) = \{u, v\}$.

Example 3.4.9 shows that the requirement stated in Proposition 3.4.6 cannot be relaxed *in general*. That might seem surprising given the fact that *all* transitive dependencies which are added to obtain the transitive closure of a dependency scheme D are actually spurious. This follows right from Definition 3.4.7: if $(x, y) \in D^*$ but $(x, y) \notin D$ then the dependency (x, y) is spurious because otherwise relation D is not a dependency scheme. Still, if only variables $D(x)$ instead of $D^*(x)$ are duplicated then expansion might also copy clauses which contain variables in $D^*(x)$, as in Example 3.4.9. In order to preserve satisfiability, these variables have to be duplicated as well.

According to Remark 1 in [113], it suffices to consider variables in D^{std} instead of $D^{\text{std}*}$ for expansion. This observation does not conflict with Proposition 3.4.6 since D^{std} is transitive by Lemma 3.4.2, that is $D^{\text{std}} = D^{\text{std}*}$. To the best of our knowledge, it is unknown whether, for a specific dependency scheme D, always the full set $D^*(x)$ has to be duplicated. Possibly, a more refined analysis with respect to the given PCNF could find out which variables can safely be excluded.

Dependency Schemes for Non-PCNFs – A Challenge

The theory of independence between variables and dependency schemes by Definitions 3.4.4, 3.4.5 and 3.4.7 takes only QBFs in PCNF into account. In the previous section we argued that dependency schemes which strictly refine D^{triv} could improve QBF solvers both based on search and on variable elimination. Solvers for non-PCNFs might also profit from dependency schemes which refine D^{tree} in terms of quantifier structure present in the parse tree of a non-PCNF. We observed that D^{std} refines D^{tree} on PCNFs by Lemma 3.4.3. However, as noted above, that result cannot easily be obtained with respect to non-PCNFs.

A possible, theoretical framework of dependency schemes for non-PCNFs does not directly allow for practical applications. Optimal dependency schemes for non-PCNFs are intractable as well like they are for PCNFs by Proposition 3.4.3. D^{tree}, which follows right from the syntactic structure of non-PCNFs, can be used directly by QBF solvers which do not rely on PCNF like [46, 62, 66], for example. Apart from that, it is unknown how to compute tractable dependency schemes like D^{std} and related refinements if the formula is not in clausal form. In that respect, the approach of [81], where universal variables are expanded in QBFs in prenex NNF rather than PCNF, extends ideas from [16, 25]. The set of variables to be duplicated is computed by an algorithm presented in [25] which also allows to compute D^{std}. However, only variables from the innermost universal quantifier block are considered in [81], like in [16].

3.5 Summary

Given a PCNF ψ, search-based solvers must assign variables in “left-to-right” ordering with respect to the quantifier prefix of ψ. Violations of this requirement might cause the solver to produce incorrect results. However, there are situations where the assignment ordering can safely be relaxed, which increases the freedom of QBF solvers. Variable independence is the theoretical foundation of possible relaxations of prefix orderings.

A dependency scheme D for a PCNF ψ is a binary relation over the set of variables of a PCNF which expresses independence between variables. If $(x, y) \notin D$ then x and y are independent and can be assigned in arbitrary order by a search-based QBF solver. This allows a solver to construct additional assignment trees where the strict ordering of paths based on the quantifier prefix of PCNFs is relaxed. Solvers based on variable elimination might profit from independence by reducing the costs of eliminating a variable. Since computing optimal dependency schemes is infeasible in practice, we have to make a tradeoff between tractability and optimality. A dependency scheme D is not optimal for a PCNF if it contains spurious dependencies. A dependency $(x, y) \in D$ is spurious if x and y are actually independent. Comparing dependency schemes with respect to subset relationship gives rise to a hierarchy in terms of a partial order.

We considered three tractable dependency schemes which can be constructed by analyzing the syntactic structure of a given PCNF. The trivial dependency scheme D^{triv} corresponds to the prefix ordering of variables in a PCNF. It is the worst of all possible dependency schemes in the sense that it is most restrictive for QBF solvers. The dependency scheme D^{tree} can be obtained from tree-like quantifier structure given by mini-scoping. Mini-scoping suffers from non-deterministic applications of quantifier shifting rules. We pointed out that the standard dependency scheme D^{std} refines both D^{triv} and D^{std} and can be computed deterministically.

Chapter 4

The Standard Dependency Scheme

4.1 Introduction

In the previous chapter, we considered dependency schemes as a means of dependency analysis for PCNFs. We observed that the computation of optimal dependency schemes is infeasible and cannot be applied in practice. Instead, we have to confine our interest to tractable dependency schemes. Whereas until now our focus was on theoretical aspects, we move on to practical applications in this chapter. Tractable dependency schemes by Definition 3.4.12 on page 47 can be computed in polynomial time, but that does not necessarily imply feasibility in practice. For example, the run time of a quadratic algorithm to compute the standard dependency scheme D^{std} can still be too large.

Example 4.1.1. Given PCNF $\psi := Q_1B_1 \dots Q_nB_n.\ \phi$ over variables V where $|V|$ and $|\phi|$ are the numbers of variables and clauses, respectively. To compute D^{std} for ψ by Definition 3.4.17 on page 49, we have to search for X-paths. If we do this explicitly, then in the worst case $\mathcal{O}(|\phi|)$ clauses must be checked to find out whether $(x, y) \in D^{\text{std}}$ for variables $x, y \in V$. This has to be carried out for *all* variables to obtain D^{std}. Thus it takes $\mathcal{O}(|V| \cdot |\phi|)$ time in the worst case to compute the full dependency scheme D^{std} for ψ.

Although the worst case run time of the algorithm sketched in Example 4.1.1 is polynomial, the *observed* run time might be too large for practical applications as shown in Section 4.6 below. For large values of $|V|$ and $|\psi|$, which are not uncommon in real-world instances,[1] the cost of computing

[1]In the benchmark set used for QBFEVAL'10 [100], formulae have 23546 variables and 53857 clauses on average. The largest formula `c1_BMC_p1_k2048-shuffled.qdimacs`, which encodes an instance of bounded model checking (BMC) [19], has 2202779 variables and 5534890 clauses.

D^{std} might outweigh the benefits when combining QDPLL with D^{std}, for example. This holds for dependency schemes in general.

In order to overcome this problem, we address the question of how to compute and represent the standard dependency scheme D^{std} efficiently. We focus on D^{std} because it refines both D^{triv} and D^{tree} and can be computed deterministically. First, in Section 4.2 we introduce a general representation of dependency schemes as *directed acyclic graphs (DAGs)* which is not limited to D^{std}. Such trivial DAGs follow right from a given dependency scheme D if dependencies $(x, y) \in D$ are interpreted as edges in the DAG. Further, we obtain compressed DAGs with respect to particular equivalence relations over variables. These compressed DAGs can represent any arbitrary dependency scheme. Relying on certain theoretical properties of D^{std} to be observed in Section 4.3, we point out how to compute such compressed DAGs for D^{std} efficiently. Thereby, we improve upon direct applications of Definition 3.4.17 as demonstrated in Example 4.1.1. In Section 4.6 we show by experimental analysis that computation time is negligible in practice while, at the same time, the defined equivalence relations allow for compact representations. Informally, we consider a representation of a dependency scheme to be compact if it makes use of equivalence classes rather than individual variables. Our presented algorithms are tailored towards D^{std}. However, they might also give insights into novel approaches to compute dependency schemes which refine D^{std} like the triangle or quadrangle dependency schemes [51, 113].

As described in Chapter 5, our goal is to combine graph-based representations of dependency schemes with search-based QBF solvers relying on QDPLL. Despite the compactness of DAG-based representations to be introduced below, additional efforts have to be made to apply them efficiently in the context of QDPLL. We show how the compact DAGs from this chapter can be augmented for that purpose. In general, these DAGs can be combined with any kind of QBF solver.

4.2 General Dependency Graphs

In this section, we develop a general graph representation suitable for any arbitrary dependency scheme. We begin with a trivial directed acyclic graph (DAG) which directly corresponds to the dependency scheme. This DAG is revised step by step, until we finally obtain a graph over equivalence classes of variables. We illustrate the definitions by Example 4.2.1 below.

Definition 4.2.1. Given a dependency scheme D for a PCNF ψ, $D^{-1} := \{(y, x) \in V \times V \mid (x, y) \in D\}$ is the *inverse* of D.

Definition 4.2.2. Given a dependency scheme D for a PCNF ψ over variables V, the *explicit dependency graph* for D is the DAG $G(D)$ with vertices V and directed edges $E := \{(x, y) \mid (x, y) \in D\}$.

In Definition 4.2.2, vertices in $G(D)$ are variables and edges are dependencies by D, which is explicitly represented by $G(D)$. We might find out that, for two variables x and x', the same set of variables depends on x and x', that is $D(x) = D(x')$. In this case, the vertices of x and x' have the same set of "outgoing" edges in $G(D)$. With respect to that, there is no need to distinguish between x and x'. Based on this observation, we define an equivalence relation on variables as follows.

Definition 4.2.3. Given a dependency scheme D for a PCNF ψ over variables V, two variables x and x' are *equivalent with respect to* D, written as $x \approx_{D,\downarrow} x'$, if and only if $D(x) = D(x')$. If D is arbitrary or clear from the context, then we write $x \approx_{\downarrow} x'$.

The symbol $\downarrow$ which appears as subscript in $\approx_{D,\downarrow}$ and $\approx_{\downarrow}$ indicates that the equivalence relation by Definition 4.2.3 merges vertices in $G(D)$ which have the same set of *outgoing* edges. Similar to the observation above, we might find out that two variables y and y' depend on the same set of variables, that is $D^{-1}(y) = D^{-1}(y')$. Consequently, the vertices of y and y' have the same set of "incoming" edges in $G(D)$. Like before, there is no need to distinguish between y and y' with respect to that.

Definition 4.2.4. Given a dependency scheme D for a PCNF ψ over variables V, two variables y and y' are *equivalent with respect to* D^{-1}, written as $y \approx_{D,\uparrow} y'$, if and only if $D^{-1}(y) = D^{-1}(y')$. If D is arbitrary or clear from the context, then we write $y \approx_{\uparrow} y'$.

The symbol $\uparrow$ which appears as subscript in $\approx_{D,\uparrow}$ and $\approx_{\uparrow}$ indicates that the equivalence relation by Definition 4.2.4 merges vertices in $G(D)$ which have the same set of *incoming* edges.

Definition 4.2.5. Given a PCNF ψ over variables V, a dependency scheme D and the equivalence relation $\approx_{\downarrow}$ on V, the set

$$V/_{\approx_{\downarrow}} := \{S \subseteq V \mid S = \{x \in V \mid \exists x' \in V : x \approx_{\downarrow} x'\}\}$$

is the set of *equivalence classes* or the *partition* of V induced by $\approx_{\downarrow}$. Given $\approx_{\uparrow}$ by Definition 4.2.4, $V/_{\approx_{\uparrow}}$ is defined analogously.

Definition 4.2.6. Given a PCNF ψ over variables V, a dependency scheme D for ψ and the equivalence relations $\approx_{\downarrow}$ and $\approx_{\uparrow}$ by Definitions 4.2.3 and 4.2.4. For $x \in V$, $[x]_{\downarrow}$ and $[x]_{\uparrow}$ is the *equivalence class* of x with respect to the partitions $V/_{\approx_{\downarrow}}$ and $V/_{\approx_{\uparrow}}$ of V, respectively. We also refer to $[x]_{\downarrow}$ and $[x]_{\uparrow}$ as *representatives* of the respective classes of x.

The equivalence relations $\approx_{\downarrow}$ and $\approx_{\uparrow}$ also induce partitions on the set of vertices in the explicit dependency graph $G(D)$ by Definition 4.2.2. We obtain a graph over equivalence classes which is potentially more compact.

Definition 4.2.7. Given a dependency scheme D for a PCNF ψ over variables V, the *compressed dependency graph* for D is the DAG $G_{\approx}(D)$ with vertices $V/_{\approx_{\downarrow}} \cup V/_{\approx_{\uparrow}}$ and directed edges $E := \{([x]_{\downarrow}, [y]_{\uparrow}) \mid (x, y) \in D\}$.

Edges in graph $G_{\approx}(D)$ connect classes of variables instead of individual ones. Thus we expect a smaller number of edges altogether compared to explicit dependency graphs by Definition 4.2.2. Finally, we add auxiliary edges to $G_{\approx}(D)$ based on subset relationships with respect to D^{-1} and $\approx_{\uparrow}$.

Definition 4.2.8. Given a dependency scheme D for a PCNF ψ over variables V, the *augmented compressed dependency graph* for D is the DAG $G_{\approx,\subseteq}(D)$ with vertices $V/_{\approx_{\downarrow}} \cup V/_{\approx_{\uparrow}}$ and directed edges $E := E_d \cup E_s$, where $E_d := \{([x]_{\downarrow}, [y]_{\uparrow}) \mid (x, y) \in D\}$ is the set of *dependency edges* and $E_s := \{([x]_{\uparrow}, [x']_{\uparrow}) \mid D^{-1}(x) \subseteq D^{-1}(x')\}$ is the set of *subset edges.*

By Definition 4.2.8, an edge $([x]_{\uparrow}, [x']_{\uparrow})$ in E_s connects the classes $[x]_{\uparrow}$ and $[x']_{\uparrow}$ if $[x']_{\uparrow}$ has at least the same and possibly more incoming dependency edges than $[x]_{\uparrow}$. This set E_s of auxiliary edges does not add additional information related to dependencies. Due to edges in E_s, we even expect the graph $G_{\approx,\subseteq}(D)$ to be larger than $G_{\approx}(D)$ by Definition 4.2.7. Despite of that overhead, in Chapter 5 we point out that subset edges E_s are important to efficiently integrate dependency schemes into QDPLL-based QBF solvers. Further, in practice it is actually not necessary to include transitive edges in E_s (see also Section 4.5.2 for related comments).

In general, $G_{\approx,\subseteq}(D)$ can be constructed for any arbitrary dependency scheme D. In a trivial approach, we could start with the explicit dependency graph $G(D)$ by Definition 4.2.2 and then obtain $G_{\approx}(D)$ and $G_{\approx,\subseteq}(D)$ by merging variables into equivalence classes and finally adding subset edges. However, we want to avoid the construction of the explicit graph $G(D)$ at all because it might be large for PCNFs with large numbers of variables and clauses. In Sections 4.4 and 4.5 below, we argue that for $D := D^{\mathrm{std}}$ a variant of the graph $G_{\approx,\subseteq}(D)$ is compact and can be constructed efficiently. This allows QDPLL to benefit from D^{std}, which is the topic of Chapter 5.

Example 4.2.1. Assume that we are given a PCNF with prefix

$$\forall a \exists x_1,x_2,x_3 \forall y_1,y_2,y_3 \exists z_1,z_2$$

and some dependency scheme D. On the left in Figure 4.1 the explicit dependency graph $G(D)$ by Definition 4.2.2 is shown. We list the members of equivalence classes in brackets '[' and ']'. We have $D(x_1) = D(x_2)$ and hence $x_1 \approx_{D,\downarrow} x_2$ or simply $x_1 \approx_{\downarrow} x_2$. Trivially, also $D^{-1}(x_1) = D^{-1}(x_2) = \emptyset$ and hence $x_1 \approx_{\uparrow} x_2$. Further $D^{-1}(y_1) = D^{-1}(y_2) = \{x_1, x_2\}$ and hence $y_1 \approx_{\uparrow} y_2$. The augmented compressed dependency graph $G_{\approx,\subseteq}(D)$ by Definition 4.2.8 is shown on the right in Figure 4.1. We have $D^{-1}(x_3) = \{a\}$, $D^{-1}(z_2) = \{a, y_1, y_2, y_3\}$, $D^{-1}(x_3) \subseteq D^{-1}(z_2)$ and hence we add a subset edge (dashed)

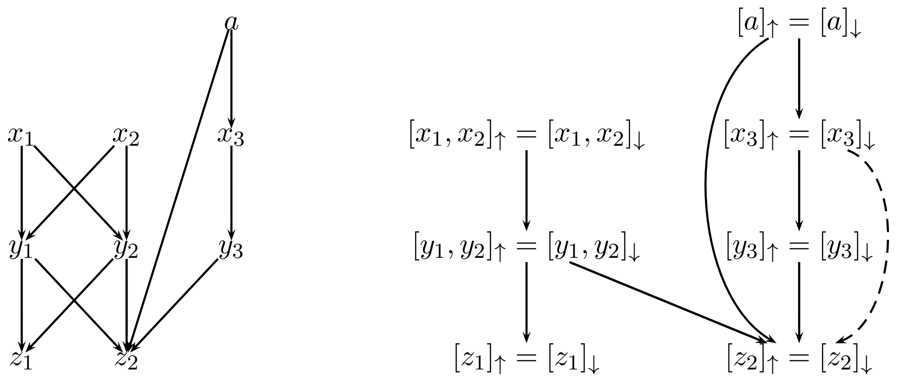

a
x_1
x_2
x_3
y_1
y_2
y_3
z_1
z_2
$[a]_\uparrow = [a]_\downarrow$
$[x_1, x_2]_\uparrow = [x_1, x_2]_\downarrow$
$[x_3]_\uparrow = [x_3]_\downarrow$
$[y_1, y_2]_\uparrow = [y_1, y_2]_\downarrow$
$[y_3]_\uparrow = [y_3]_\downarrow$
$[z_1]_\uparrow = [z_1]_\downarrow$
$[z_2]_\uparrow = [z_2]_\downarrow$

Figure 4.1: The explicit dependency graph $G(D)$ by Definition 4.2.2 (left) and the corresponding augmented compressed dependency graph $G_{\approx,\subseteq}(D)$ by Definition 4.2.8 (right) for a presumed PCNF with prefix $\forall a \exists x_1,x_2,x_3 \forall y_1,y_2,y_3 \exists z_1,z_2$ and dependency scheme D from Example 4.2.1 (right). The dashed edge from a to z_3 represents a subset edge.

from $[x_3]_\uparrow$ to $[z_2]_\uparrow$. Note that reflexive subset edges and trivial ones of the form $([x]_\uparrow, [x']_\uparrow)$ where $D^{-1}(x) = \emptyset$ are omitted in Figure 4.1.

4.3 Theoretical Properties

In this section, we analyze theoretical properties of D^{std} and X-paths by Definition 3.4.17. We finally obtain an algorithm and data structures which allow to compute D^{std} efficiently in practice. Different from Example 4.1.1, the idea is to avoid searching for X-paths explicitly for each pair (x, y) of variables $x, y \in V$. For example, we might be able to justify that $(x, y) \in D^{\text{std}}$ and $(x', y') \in D^{\text{std}}$ by one and the same X-path.

Example 4.3.1. Given PCNF $\psi := \forall x,x' \exists y,y',y_1,\ldots,y_n.\ (x \lor x' \lor y_1) \land (y_1 \lor y_2) \land \ldots \land (y_{n-1} \lor y_n) \land (y_n \lor y \lor y')$. The full set of clauses $(x \lor x' \lor y_1) \land \ldots \land (y_n \lor y \lor y')$ is an X-path between x and y for $X := \{y, y', y_1, \ldots, y_n\}$. The same X-path also connects x' and y'.

We pick up ideas from [82, 83]. In order to avoid searching for X-paths explicitly, first we determine connections between existential variables in a given PCNF. Thereby, we focus on those particular connections which are relevant to establish X-paths as needed in Definition 3.4.17. That precomputed information is re-used when searching for X-paths. Relying on our theoretical analysis, we construct an *approximation* of the augmented compressed dependency graph $G_{\approx,\subseteq}(D^{\text{std}})$ in Section 4.5 on page 70.

It is important to note that the term "approximation" refers only to the structure of the graph and not to the set of dependencies that is represented thereby. That is, the approximation of $G_{\approx,\subseteq}(D^{\text{std}})$ we introduce in Section 4.5 below is precise with respect to D^{std}. There is a dependency (x, y) in the approximation of the graph if and only if there is a dependency (x, y) with respect to D^{std} by Definition 3.4.17. However, the approximation of $G_{\approx,\subseteq}(D^{\text{std}})$ might not be as compact as the original graph $G_{\approx,\subseteq}(D^{\text{std}})$ by Definition 4.2.8. The reason for less compaction is that equivalences between variables by Definitions 4.2.3 and 4.2.4 might not be considered to the full extent. Concerning practical applications, it is not necessary to start with $G(D^{\text{std}})$ and then merge classes explicitly, as in the trivial approach outlined above. Instead, as we show, the approximation of $G_{\approx,\subseteq}(D^{\text{std}})$ can largely be computed right away from a given PCNF. Although we focus on compact representations of D^{std} in this chapter, our approaches related to the computation of approximations of $G_{\approx,\subseteq}(D^{\text{std}})$ might also be extended to other dependency schemes which refine D^{std}.

In the remaining parts of this section, we apply the following notion of transitive relations.

Definition 4.3.1 ([1])**.** Let $R \subseteq V \times V$ be a binary relation on some set of variables V. The *reflexive and transitive closure* of R is the smallest reflexive and transitive $R' \subseteq V \times V$ such that $R \subseteq R'$. The *reflexive and transitive reduction* of R is the smallest $R' \subseteq V \times V$ such that R and R' have the same reflexive and transitive closure.

Definition 4.3.1 differs from transitivity of dependency schemes by Definition 3.4.9 on page 45. The former is related to arbitrary binary relations and is well-known. In this section we refer to transitive closure by Definition 4.3.1, unless stated otherwise.

We consider closed PCNFs where, for all clauses $C_i := (l_1 \vee \ldots \vee l_{k_i})$, $l_j < l_{j'}$ for $1 \leq j < j' \leq k_i$ and $q(v(l_{k_i})) = \exists$. That is, literals in clauses are sorted ascendingly with respect to prefix ordering and the largest literal is existential. Further, we assume that clauses neither contain multiple nor complementary literals of one and the same variable.

Recall Definition 3.4.17 on page 49 where the standard dependency scheme D^{std} was introduced. By setting $i := \delta(x) + 1$ and $X := V_{\exists,i}$, universal variables as well as variables which are from the block of x or from any smaller block are excluded from X. Hence these variables cannot participate in any X-path.

Example 4.3.2. For the PCNF in Figure 4.2, we have $i = \delta(a_1) = 1$, $e_{13} \in D^{\text{std}}(a_1)$ by clauses $(a_1 \vee a_6 \vee e_8 \vee e_{14})$ and $(e_3 \vee e_8 \vee e_{13})$, and $X = V_{\exists,i+1} = \{e_3, e_4, e_5, e_8, e_9, e_{10}, e_{13}, e_{14}, e_{15}\}$. Note that variable e_8 connects the two clauses and $e_8 \in V_{\exists,i+1}$ since $i \leq \delta(e_8)$. However, $e_{14} \notin D^{\text{std}}(a_7)$ since a_7 and e_{10} occur only in clause $(e_4 \vee a_7 \vee e_{10})$. Variable e_4 cannot be used to connect a_7 and e_{14} by clause $(e_4 \vee e_{13} \vee e_{14})$ because $\delta(e_4) < \delta(a_7)$.

i	$q(B_i)$	B_i	$(a_2 \vee e_5 \vee e_9)$
1	$\forall$	a_1, a_2	$(e_5 \vee e_9 \vee e_{15})$
2	$\exists$	e_3, e_4, e_5	$(e_3 \vee e_8 \vee e_{13})$
3	$\forall$	a_6, a_7	$(e_4 \vee a_7 \vee e_{10})$
4	$\exists$	e_8, e_9, e_{10}	$(e_4 \vee e_{13} \vee e_{14})$
5	$\forall$	a_{11}, a_{12}	$(a_1 \vee a_6 \vee e_8 \vee e_{14})$
6	$\exists$	e_{13}, e_{14}, e_{15}	$(a_{11} \vee a_{12} \vee e_{13})$

Figure 4.2: PCNF example used in Sections 4.3 and 4.5.3. The table shows the levels, quantifiers and variables for each quantifier block in the first three columns and clauses in the last column. Variable names are prefixed with "e" and "a" to indicate their types $\exists$ and $\forall$, respectively.

Building on related work preceding D^{std} by Definition 3.4.17 [16, 25, 113], we first introduce a connection relation between variables which is aware of the levels of quantifier blocks.

Definition 4.3.2. For $x, y \in V$, x is *connected* to y with respect to block B_i, written as $x \rightarrow_i y$, if and only if $y \in V_{\exists,i}$ and there is a clause C such that $x \in C$ and $y \in C$. Relation $\rightarrow_i^*$ is the reflexive and transitive closure of $\rightarrow_i$.

Relation $\rightarrow_i^*$ is defined with respect to some block B_i. If $x \rightarrow_i^* y$ then x is connected to y over existential variables from blocks larger than or equal to B_i only. As pointed out in Theorem 4.3.2 below, this relation can be applied to compute D^{std}. There is a close correspondence between X-paths as used in Definition 3.4.17 and relation $\rightarrow_i^*$.

Corollary 4.3.1. *Given $x, y \in V$, if $x \rightarrow_i^* y$ then there is an X-path between x and y for $X = V_{\exists,i}$.*

Note that, due to Definition 4.3.2, the converse of Corollary 4.3.1 does not hold in general. For example, if there is an X-path between $x \in V_\exists$ and $y \in V_\forall$ then $x \not\rightarrow_i^* y$ for all i. A weaker variant can be stated as follows.

Corollary 4.3.2. *Given $x \in V$ and $y \in V_\exists$, if there is an X-path between x and y for $X = V_{\exists,i}$ and $i \leq min(\delta(x), \delta(y))$ then $x \rightarrow_i^* y$.*

We observe that connections with respect to a block B_j are preserved for any smaller block B_i.

Corollary 4.3.3. *Given $x, y \in V$ and $i \leq j$, if $x \rightarrow_j^* y$ then also $x \rightarrow_i^* y$.*

For proper values of i in Definition 4.3.2, connections between existential variables are symmetric because X-paths resulting from Corollary 4.3.1 can trivially be reversed.

Lemma 4.3.1. *Given $x, y \in V_\exists$ and $i \leq min(\delta(x), \delta(y))$, if $x \rightarrow_i^* y$ then $y \rightarrow_i^* x$.*

Example 4.3.3. For the PCNF in Figure 4.2, $e_3 \rightarrow_4 e_8$ by clause $(e_3 \vee e_8 \vee e_{13})$ but $e_3 \not\rightarrow_5 e_8$, $e_8 \rightarrow_6 e_{14}$ by clause $(a_1 \vee a_6 \vee e_8 \vee e_{14})$ and by Corollary 4.3.3 also $e_8 \rightarrow_1 e_{14}$. Further $e_3 \rightarrow_2^* e_{14}$ by clauses $(e_3 \vee e_8 \vee e_{13})$ and $(e_4 \vee e_{13} \vee e_{14})$ and by Lemma 4.3.1 also $e_{14} \rightarrow_2^* e_3$.

As a first step towards efficient computation and compact representation of D^{std} we want to take advantage of situations where two variables can be regarded as equivalent with respect to connections by Definition 4.3.2.

Definition 4.3.3. For $x, y \in V$, x is *equivalent* to y *with respect to connection relation* $\rightarrow_i^*$, written as $x \approx y$, if and only if either (1) $x = y$ or (2) $q(x) = q(y) = \exists$, $\delta(x) = \delta(y) = i$ and $x \rightarrow_i^* y$.

Variables x and y are equivalent by Definition 4.3.3 if $x = y$ or both are from the same existential block B_i and are connected by existential variables larger than or equal to B_i. Note that relation $\approx$ is different from the equivalence relations $\approx_\downarrow$ and $\approx_\uparrow$ introduced previously in Definitions 4.2.3 and 4.2.4. However, as pointed out in Section 4.5.1 below, $\approx$ has certain properties which allow to obtain an approximation of $\approx_\uparrow$ for existential variables in order to construct an approximation of graph $G_{\approx,\subseteq}(D^{\text{std}})$.

Theorem 4.3.1. *Given a PCNF ψ over variables V, relation $\approx$ by Definition 4.3.3 is an equivalence relation.*

Proof. Reflexivity is trivial since $x \approx x$ for $x \in V$ by Definition 4.3.3. If *not* $q(x) = q(y) = \exists$ then by Definition 4.3.3 $x \approx y$ if and only if $x = y$. Since $=$ is an equivalence relation, symmetry and transitivity of $\approx$ follow immediately.

Otherwise, assume $q(x) = q(y) = \exists$. If $x \approx y$ and $x = y$, then also $y \approx x$ by Definition 4.3.3. If $x \approx y$ and $x \neq y$ then by Definition 4.3.2 and Definition 4.3.3 $\delta(x) = \delta(y)$ and $x \rightarrow_i^* y$ for $i = \delta(x) = \delta(y)$. Then by Lemma 4.3.1 also $y \rightarrow_i^* x$ and hence $y \approx x$. Therefore, $\approx$ is symmetric.

To show transitivity, assume $x \approx y'$ and $y' \approx y$ for $y' \in V$. Then more precisely $y' \in V_\exists$ (because otherwise $x \not\approx y'$ and $y' \not\approx y$) and by Definition 4.3.3 also $x \rightarrow_i^* y'$, $y' \rightarrow_i^* y$ for $i = \delta(x) = \delta(y') = \delta(y)$ and $q(x) = q(y') = q(y)$. By $x \rightarrow_i^* y'$, $y' \rightarrow_i^* y$ and transitivity of $\rightarrow_i^*$, also $x \rightarrow_i^* y$, hence $x \approx y$. $\square$

Definition 4.3.4. Given a PCNF ψ over variables V, the *partition* $V/_\approx$ of V induced by $\approx$ is obtained similarly to Definition 4.2.5. Given $x \in V$, $[x]$ is the *equivalence class* of x with respect to partition $V/_\approx$.

Example 4.3.4. For the PCNF in Figure 4.2, $e_3 \approx e_4$ since $q(e_3) = q(e_4) = \exists$, $\delta(e_3) = \delta(e_4) = 2$ and $e_3 \rightarrow_2^* e_4$ by $e_3 \rightarrow_2 e_8 \rightarrow_2 e_{14} \rightarrow_2 e_4$. Also $e_{13} \approx e_{14}$

since $e_{13} \rightarrow_6 e_{14}$ but $e_5 \not\approx e_4$ because $e_5 \not\rightarrow_2^* e_4$. Trivially, $a_{11} \approx a_{11}$ and $e_3 \not\approx e_{14}$.

Relation $\rightarrow_i^*$ is compatible with $\approx$. If two variables are connected then so are all members of their respective classes and vice versa as stated in the following lemma.

Lemma 4.3.2. *Given $x, y \in V$ and $i \leq min(\delta(x), \delta(y))$. Then $x \rightarrow_i^* y$ if and only if $x' \rightarrow_i^* y'$ for all $x' \in [x], y' \in [y]$.*

Proof. The proof works regardless of the types of x and y by Definition 4.3.2 (reflexivity of $\rightarrow_i^*$), Corollary 4.3.3 and Definition 4.3.3. Trivial cases arise for $V_\forall$. Assume $x \rightarrow_i^* y$ for $x, y \in V$ and $i \leq min(\delta(x), \delta(y))$. Then for $x' \in [x], y' \in [y]$, $x' \rightarrow_i^* x$ and $y \rightarrow_i^* y'$ by Corollary 4.3.3 and Definition 4.3.3. Since $x' \rightarrow_i^* x$, $x \rightarrow_i^* y$ (by assumption), $y \rightarrow_i^* y'$, also $x' \rightarrow_i^* y'$ by transitivity of $\rightarrow_i^*$. The other direction can be shown similarly by Lemma 4.3.1. $\square$

As in Definition 4.2.6, we regard $[x]$ as an arbitrary member of the equivalence class. We can write, for example, $[x] \rightarrow_i^* [y]$ by Lemma 4.3.2. This notation denotes connections between classes. Note that Lemma 4.3.2 would not hold for arbitrary values of i. For example, if $\delta(x) < i$ then $x \not\rightarrow_i^* x'$ for $x' \in [x]$, which contradicts Definition 4.3.3. The following variant of Lemma 4.3.2 does not refer to $[x]$ and holds for arbitrary values of i.

Lemma 4.3.3. *Let $x, y \in V$ with $\delta(x) \leq \delta(y)$. Then $x \rightarrow_i^* y$ if and only if $x \rightarrow_i^* y'$ for all $y' \in [y]$.*

Example 4.3.5. For the PCNF in Figure 4.2, $e_3 \approx e_4$, $e_{10} \approx e_{10}$, where $[e_{10}]$ is a singleton class, and $e_4 \rightarrow_2^* e_{10}$ because $e_4 \rightarrow_2 e_{10}$. By Lemma 4.3.2, also $e_3 \rightarrow_2^* e_{10}$ because $e_3 \rightarrow_2 e_8 \rightarrow_2 e_{14} \rightarrow_2 e_4 \rightarrow_2 e_{10}$.

Apart from considering equivalence classes in $\rightarrow_i^*$ by Lemma 4.3.2, the following relation allows to share information about connections, which is pointed out in Section 4.4.1 below.

Definition 4.3.5. Relation $\rightsquigarrow^*$ denotes the *directed connection relation.* Given $x \in V$ and $y \in V_\exists$, $[x] \rightsquigarrow^* [y]$ if and only if $\delta(x) \leq \delta(y)$ and $x \rightarrow_i^* y$ for $i = \delta(x)$. The reflexive and transitive reduction of $\rightsquigarrow^*$ is denoted by $\rightsquigarrow$.

Corollary 4.3.4. *Given $x, y \in V$, if $[x] \rightsquigarrow^* [y]$ then either $[x] = [y]$ or $\delta(x) < \delta(y)$.*

Relation $\rightsquigarrow^*$ is defined on equivalence classes with respect to $\approx$ only and respects the ordering of quantifier blocks in the prefix. If $[x] \rightsquigarrow^* [y]$ then variables smaller than x are excluded in the connection between x and y. By Corollary 4.3.4, if $[x] \rightsquigarrow^* [y]$ then either x and y are in the same class or in different classes but also from different blocks. We now prove that the definitions introduced above can be used to compute D^{std}.

Theorem 4.3.2. *Given PCNF ψ over variables V, D^{std} for ψ can be computed as follows:*

$$\begin{aligned}
D^{\text{std}} &= \{(x,y) \in V_\times \mid i := \delta(x)+1, \exists w \in V_{\exists,i} : && (4.1)\\
&\qquad\qquad x \to_i^* w \text{ and } y \to_i^* w\}\\
&= \{(x,y) \in V_\times \mid i := \delta(x)+1, \exists w \in V_{\exists,i} : && (4.2)\\
&\qquad\qquad x \to_i^* [w] \text{ and } [y] \to_i^* [w]\}\\
&= \{(x,y) \in V_\times \mid i := \delta(x)+1, \exists w \in V_{\exists,i} : && (4.3)\\
&\qquad\qquad x \to_i^* [w] \text{ and } [y] \leadsto^* [w]\}
\end{aligned}$$

Proof. We prove equivalence of *left (LHS)* and *right-hand sides (RHS)* of Equations 4.1 to 4.3.

In order to show that LHS(4.1) = RHS(4.1), assume by Definition 3.4.17 that there is an X-path P between x and y by clauses $C_1, \ldots, C_k$ where $q(x) \neq q(y)$. P can be split into P_1 between x, w for clauses $C_1, \ldots, C_j$ where $w \in C_j, 1 \leq j \leq k, w \in V_{\exists,i}$ and P_2 between w, y by clauses $C_j, \ldots, C_k$. By P_1 and Corollary 4.3.2 also $x \to_i^* w$ and by reversing P_2 and Corollary 4.3.2, also $y \to_i^* w$ and hence $y \in$ RHS(4.1). For the other direction, assume $x \to_i^* w$ and $y \to_i^* w$. Then by Corollary 4.3.1 there are X-paths P_1 between x, w and P_2 between y, w for $X = V_{\exists,i}$. An X-path P between x, y can be constructed by combining P_1 with reversed P_2, thus $y \in$ LHS(4.1).

In order to show that RHS(4.1) = RHS(4.2), assume $x \to_i^* w$ and $y \to_i^* w$. Since $w \in V_{\exists,i}$, also $\delta(x) \leq \delta(w)$ and hence by Lemma 4.3.3 and Definition 4.3.3 also $x \to_i^* [w]$. Further, because $i \leq \delta(y)$ and $i \leq \delta(w)$ and hence $i \leq min(\delta(y), \delta(w))$, also $[y] \to_i^* [w]$ by Lemma 4.3.2 and Definition 4.3.3. Since $x \to_i^* [w]$ and $[y] \to_i^* [w]$, also $y \in$ RHS(4.2). For the other direction, assume $x \to_i^* [w]$ and $[y] \to_i^* [w]$. Similar arguments apply to derive $x \to_i^* w$ and $y \to_i^* w$ by Lemma 4.3.2, Lemma 4.3.3 and Definition 4.3.3. Hence $y \in$ RHS(4.1).

In order to show that RHS(4.2) = RHS(4.3), assume $x \to_i^* [w]$ and $[y] \to_i^* [w]$. Since LHS(4.1) = RHS(4.1) = RHS(4.2), there is an X-path P between x, y for $X = V_{\exists,i}$ and clauses $C_1, \ldots, C_k$ where $y \in C_k$. Let l denote the largest literal in C_k. By assumptions in Section 2.1.4, clauses are sorted by prefix ordering and the largest literal is existential. That is, $v(l) \in V_\exists$ and $\delta(y) \leq \delta(l)$ (if $q(y) = \forall$ then $\delta(y) < \delta(l)$). Assume that $w = v(l)$. Then $\delta(y) \leq \delta(w)$. By $y, w \in C_k$ also $y \to_j w$ for $j = \delta(y)$ and $y \to_j^* w$ by Definition 4.3.2. By $y \to_j^* w$ and $\delta(y) \leq \delta(w)$ also $[y] \leadsto^* [w]$. Since $x \to_i^* [w]$ and $[y] \leadsto^* [w]$ also $y \in$ RHS(4.3). For the other direction, Definition 4.3.5, Corollary 4.3.3 and Lemma 4.3.2 apply. □

4.4 Towards Efficient Computation

According to Equation 4.1 in Theorem 4.3.2, D^{std} can be computed by applying relation $\rightarrow_i^*$, which corresponds to computation by X-paths in Definition 3.4.17. Equation 4.2 refers to equivalence classes by $\approx$ rather than individual variables, which is already an improvement. The step from Equation 4.2 to Equation 4.3 is the most interesting one for practical applications, yet this is not apparent from theory. Since $\rightsquigarrow^*$ is directed, it restricts the set of classes to be considered when connections of a variable are determined.

In this section, we first examine properties of $\rightsquigarrow^*$ over existential variables. We observe that its reflexive and transitive reduction $\rightsquigarrow$ can be represented efficiently as a compact tree over equivalence classes with respect to $\approx$. As pointed out in Section 4.5.2 below, this tree can be shared between all variables and is the basis for constructing approximations of the graph $G_{\approx,\subseteq}(D^{\mathrm{std}})$ by Definition 4.2.8.

4.4.1 A Tree-Shaped Representation of Connections

Since $\rightsquigarrow^*$ is directed by Definition 4.3.5 and hence also antisymmetric and acyclic, its transitive reduction $\rightsquigarrow$ is unique by Theorem 1 in [1]. The following lemma states a property of $\rightsquigarrow^*$ which accounts for the tree structure of $\rightsquigarrow$.

Lemma 4.4.1. *Given $x, y, z \in V_\exists$ where $\delta(x) \leq \delta(y)$, if $[x] \rightsquigarrow^* [z]$ and $[y] \rightsquigarrow^* [z]$ then $[x] \rightsquigarrow^* [y]$.*

Proof. Assume $[x] \rightsquigarrow^* [z]$ and $[y] \rightsquigarrow^* [z]$ where $\delta(x) \leq \delta(y)$. Then by Definition 4.3.5, $x \rightarrow_i^* z$ for $i = \delta(x)$ and $y \rightarrow_j^* z$ for $j = \delta(y)$ and $\delta(x) \leq \delta(y) \leq \delta(z)$. By Corollary 4.3.3 also $y \rightarrow_i^* z$ and by Lemma 4.3.1 $z \rightarrow_i^* y$. By Definition 4.3.2, $x \rightarrow_i^* z$ and $z \rightarrow_i^* y$, also $x \rightarrow_i^* y$ and $[x] \rightsquigarrow^* [y]$. □

If $[x] \rightsquigarrow^* [z]$ and $[y] \rightsquigarrow^* [z]$ for existential variables x, y, z and $\delta(x) \leq \delta(y)$ then $[x] \rightsquigarrow^* [z]$ is transitive by Lemma 4.4.1. Consequently, $[x]$ is not related to $[z]$ in the transitive reduction of $\rightsquigarrow^*$, that is $[x] \not\rightsquigarrow [z]$. Hence at most one class is related to another one in $\rightsquigarrow$. Therefore, $\rightsquigarrow$ can be represented as a forest, that is a collection of trees, which we introduce informally.

Definition 4.4.1. Given a PCNF ψ over variables V with m existential quantifier blocks. The *connection forest* for ψ is a collection of trees over the set $V_\exists/_\approx$ of vertices with the following properties:

1. For $x, y \in V_\exists$, there is an edge $([x], [y])$ if and only if $[x] \rightsquigarrow [y]$.
2. For $x, y \in V_\exists$ there is a path from $[x]$ to $[y]$ if and only if $[x] \rightsquigarrow^* [y]$.
3. The maximum length of a path by counting the number of edges is $m - 1$ (by Corollary 4.3.4).

4.4.2 Dependency Computation Using Connection Forests

The connection forest represents directed connections between existential variables. In general, we expect it to be compact due to equivalence classes in partition $V_\exists/_\approx$ and because it considers the transitive reduction $\rightsquigarrow$ instead of $\rightsquigarrow^*$. Our goal is to compute and represent D^{std} efficiently by making use of the connection forest of a PCNF (see also Section 4.5.3 below).

To compute $D^{\text{std}}(x)$ for an arbitrary $x \in V$, a set of proper classes of $V_\exists/_\approx$ has to be found in the connection forest which exactly denote *all* connections of variable x to larger existential variables. Exactly those variables are relevant by Definition 3.4.17 and Theorem 4.3.2. Classes in such a proper set must be connected to x and be *minimal* with respect to the prefix ordering since edges in the connection forest are directed. We require minimality since if there is a non-trivial path from $[x]$ to $[y]$ in the connection forest then $\delta(x) < \delta(y)$ by Corollary 4.3.4. Hence $[x]$ can only reach classes of larger variables. Descendants of such minimal classes in the connection forest comprise *all* connections of x by $\rightsquigarrow^*$ which are relevant for the computation of D^{std}. We identify the desired set of minimal classes by smallest ancestors in the connection forest.

Definition 4.4.2. For $y \in V_\exists$ and $i \leq \delta(y)$, the *smallest ancestor* of $[y]$ with respect to i in the connection forest is the class $h(i, [y]) := [y']$ such that $y' \in V_{\exists,i}, [y'] \rightsquigarrow^* [y]$ and there is no $y'' \in V_{\exists,i}$ with $i \leq \delta(y'') < \delta(y')$ and $[y''] \rightsquigarrow^* [y]$.

Class $h(i, [y])$ is the smallest ancestor of $[y]$ in the connection forest which is larger than or equal to block B_i. Hence $h(i, [y])$ is minimal with respect to B_i and the prefix ordering. Smallest ancestors are used to compute the set of descendants, which are classes reachable by directed edges in the connection forest, as follows.

Definition 4.4.3. For $x \in V$ and the connection forest, the set of *descendants* $H_i^*(x)$ with respect to x and block B_i is defined as follows:

1. $V_{C,i}(x) := \{[y] \mid y \in V_{\exists,i} \text{ and } x \rightarrow_i y\}$
2. $H_i(x) := \{[z] \mid [z] = h(i, [y]) \text{ for } [y] \in V_{C,i}(x)\}$
3. $H_i^*(x) := \{[y] \mid [z] \rightsquigarrow^* [y] \text{ for } [z] \in H_i(x)\}$

Note that in the definition of set $V_{C,i}(x)$, we have $x \rightarrow_i y$ if there is a clause which contains both x and y for proper values of i. Starting from clauses containing a literal of variable x, classes of existential variables larger than or equal to B_i are collected in $V_{C,i}(x)$. Set $H_i(x)$ contains all smallest ancestors with respect to B_i for classes in $V_{C,i}(x)$. Finally, $H_i^*(x)$ comprises descendants of classes in $H_i(x)$ and represents all connections of x to existential variables larger than or equal to B_i.

Corollary 4.4.1. *Given $x \in V$, if $[y] \in H_i^*(x)$ then $x \rightarrow_i^* y$.*

For some variable $x \in V$, the set $H_i^*(x)$ of descendants in the connection forest exactly characterizes relevant connections of x to existential variables which are from block B_i or any larger block. This is precisely what is needed to compute D^{std}. Informally, there is a dependency $(x, y) \in D^{\text{std}}$ if the sets of descendants of x and y are not disjoint. In this case, there exists an X-path between x and y as required by Definition 3.4.17.

Theorem 4.4.1. *For a PCNF ψ over variables V,*

$$D^{\text{std}} = \{(x, y) \in V_\times \mid H_i^*(x) \cap H_j^*(y) \neq \emptyset \textit{ for } i := \delta(x) + 1, j := \delta(y)\}.$$

Proof. We show that D^{std} by Definition 3.4.17 is equal to the definition shown above. Assume $x \in V$ and $i = \delta(x) + 1$. Direction $\supseteq$ follows right from Definition 4.4.3, Corollary 4.4.1, Corollary 4.3.3 and Theorem 4.3.2.

To show $\subseteq$, assume $(x, y) \in D^{\text{std}}$. Then there is an X-path P between x, y for $X = V_{\exists,i}$. Hence there are clauses $C_1, \ldots, C_k$ where $y, y_k \in C_k$ for some $y_k \in V_{\exists,i}$ with $\delta(y) \leq \delta(y_k)$. Such y_k always exists since by assumption the largest literal in a clause is existential (if $x \in V_\forall$ then $y \in V_\exists$ and we can choose $y_k := y$). Then P is also an X-path between x and y_k by $C_1, \ldots, C_k$ and hence $x \rightarrow_i^* y_k$ and $\delta(x) < \delta(y_k)$ since $i \leq \delta(y_k)$, $i = \delta(x) + 1$. We show that $[y_k] \in H_i^*(x) \cap H_j^*(y)$ for $i = \delta(x) + 1, j = \delta(y)$.

First, we show that $[y_k] \in H_j^*(y)$ for $j = \delta(y)$. Since $y, y_k \in C_k$ by P, also $[y_k] \in V_{C,j}(y)$. Then $[z'] \in H_j(y)$ where $[z'] = h(j, [y_k])$ for $j = \delta(y)$. By Definition 4.4.2, $[z'] \rightsquigarrow^* [y_k]$, hence $[y_k] \in H_j^*(y)$.

Second, we show that $[y_k] \in H_i^*(x)$ for $i = \delta(x) + 1$. Since P connects x and y_k, also $x, y_1 \in C_1$ for some $y_1 \in V_{\exists,i}$. Thus $[y_1] \in V_{C,i}(x)$ and $[z_1] \in H_i(x)$ for $[z_1] = h(i, [y_1])$. Then by Definition 4.4.2, $[z_1] \rightsquigarrow^* [y_1]$. P is also an X-path between y_1 and y_k by $C_1, \ldots, C_k$, hence $y_1 \rightarrow_i^* y_k$ and $\delta(x) < \delta(y_1), \delta(x) < \delta(y_k)$. Let w denote the smallest connecting variable in P between y_1, y_k: $m = \delta(w) = min(\{\delta(v) \mid v \in C_i \cap C_{i+1} \cap X, 1 \leq i < k\})$. Since m is minimal, also $y_1 \rightarrow_m^* w$, $w \rightarrow_m^* y_k$ and by Lemma 4.3.1 $w \rightarrow_m^* y_1$. By Definition 4.3.5 and since $m = \delta(w)$, also $[w] \rightsquigarrow^* [y_1]$, $[w] \rightsquigarrow^* [y_k]$. By Lemma 4.4.1, $[z_1] \rightsquigarrow^* [y_1]$ and $[w] \rightsquigarrow^* [y_1]$, also $[z_1] \rightsquigarrow^* [w]$. Then by $[z_1] \rightsquigarrow^* [w]$, $[w] \rightsquigarrow^* [y_k]$ and transitivity also $[z_1] \rightsquigarrow^* [y_k]$, hence $[y_k] \in H_i^*(x)$ because $[z_1] \in H_i(x)$.

Since $[y_k] \in H_j^*(y)$ and $[y_k] \in H_i^*(x)$, also $[y_k] \in H_i^*(x) \cap H_j^*(y)$ for $i = \delta(x) + 1$ and $j = \delta(y)$. □

In contrast to Theorem 4.3.2, practical application follows right from Theorem 4.4.1. Dependencies in D^{std} can be identified by checking descendants in the connection forest of a PCNF ψ. The advantage compared to Theorem 4.3.2 is that the connection forest for a PCNF ψ can be computed once and for all. It can be re-used each time the condition $H_i^*(x) \cap H_j^*(y) \neq \emptyset$

in Theorem 4.4.1 is checked. This way, we also avoid searching for X-paths explicitly like in direct applications of Definition 3.4.17. Further, we expect the connection forest to be compact because it is defined over equivalence classes of variables.

4.5 Compact Dependency Graphs

In the previous section, we observed several theoretical properties of connections involved in the computation of D^{std}. Starting from explicit connections as in Equation 4.1, we defined the equivalence relation $\approx$ on variables. In Equation 4.2 we pointed out that dependencies can be computed by considering connections between equivalence classes. Finally, we applied a tree-like representation of connections between classes of existential variables as in Equation 4.3 and Theorem 4.4.1.

Now we focus on practical application. Our goal is to obtain a compact graph representation for D^{std}. Instead of the augmented compressed dependency graph $G_{\approx,\subseteq}(D^{\text{std}})$ by Definition 4.2.8 on page 60, we construct an *approximation* thereof. As noted at the beginning of Section 4.3, *the approximation is precise with respect to dependencies by D^{std} but not with respect to graph structure.* There is a dependency with respect to the approximation if and only if there is a dependency with respect to D^{std}. In Section 4.6 below, we show that the approximation can be computed efficiently for formulae from real-world applications, although it might not allow for full compactness as the original graph $G_{\approx,\subseteq}(D^{\text{std}})$ by Definition 4.2.8. Thus we trade compactness for efficiency of computation.

First, in Section 4.5.1 we relate theoretical results from the previous section to the definition of the original graph $G_{\approx,\subseteq}(D^{\text{std}})$. These observations are the basis for defining an approximation of $G_{\approx,\subseteq}(D^{\text{std}})$. As we consider only D^{std}, we write $\approx_{\uparrow}$ and $\approx_{\downarrow}$ instead of $\approx_{D^{\text{std}},\uparrow}$ and $\approx_{D^{\text{std}},\downarrow}$ throughout this section. It turns out that on existential variables the equivalence relation $\approx_{\uparrow}$ by Definition 4.2.4 can be approximated by equivalence relation $\approx$ by Definition 4.3.3. Further, on universal variables the equivalence relation $\approx_{\uparrow}$ can be approximated by $\approx_{\downarrow}$. For existential variables, subset edges E_s by Definition 4.2.8 can be approximated by the directed connection relation $\rightsquigarrow^*$ from Definition 4.3.5, which is represented by the connection forest.

Based on these observations, we present an algorithm to construct an approximation of $G_{\approx,\subseteq}(D^{\text{std}})$ for a given PCNF ψ in Section 4.5.2. An experimental evaluation in Section 4.6 confirms its practical efficiency on formulae from recent QBF evaluations.

4.5.1 Approximations

We point out properties of D^{std} related to theoretical results from Section 4.3. We want to approximate $G_{\approx,\subseteq}(D^{\text{std}})$ by Definition 4.2.8 by approximating the set $V/_{\approx_\downarrow} \cup V/_{\approx_\uparrow}$ of vertices and edges E. The idea is to avoid computing the exact equivalence relations $\approx_\uparrow$ and $\approx_\downarrow$ by Definitions 4.2.3 and 4.2.4 where $G_{\approx,\subseteq}(D^{\mathrm{std}})$ is based on. Instead, we approximate $\approx_\uparrow$ and $\approx_\downarrow$ by other equivalence relations to be introduced below.

Definition 4.5.1 (see also [98], for example)**.** Given a set of variables V, equivalence relations $\sim$ and $\dot\sim$ over V and the partitions $V/_\sim$ and $V/_{\dot\sim}$ of V induced by $\sim$ and $\dot\sim$, respectively. Partition $V/_{\dot\sim}$ is an *underapproximation* of $V/_\sim$ and $V/_\sim$ is an *overapproximation* of $V/_{\dot\sim}$ if for each equivalence class $S \in V/_{\dot\sim}$ there is an equivalence class $S' \in V/_\sim$ such that $S \subseteq S'$.

If variables $x, x' \in V$ are equivalent with respect to $V/_{\dot\sim}$ in Definition 4.5.1, then also with respect to $V/_\sim$ but not necessarily vice versa. Therefore, the underapproximation $V/_{\dot\sim}$ might achieve less compaction on the set of variables than $V/_\sim$ because of smaller equivalence classes.

In this section, we show how to underapproximate $\approx_\uparrow$ and $\approx_\downarrow$ from Definitions 4.2.3 and 4.2.4. In these underapproximations defined below, we only consider variables within the same quantifier block as equivalent under certain additional criteria. This way, we obtain a partition of quantifier blocks rather than variables which allows for simpler implementation as pointed out in Section 4.5.2. Finally, the underapproximations of $\approx_\uparrow$ and $\approx_\downarrow$ give rise to an approximation of $G_{\approx,\subseteq}(D^{\mathrm{std}})$. Note that here we focus on D^{std}. It is unclear how to obtain similar theoretical results and approximations with respect to other dependency schemes which refine D^{std} such as the triangle or quadrangle dependency scheme [51, 113].

In the following, we present the theoretical background where the underapproximations of $\approx_\uparrow$ and $\approx_\downarrow$ and finally the approximation of graph $G_{\approx,\subseteq}(D^{\mathrm{std}})$ are based on.

Lemma 4.5.1. *Given a PCNF ψ over variables V and $y, y' \in V$ where $q(y) = q(y') = \exists$ and $\delta(y) = \delta(y') = j$: if $y \approx y'$ then $y \approx_\uparrow y'$.*

Proof. Assume that $y \approx y'$ and that there is $x \in V$ such that $x \in (D^{\mathrm{std}})^{-1}(y)$. Note that actually $q(x) = \forall$ and $\delta(x) = i$ for some i with $i < j$ by Theorem 4.3.2. We show that also $x \in (D^{\mathrm{std}})^{-1}(y')$ and hence $y \approx_\uparrow y'$ by Definition 4.2.4. Alternatively to $x \in (D^{\mathrm{std}})^{-1}(y)$ we can write $(x, y) \in D^{\mathrm{std}}$. By Equation 4.2 in Theorem 4.3.2 there is $w \in V_{\exists,\delta(x)+1}$ such that $x \rightarrow^*_{\delta(x)+1} [w]$ and $[y] \rightarrow^*_{\delta(x)+1} [w]$. Since $y \approx y'$, by Definition 4.3.3 $y \rightarrow^*_j y'$ for $j = \delta(y) = \delta(y')$ where $\delta(x) + 1 \leq j$. Hence $[y'] \rightarrow^*_{\delta(x)+1} [w]$ and therefore $(x, y') \in D^{\mathrm{std}}$ by Equation 4.2 in Theorem 4.3.2 and finally $x \in (D^{\mathrm{std}})^{-1}(y')$. □

By Lemma 4.5.1, if two existential variables y and y' from the same block are connected with respect to variables from the same or any larger existential block as in Definition 4.3.3, then they are equivalent with respect to $\approx$. In this case, y and y' also depend on the same set of universal variables and hence are equivalent by $\approx_\uparrow$. The converse does non necessarily hold. Two variables from different blocks but of the same quantifier type could depend on the same set of variables even if they are not equivalent by $\approx$.

Lemma 4.5.2. *Given a PCNF ψ over variables V and $y, y' \in V$ where $q(y) = q(y') = \forall$ and $\delta(y) = \delta(y') = j$: if $y \approx_\downarrow y'$ then $y \approx_\uparrow y'$.*

Proof. Assume that $y \approx_\downarrow y'$. By Definition 4.2.3 $D^{\text{std}}(y) = D^{\text{std}}(y')$. Assume that $(x, y) \in D^{\text{std}}$ for some $x \in V_\exists$ and $i = \delta(x)$ where $i < j$ by Theorem 4.3.2. By Equation 4.3 there is an arbitrary but fixed $w \in V_{\exists,\delta(x)+1}$ such that $x \rightarrow^*_{\delta(x)+1} [w]$ and $[y] \rightsquigarrow^* [w]$. By $[y] \rightsquigarrow^* [w]$, Definition 4.3.5 and Theorem 4.3.2, also $(y, w) \in D^{\text{std}}$. By assumption $D^{\text{std}}(y) = D^{\text{std}}(y')$, also $(y', w) \in D^{\text{std}}$. Since $(y', w) \in D^{\text{std}}$, by Equation 4.3 in Theorem 4.3.2 also $y' \rightarrow^*_{\delta(y')+1} [w]$ and $[w] \rightsquigarrow^* [w]$. By $y' \rightarrow^*_{\delta(y')+1} [w]$, Definition 4.3.5 and Corollary 4.3.3, also $[y'] \rightsquigarrow^* [w]$. By assumption $x \rightarrow^*_{\delta(x)+1} [w]$ and since $[y'] \rightsquigarrow^* [w]$, also $(x, y') \in D^{\text{std}}$ by Equation 4.3 in Theorem 4.3.2. □

By Lemma 4.5.2, if two universal variables from the same block have equal sets of depending variables, then they also depend on the same set of variables.

Lemma 4.5.3. *Given a PCNF ψ over variables V and $x, x' \in V$ where $\delta(x) = \delta(x') = i$: if $H_{i+1}(x) = H_{i+1}(x')$ then $x \approx_\downarrow x'$.*

Proof. Assume that $H_{i+1}(x) = H_{i+1}(x')$. Then by Definition 4.4.3 also $H^*_{i+1}(x) = H^*_{i+1}(x')$. Assume that $(x, y) \in D^{\text{std}}$ by Theorem 4.4.1, that is $H^*_{i+1}(x) \cap H^*_j(y) \neq \emptyset$ for $j := \delta(y)$. Let $[z] \in H^*_{i+1}(x) \cap H^*_j(y)$. Since by assumption $H^*_{i+1}(x) = H^*_{i+1}(x')$, also $[z] \in H^*_{i+1}(x') \cap H^*_j(y)$. Then $H^*_{i+1}(x') \cap H^*_j(y) \neq \emptyset$ for $j := \delta(y)$ and by Theorem 4.4.1 also $(x', y) \in D^{\text{std}}$. □

By Lemma 4.5.3, variables with equal sets of ancestors in the connection forest also have equal sets of depending variables. This follows right from Definition 4.4.3. If the sets of descendants in the connection forest are equal, then the sets of connected existential variables are equal. By Theorem 4.4.1 this information is precise and relevant to compute dependencies.

Lemma 4.5.4. *Given $y, y' \in V_\exists$, if $[y] \rightsquigarrow^* [y']$ then $([y]_\uparrow, [y']_\uparrow) \in E_s$, where E_s as in Definition 4.2.8.*

Proof. We show that if $[y] \rightsquigarrow^* [y']$ then $(D^{\text{std}})^{-1}(y) \subseteq (D^{\text{std}})^{-1}(y')$ and hence $([y]_\uparrow, [y']_\uparrow) \in E_s$ by Definition 4.2.8. Assume that $[y] \rightsquigarrow^* [y']$ and $x \in (D^{\text{std}})^{-1}(y)$ for some $x \in V_\forall$. Alternatively to $x \in (D^{\text{std}})^{-1}(y)$ we can write $(x, y) \in D^{\text{std}}$. By Equation 4.3 in Theorem 4.3.2, there is an arbitrary but fixed $w \in V_{\exists,\delta(x)+1}$ such that $x \rightarrow^*_{\delta(x)+1} [w]$ and $[y] \rightsquigarrow^* [w]$. Since $[y] \rightsquigarrow^* [w]$, also $\delta(y) \leq \delta(w)$ and $[y] \rightarrow^*_{\delta(y)} [w]$ by Definition 4.3.5. Then by Lemma 4.3.1, also $[w] \rightarrow^*_{\delta(y)} [y]$. Since $[y] \rightsquigarrow^* [y']$, also $\delta(y) \leq \delta(y')$ and $[y] \rightarrow^*_{\delta(y)} [y']$ by Definition 4.3.5. From $[w] \rightarrow^*_{\delta(y)} [y]$, $[y] \rightarrow^*_{\delta(y)} [y']$ and transitivity, we obtain $[w] \rightarrow^*_{\delta(y)} [y']$. By Lemma 4.3.1, also $[y'] \rightarrow^*_{\delta(y)} [w]$. Since $\delta(x) < \delta(y) \leq \delta(y')$, also $[y'] \rightarrow^*_{\delta(x)+1} [w]$ by Corollary 4.3.3. From $x \rightarrow^*_{\delta(x)+1} [w]$, $[y'] \rightarrow^*_{\delta(x)+1} [w]$ and Equation 4.2 in Theorem 4.3.2, we conclude that $(x, y') \in D^{\text{std}}$, or alternatively $x \in (D^{\text{std}})^{-1}(y')$. Therefore $(D^{\text{std}})^{-1}(y) \subseteq (D^{\text{std}})^{-1}(y')$ and hence $([y]_\uparrow, [y']_\uparrow) \in E_s$ by Definition 4.2.8. $\square$

Lemma 4.5.4 relates paths between classes in the connection forest to subset edges in $G_{\approx,\subseteq}(D^{\text{std}})$. Given existential variables y and y' such that $[y] \rightsquigarrow^* [y']$, the connection from a universal variable x to y can be extended to y'. Thus if $(x, y) \in D^{\text{std}}$ then also $(x, y') \in D^{\text{std}}$ and hence there is a subset edge $([y]_\uparrow, [y']_\uparrow)$. Relying on the lemmata introduced above, we define approximations of equivalence relations $\approx_\downarrow$ and $\approx_\uparrow$ from Definitions 4.2.3 and 4.2.4.

Definition 4.5.2. Given a PCNF over variables V and $x, x' \in V$ such that $\delta(x) = \delta(x') = i$. The equivalence relation $\dot{\approx}_\downarrow$ is defined based on Lemma 4.5.3:

$$x \dot{\approx}_\downarrow x' \text{ if and only if } H_{i+1}(x) = H_{i+1}(x').$$

Given $x \in V$, $[x]_{\dot\downarrow}$ is the class of x with respect to $V/_{\dot{\approx}_\downarrow}$.

Corollary 4.5.1. *Given a PCNF over variables V, the partition $V/_{\dot{\approx}_\downarrow}$ is an underapproximation of $V/_{\approx_\downarrow}$ due to Lemma 4.5.3.*

Definition 4.5.3. Given a PCNF over variables V and $x, x' \in V$ such that $\delta(x) = \delta(x') = i$. The equivalence relation $\dot{\approx}_\uparrow$ is defined based on Lemmata 4.5.1 to 4.5.3 with respect to the type of x and x':

- $q(x) = q(x') = \exists$: $x \dot{\approx}_\uparrow x'$ if and only if $x \approx x'$.
- $q(x) = q(x') = \forall$: $x \dot{\approx}_\uparrow x'$ if and only if $H_{i+1}(x) = H_{i+1}(x')$.

Given $x \in V$, $[x]_{\dot\uparrow}$ is the class of x with respect to $V/_{\dot{\approx}_\uparrow}$.

Corollary 4.5.2. *Given a PCNF over variables V, the partition $V/_{\dot{\approx}_\uparrow}$ is an underapproximation of $V/_{\approx_\uparrow}$ due to Lemmata 4.5.1 to 4.5.3.*

Given the underapproximations $\dot{\approx}_\downarrow$ and $\dot{\approx}_\uparrow$ from Definitions 4.5.2 and 4.5.3, we introduce an approximation of the graph $G_{\approx,\subseteq}(D^{\text{std}})$ from Definition 4.2.8. This approximation goes without the original equivalence relations $\approx_\downarrow$ and $\approx_\uparrow$ from Definitions 4.2.3 and 4.2.4.

Definition 4.5.4. Given the standard dependency scheme D^{std} for a PCNF ψ over variables V and equivalence relations $\dot{\approx}_\downarrow$ and $\dot{\approx}_\uparrow$. The *approximated,* *augmented and compressed dependency graph* for D^{std} is a DAG $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$ with vertices $V/_{\dot{\approx}_\downarrow} \cup V/_{\dot{\approx}_\uparrow}$ and directed edges $E := E'_d \cup E'_s$ where

$$E'_d := \{([x]_{\dot{\downarrow}}, [y]_{\dot{\uparrow}}) \mid (x, y) \in D^{\text{std}}\}$$

is the set of *dependency edges* and

$$\begin{aligned} E'_s \;:=\; & \{([x]_{\dot{\uparrow}}, [x']_{\dot{\uparrow}}) \mid q(x) = q(x') = \exists \text{ and } [x]_{\dot{\uparrow}} \rightsquigarrow^* [x']_{\dot{\uparrow}}\} \cup \\ & \{([x]_{\dot{\uparrow}}, [x']_{\dot{\uparrow}}) \mid q(x) = q(x') = \forall \text{ and } (D^{\text{std}})^{-1}(x) \subseteq (D^{\text{std}})^{-1}(x')\} \end{aligned}$$

is the set of *subset edges*.

Note that, when defining E'_s in Definition 4.5.4, notation $[x]_{\dot{\uparrow}} \rightsquigarrow^* [x']_{\dot{\uparrow}}$ is well-defined because of Lemma 4.5.1 and Definition 4.5.3. That is, the equivalence relations $\approx$ and $\dot{\approx}_\uparrow$ are equal with respect to existential variables.

The reason for considering graph $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$ from Definition 4.5.4 an approximation of $G_{\approx,\subseteq}(D^{\text{std}})$ from Definition 4.2.8 is twofold. First, if two variables are equivalent in $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$, that is they occur in the same equivalence class, then they are also equivalent in $G_{\approx,\subseteq}(D^{\text{std}})$, but not necessarily vice versa. Hence classes in $G_{\approx,\subseteq}(D^{\text{std}})$ might be larger than in $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$ and therefore $G_{\approx,\subseteq}(D^{\text{std}})$ might be more compact. Second, we have $E'_s \subseteq E_s$ in general, because of underapproximations $\dot{\approx}_\downarrow$ and $\dot{\approx}_\uparrow$ in addition to $\rightsquigarrow^*$, where subset edges between classes of existential variables are based on. Note that, with respect to existential variables, every edge $([x], [x'])$ in the connection forest by Definition 4.4.1 corresponds to a subset edge $([x]_{\dot{\uparrow}}, [x']_{\dot{\uparrow}})$.

4.5.2 Computing Approximations

We illustrate an algorithm to construct the approximated dependency graph $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$ from Definition 4.5.4. The idea is to build the connection forest in advance, which represents directed connections between existential variables by Definitions 4.3.5 and 4.4.1. Due to equivalence classes we expect the connection forest to be compact. That precomputed information can then be used to complete the construction of $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$. Note that the connection forest we compute corresponds *exactly* to Definition 4.4.1 even though we finally obtain the *approximated* graph $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$. In Section 4.6, we show by experiments on formulae from QBF competitions that the run time of our algorithm is negligible. We consider an example in Section 4.5.3.

The algorithm we are going to sketch consists of different phases. It is possible to merge the first two phases shown below, which could give additional run time improvements in an implementation. However, for simplicity of presentation we keep the phases separate.

In the following, assume that we are given a PCNF ψ where we want to compute $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$ for D^{std} with respect to ψ. Initially all equivalence classes with respect to $\dot{\approx}_\uparrow$ and $\dot{\approx}_\downarrow$ are singleton. Further, all clauses in ψ are sorted ascendingly with respect to the ordering of the quantifier prefix.

Phase 1: Constructing the Connection Forest

In the first phase, the connection forest for ψ is constructed by inspecting the clauses in ψ. Clauses are inspected one after the other where only existential literals are considered. The equivalence relation $\approx$, which approximates $\dot{\approx}_{\uparrow}$ by Definition 4.5.3, is computed step by step with respect to the clauses that have been processed already.

1. Assume that the connection forest contains only singleton classes as vertices but no edges.
2. For all clauses C_j in ψ:
    - 2.1. Consider every pair (l_1, l_2) of existential literals in C_j such that the corresponding variables $v_1 := v(l_1)$ and $v_2 := v(l_2)$ are from the same existential block, that is $b(v_1) = b(v_2)$. Merge the classes $[v_1]$ and $[v_2]$ of v_1 and v_2 with respect to $\approx$. Note that by Definition 4.5.3 this corresponds to merging the classes $[v_1]_{\dot{\uparrow}}$ and $[v_2]_{\dot{\uparrow}}$. Since clauses are sorted, it is actually not necessary to consider all pairs of existential literals in C_j. One linear pass over literals in C_j is sufficient to carry out this step.
3. For all clauses C_j in ψ:
    - 3.1. Consider all pairs $([v_1], [v_2])$ of existential classes in C_j with respect to $\approx$ where $v_1 < v_2$ in the prefix ordering. Again it is not necessary to search for all pairs explicitly, as noted above. We add edges to the connection forest as follows.
        - 3.1.1. For $i := \delta(v_1)$, let $[v_2'] := h(i, [v_2])$ be the smallest ancestor of $[v_2]$ in the connection forest by Definition 4.4.2. Note that $v_2' \leq v_2$.
        - 3.1.2. If $[v_1] = [v_2']$ then do nothing and consider the next pair at the beginning of the loop.
        - 3.1.3. If $v_1 < v_2'$, that is v_1 and v_2' are from different blocks, then insert an edge $([v_1], [v_2'])$ into the connection forest as in Definition 4.4.1.
          Otherwise, if v_1 and v_2' are from the same block then merge the classes $[v_1]$ and $[v_2']$ with respect to $\approx$.
          At this point we might have destroyed the structure of the connection forest. After class merging or edge insertion, the vertex representing $[v_2']$ could have two parents.
        - 3.1.4. If class $[v_2']$ has one parent then do nothing and go to the next pair at the beginning of the loop.
        - 3.1.5. Otherwise, the forest structure has to be repaired. Consider the set of *all* predecessors of $[v_1]$ and $[v_2']$ in the connection

forest. Consider the sequence of that predecessors sorted by prefix ordering. If there are predecessors from the same block, then merge the respective classes with respect to $\approx$. Remove edges which connect a predecessor with another in the set. Connect all the sorted predecessors linearly by new edges in the connection forest.

4. At this point, the connection forest is fully constructed.

Phase 2: Computing Ancestors

Given the connection forest that has been constructed in the first phase, we compute the sets $H_i(x)$ of ancestors like in Definition 4.4.3 for each variable. The following algorithm is actually a direct application of that definition. Ancestor information is used to insert dependency edges into the graph and to compute equivalence classes with respect to Definition 4.5.2.

1. Assume that the connection forest was fully constructed.
2. For all clauses C_j in ψ, consider all the literals in C_j:
   2.1. Consider pairs $(v_1, [v_2])$ such that there are literals l_1 and l_2 in C_j with $v_1 = v(l_1)$ and $v_2 = v(l_2)$ where $v_1 < v_2$ and either $q(v_1) = q(v_2) = \exists$ or $q(v_1) = \forall$ and $q(v_2) = \exists$. Given v_2, find the smallest ancestor $[v_2'] := h(i, [v_2])$ of $[v_2]$, where $i := \delta(v_1) + 1$, which is larger than v_1 by prefix order. Let $H_i(v_1) := H_i(v_1) \cup \{[v_2']\}$ where $i := \delta(v_1) + 1$. The previous steps correspond to Definition 4.4.3.
3. At this point, all the sets $H_i(v)$ of ancestors where $i = \delta(v) + 1$ were computed for all variables in ψ.

Due to equivalence classes $[v_2]$ with respect to $\approx$, we expect that not all of the literals in a clause have to be considered explicitly during phase 2. If v is universal then set $H_i(v)$ corresponds to dependency edges from universal classes with respect to $\dot{\approx}_{\downarrow}$ by Definition 4.5.2 to existential classes with respect to $\dot{\approx}_{\uparrow}$. By Definition 4.5.3 the latter corresponds to $\approx$ already built in the first phase.

Phase 3: Computing Partitions

Equivalence relations $\dot{\approx}_{\downarrow}$ and $\dot{\approx}_{\uparrow}$ by Definitions 4.5.2 and 4.5.3 are part of the approximated dependency graph $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$. These relations take into account equivalences of ancestors sets $H_i(v)$ that were computed in the previous phase. In this phase, we explicitly compare ancestor sets to compute the partitions $V/_{\dot{\approx}_{\downarrow}}$ and $V/_{\dot{\approx}_{\uparrow}}$. For existential variables, $V/_{\dot{\approx}_{\uparrow}}$ was computed already implicitly by $\approx$ in phase 1 due to Definition 4.5.3.

Additionally by Definition 4.5.2, $\dot{\approx}_{\downarrow}$ equals $\dot{\approx}_{\uparrow}$ for universal variables. Thus it remains to consider $\dot{\approx}_{\downarrow}$ for both universal and existential variables.

1. Assume that the connection forest and sets $H_i(v)$ where $i = \delta(v) + 1$ for all variables v were constructed previously.

2. For all existential blocks B_j:

   2.1. As long as there are classes $[e]_{\dot{\downarrow}}$ and $[e']_{\dot{\downarrow}}$ with $[e]_{\dot{\downarrow}} \neq [e']_{\dot{\downarrow}}$ by $\dot{\approx}_{\downarrow}$ for variables e and e' in B_i such that $H_{i+1}(e) = H_{i+1}(e')$, merge $[e]_{\dot{\downarrow}}$ and $[e']_{\dot{\downarrow}}$.

3. For all universal blocks B_j:

   3.1. As long as there are classes $[a]_{\dot{\downarrow}}$ and $[a']_{\dot{\downarrow}}$ with $[a]_{\dot{\downarrow}} \neq [a']_{\dot{\downarrow}}$ by $\dot{\approx}_{\downarrow}$ for variables a and a' in B_i such that $H_{i+1}(a) = H_{i+1}(a')$, merge $[a]_{\dot{\downarrow}}$ and $[a']_{\dot{\downarrow}}$ as well as $[a]_{\dot{\uparrow}}$ and $[a']_{\dot{\uparrow}}$.

Phase 4: Inserting Dependency Edges

In phase 2 we computed ancestor sets $H_i(v)$ for all variables in ψ and $i := \delta(v)+1$. Regarding universal variables v, each ancestor $[v']$ in $H_i(v)$ corresponds to a dependency edge $([v]_{\dot{\downarrow}}, [v']_{\dot{\uparrow}})$ in the approximated dependency graph by Definition 4.5.4. For existential variables, we explicitly insert dependency edges based on the connection forest and partitions $V/_{\dot{\approx}_{\downarrow}}$ that were computed in the previous phase. Thereby, we directly apply Theorem 4.4.1.

1. Assume that the connection forest and partitions $V/_{\dot{\approx}_{\downarrow}}$ and $V/_{\dot{\approx}_{\uparrow}}$ were computed.

2. For all universal variables a in ψ, consider the class $[a]_{\dot{\downarrow}}$ with respect to $\dot{\approx}_{\downarrow}$:

   2.1. For all classes $[e]$ of existential variables with respect to $\approx$ (note that $[e]$ corresponds to $[e]_{\dot{\uparrow}}$ by Definition 4.5.3) such that $[e] \in H_i(a)$ where $i = \delta(a) + 1$:

      2.1.1. For all predecessors $[e']$ of $[e]$ (including $[e'] := [e]$ itself) in the connection forest:

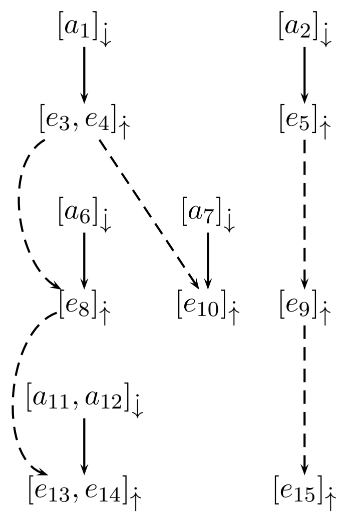
         2.1.1.1. If there is a class $[e'']_{\dot{\downarrow}}$ with respect to $\dot{\approx}_{\downarrow}$ where e'' is existential such that $[e'] \in H_i(e'')$ and $i = \delta(e'') + 1$, then insert a dependency edge $([e'']_{\dot{\downarrow}}, [a]_{\dot{\downarrow}})$. Note that we may write $([e'']_{\dot{\downarrow}}, [a]_{\dot{\downarrow}})$ instead of $([e'']_{\dot{\downarrow}}, [a]_{\dot{\uparrow}})$ since $[a]_{\dot{\downarrow}}$ equals $[a]_{\dot{\uparrow}}$ by Definitions 4.5.2 and 4.5.3.

Phase 5: Inserting Subset Edges

We finish the construction of the approximated dependency graph by inserting missing subset edges. For existential variables, edges in the connection forest by Definition 4.4.1 correspond to subset edges between classes with respect to $\dot{\approx}_{\uparrow}$ because of Definition 4.5.3 and Lemma 4.5.4. Since the connection forest represents the transitive reduction of relation $\leadsto^*$, there are no transitive subset edges between classes of existential variables. This property is different from Definition 4.5.4 and can be regarded as an improvement.

In the final phase, we insert subset edges between classes of universal variables explicitly. Optionally, transitive edges can be removed afterwards. We consider classes $[a]_{\dot{\uparrow}}$ of universal variables a from blocks B_i from the outermost to the innermost. If there is a class $[a']_{\dot{\uparrow}}$ of a universal variable a' from a larger block B_j where $i < j$ such that $(D^{\text{std}})^{-1}(a) \subseteq (D^{\text{std}})^{-1}(a')$ then we insert a subset edge $([a]_{\dot{\uparrow}}, [a']_{\dot{\uparrow}})$.

4.5.3 Graph Example



$$\begin{tikzcd}
{[a_1]_{\dot{\downarrow}}} \arrow[d] & & {[a_2]_{\dot{\downarrow}}} \arrow[d] \\
{[e_3, e_4]_{\dot{\uparrow}}} \arrow[dd, dashed, bend right] \arrow[ddr, dashed] & & {[e_5]_{\dot{\uparrow}}} \arrow[dd, dashed] \\
{[a_6]_{\dot{\downarrow}}} \arrow[d] & {[a_7]_{\dot{\downarrow}}} \arrow[d] & \\
{[e_8]_{\dot{\uparrow}}} \arrow[dd, dashed, bend right] & {[e_{10}]_{\dot{\uparrow}}} & {[e_9]_{\dot{\uparrow}}} \arrow[dd, dashed] \\
{[a_{11}, a_{12}]_{\dot{\downarrow}}} \arrow[d] & & \\
{[e_{13}, e_{14}]_{\dot{\uparrow}}} & & {[e_{15}]_{\dot{\uparrow}}}
\end{tikzcd}$$

x	$H_{\delta(x)+1}(x)$	x	$H_{\delta(x)+1}(x)$
a_1	$[e_3, e_4]_{\dot{\uparrow}}$	e_9	$[e_{15}]_{\dot{\uparrow}}$
a_2	$[e_5]_{\dot{\uparrow}}$	e_{10}	–
e_3	$[e_8]_{\dot{\uparrow}}$	a_{11}	$[e_{13}, e_{14}]_{\dot{\uparrow}}$
e_4	$[e_8]_{\dot{\uparrow}}, [e_{10}]_{\dot{\uparrow}}$	a_{12}	$[e_{13}, e_{14}]_{\dot{\uparrow}}$
e_5	$[e_9]_{\dot{\uparrow}}$	e_{13}	–
a_6	$[e_8]_{\dot{\uparrow}}$	e_{14}	–
a_7	$[e_{10}]_{\dot{\uparrow}}$	e_{15}	–
e_8	$[e_{13}, e_{14}]_{\dot{\uparrow}}$		

i	$q(B_i)$	B_i	$(a_2 \vee e_5 \vee e_9)$
1	$\forall$	a_1, a_2	$(e_5 \vee e_9 \vee e_{15})$
2	$\exists$	e_3, e_4, e_5	$(e_3 \vee e_8 \vee e_{13})$
3	$\forall$	a_6, a_7	$(e_4 \vee a_7 \vee e_{10})$
4	$\exists$	e_8, e_9, e_{10}	$(e_4 \vee e_{13} \vee e_{14})$
5	$\forall$	a_{11}, a_{12}	$(a_1 \vee a_6 \vee e_8 \vee e_{14})$
6	$\exists$	e_{13}, e_{14}, e_{15}	$(a_{11} \vee a_{12} \vee e_{13})$

Figure 4.3: The connection forest (dashed edges) and dependency edges (solid) from universal to existential classes for the PCNF from Figure 4.2, which is replicated in the table. For simplicity, we omit dependency edges from existential to universal classes and subset edges between universal classes. Further, classes by $\dot{\approx}_{\downarrow}$ and $\dot{\approx}_{\uparrow}$ of existential and universal variables, respectively, are omitted. Note that the dashed edges of the connection forest correspond to subset edges between classes of existential variables. The table shows the set of ancestors $H_{\delta(x)+1}(x)$ by Definition 4.4.3.

Figure 4.3 shows part of the approximated graph for the PCNF from Figure 4.2 as constructed by the algorithm described in the previous section. Dashed edges are part of the connection forest. Solid edges represent dependencies of universal classes by equivalence relation $\dot{\approx}_{\downarrow}$ according to Definition 4.5.2. For example, we have $a_{11} \dot{\approx}_{\downarrow} a_{12}$ because $H_{\delta(a_{11})+1}(a_{11}) = H_{\delta(a_{12})+1}(a_{12}) = \{[e_{13}, e_{14}]_{\dot{\uparrow}}\}$ as indicated in the table. Hence variables a_{11} and a_{12} have the same set of dependencies and the single dependency edge from $[a_{11}, a_{12}]_{\dot{\downarrow}}$ to $[e_{13}, e_{14}]_{\dot{\uparrow}}$ in the graph compactly represents the four explicit dependencies (a_{11}, e_{13}), (a_{11}, e_{14}), (a_{12}, e_{13}) and (a_{12}, e_{14}) in D^{std}. Further, we have $e_{13} \dot{\approx}_{\uparrow} e_{14}$ by Definition 4.5.3, which is due to $e_{13} \approx e_{14}$ by Definition 4.3.3 and by clause $(e_4 \vee e_{13} \vee e_{14})$.

In addition to classes, subset edges between existential variables in Figure 4.3 allow to reduce the number of explicit dependency edges. For example, the dependency (a_2, e_{15}) is represented implicitly by the dependency edge from $[a_2]_{\dot{\downarrow}}$ to $[e_5]_{\dot{\uparrow}}$ and by the path from $[e_5]_{\dot{\uparrow}}$ to $[e_{15}]_{\dot{\uparrow}}$ given by dashed subset edges.

We briefly sketch how to insert dependency edges from existential to universal classes (not shown in Figure 4.3) as done in phase four of the algorithm from the previous section. Recall that, for a universal variable a, $[a]_{\dot{\downarrow}}$ equals $[a]_{\dot{\uparrow}}$ by Definitions 4.5.2 and 4.5.3. We start at the universal class $[a_{11}, a_{12}]_{\dot{\downarrow}}$ and consider the existential class $[e_{13}, e_{14}]_{\dot{\uparrow}}$, which is the only class in the set $H_{\delta(a_{11})+1}(a_{11})$. Further, we consider predecessors of $[e_{13}, e_{14}]_{\dot{\uparrow}}$ in the connection forest, which are $[e_{13}, e_{14}]_{\dot{\uparrow}}$, $[e_8]_{\dot{\uparrow}}$ and $[e_3, e_4]_{\dot{\uparrow}}$. We have $[e_{13}, e_{14}]_{\dot{\uparrow}} \in H_{\delta(e_8)+1}(e_8)$ for predecessor $[e_{13}, e_{14}]_{\dot{\uparrow}}$ and $[e_8]_{\dot{\uparrow}} \in H_{\delta(e_3)+1}(e_3)$ and $[e_8]_{\dot{\uparrow}} \in H_{\delta(e_4)+1}(e_4)$ for predecessor $[e_8]_{\dot{\uparrow}}$. Therefore, we add dependency edges $([e_8]_{\dot{\downarrow}}, [a_{11}, a_{12}]_{\dot{\uparrow}})$, $([e_3]_{\dot{\downarrow}}, [a_{11}, a_{12}]_{\dot{\uparrow}})$ and $([e_4]_{\dot{\downarrow}}, [a_{11}, a_{12}]_{\dot{\uparrow}})$.

For example, note that $e_3 \not\dot{\approx}_{\downarrow} e_4$ by Definition 4.5.2 since $H_{\delta(e_3)+1}(e_3) \neq H_{\delta(e_4)+1}(e_4)$. Hence $a_7 \in D^{\text{std}}(e_4)$ but $a_7 \notin D^{\text{std}}(e_3)$.

4.6 Experimental Results

We implemented a variant of the algorithm presented in Section 4.5.2 in an experimental tool called qdag.[2] The tool constructs the approximated graph $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$ similar to Definition 4.5.4 for a given PCNF ψ. An efficient union-find data structure [126] is used to represent equivalence classes. However, the tool is actually subsumed by an implementation of approximated dependency graphs which is part of the search-based QBF solver DepQBF described in Chapter 5.

In this section, we report experimental results with respect to qdag which were already published in [82]. Different from the algorithm from Section 4.5.2, the first two phases were merged in the implementation of qdag. Thus one of the passes over the set of clauses of PCNF ψ can be

[2]A binary and log-files of experiments are available from `http://fmv.jku.at/qdag/`.

	2005	*2006*	*2007*	*2008*
Size	211	216	1136	3328
Explicit Search for X-Paths				
Total Time	19527.3	132.04	101424	173349
Max. Time	900 (15)	8.67	900 (83)	900 (135)
Avg. Time	92.54	0.61	89.28	52.12
Graph-Based Computation				
Total Time	7.94	1.35	227.05	300.31
Max. Time	0.58	0.03	7.96	8.11
Avg. Time	0.04	0.01	0.2	0.09
$x \in V_\forall$				
Max. $\lvert D^{\mathrm{std}}(x)\rvert$	256535	9993	2177280	2177280
Avg. $\lvert D^{\mathrm{std}}(x)\rvert$	82055.87	4794.60	33447.6	19807
Max. $\lvert H_i(x)\rvert$	256	1	518	518
Avg. $\lvert H_i(x)\rvert$	3.26	0.98	2.02	1.14
Max. $\lvert H_i^*(x)\rvert$	797	5	797	1872
Avg. $\lvert H_i^*(x)\rvert$	19.51	1.12	39.06	8.24
Avg. $\frac{\lvert\{[y]_{\dot\uparrow} \mid y \in D^{\mathrm{std}}(x)\}\rvert}{\lvert\{y \mid y \in D^{\mathrm{std}}(x)\}\rvert}$	3.44%	0.04%	6.42%	1.21%
Classes per Variables	28.2%	10.23%	40.31%	21.29%
$x \in V_\exists$				
Max. $\lvert D^{\mathrm{std}}(x)\rvert$	5040	440	5040	22696
Avg. $\lvert D^{\mathrm{std}}(x)\rvert$	12.76	2.98	3.24	4
Max. $\lvert H_i(x)\rvert$	24	7	490	490
Avg. $\lvert H_i(x)\rvert$	0.14	0.13	0.17	0.13
Max. $\lvert H_i^*(x)\rvert$	797	7	797	1872
Avg. $\lvert H_i^*(x)\rvert$	5.16	0.16	1.32	1.31
Avg. $\frac{\lvert\{[y]_{\dot\uparrow} \mid y \in D^{\mathrm{std}}(x)\}\rvert}{\lvert\{y \mid y \in D^{\mathrm{std}}(x)\}\rvert}$	2.37%	0.4%	2.76%	2.09%
Classes per Variables	10.96%	4.99%	11.45%	7.11%

Table 4.1: Computing the standard dependency scheme D^{std} on structured instances from QBF evaluations 2005 to 2008 [56]. We compare explicit search for X-paths to our algorithm which constructs a variant of the approximated, augmented and compressed dependency graph $\dot{G}_{\approx,\subseteq}(D^{\mathrm{std}})$.

avoided. The remaining phases consider equivalence classes rather than individual variables. Depending on the size of the classes, we expect that the effort spend on that phases can be kept low in practice. In our implementation of qdag, we also skip phase five where subset edges between classes of universal variables are inserted. This information is not relevant as we are only interested in the sets of dependencies by D^{std} in the experiments reported here. Further, the partition with respect to $\dot{\approx}_{\downarrow}$ for existential variables is not computed in phase three. This was included in a later version of DepQBF. For the experimental analysis, we considered structured instances (called "fixed class") from QBF evaluations 2005 to 2008 [56]. We did not include random instances. Experiments were run on 64-bit Ubuntu Linux 8.04, Intel® Q6700 at 2.66 GHz and 8 GB of memory.

Table 4.1 shows experimental results. For all formulae in the benchmark sets a variant of the approximated dependency graph $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$ was constructed using the tool qdag as described above. The first line in Table 4.1 shows the numbers of formulae per benchmark set.

First, we compare the effort of computing D^{std} by explicit search for X-paths by Definition 3.4.17 on page 49 and by construction of $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$. The former approach was pointed out in Example 4.1.1 on page 57 and is also implemented in qdag.

We report the total run time, the maximum over all formulae and the average per formula in seconds for the two approaches in the table. It can be seen that explicit search is significantly worse. In contrast to the graph-based approach, explicit search times out on several formulae, where we used a time limit of 900 seconds. The number of timeouts is given in parentheses in row "*Max. Time*". Further, the average time spent by the graph-based approach is negligible in practice. Consequently, when computing $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$ to be used in combination with a QBF solver, we expect almost no overhead with respect to run time. Combinations of dependency graphs with search-based QBF solvers are the topic of Chapter 5.

Next, we measure the quality of $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$ in terms of compactness with respect to partitions $V/_{\dot{\approx}_{\downarrow}}$ and $V/_{\dot{\approx}_{\uparrow}}$. We relate the number of dependencies of a variable to the sizes of the equivalence classes which represent that dependencies in the graph. This way, the effectiveness of the graph-based approach can be evaluated. Statistics are divided into two sections for existential and universal variables, respectively. Maximum and average numbers of dependencies by D^{std} over all variables are shown.

Compactness of the graph is indicated several times. For $x \in V_{\forall}$, the set $H_i^*(x)$ of descendants for $i = \delta(x) + 1$ in the connection forest efficiently represents $D^{\text{std}}(x)$. These classes are reachable from ancestors in $H_i(x)$. When comparing the numbers $|H_i(x)|$, $|H_i^*(x)|$ and $|D^{\text{std}}(x)|$, it becomes apparent that $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$ is compact. For example, on the benchmark set from 2008, universal variables have 19807 dependencies on average but this information is stored in $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$ using only 8.24 descendants on average.

Recall that the connection forest is computed only once an thus shared between all the variables in a PCNF. Descendants $H_i^*(x)$ are *not* stored explicitly for each variable. The data reported for $|H_i^*(x)|$ in the table was obtained by traversing the connection forest starting from ancestors $H_i(x)$. The small values of $|H_i^*(x)|$ for universal variables x show that many existential variables occur in the same equivalence class by partition $V/_{\dot{\approx}_\uparrow}$.

For existential variables $x \in V_\exists$, $|H_i(x)|$ and $|H_i^*(x)|$ measure the effort for inserting dependency edges in phase four since $H_i(x)$ and $H_i^*(x)$ have to be checked explicitly. Note that in these experiments we did not compute the partition $V/_{\dot{\approx}_\downarrow}$ for existential variables with respect to sets $|H_i(x)|$ by Definitions 4.5.2. Making use of classes with respect to $V/_{\dot{\approx}_\downarrow}$ rather than individual variables as we did in Section 4.5.2 could further improve the computation of dependency edges.

Finally, we evaluate the size of equivalence classes in partition $V/_{\dot{\approx}_\uparrow}$. Recall that for universal variables $V/_{\dot{\approx}_\uparrow}$ is equal to $V/_{\dot{\approx}_\downarrow}$. For existential variables, we did not construct $V/_{\dot{\approx}_\downarrow}$ as noted above. Given a variable x, we computed the number $|\{y \mid D^{\text{std}}(x)\}|$ of dependencies and the number $|\{[y]_{\dot{\uparrow}} \mid y \in D^{\text{std}}(x)\}|$ of classes of depending variables by $\dot{\approx}_\uparrow$. The average fraction of these numbers is shown in the table. The worst-case is 100%, where each depending variable occurs in a singleton class with respect to $\dot{\approx}_\uparrow$. This is clearly not the case. The partition $V/_{\dot{\approx}_\uparrow}$ compactly represents the sets of dependencies. Line "*Classes per Variables*" shows the average number of classes per variable with respect to partition $V/_{\dot{\approx}_\uparrow}$ in each formula. Again, values are far below 100%, hence many variables can be regarded as equivalent by $\dot{\approx}_\uparrow$.

4.7 Summary

The standard dependency scheme D^{std} can be computed in polynomial time by direct applications of Definition 3.4.17. However, explicitly searching for connections between variables can be infeasible on large formulae.

As an alternative, we considered graph representations for arbitrary dependency schemes. In explicit dependency graphs, edges correspond exactly to dependencies of the form (x, y) in a given dependency scheme D. We introduced equivalence relations over the set of variables (or vertices) based on dependency information by D. Thereby, we obtained graphs over classes of variables which are potentially more compact than explicit graphs. Finally, we suggested augmented, compressed dependency graphs to represent arbitrary dependency schemes. These graphs contain auxiliary edges which are useful for practical applications, as pointed out in the following chapter.

As an example for our graph-based representations, we presented an algorithm to compute an approximation of the augmented, compressed dependency graph for D^{std}. The approximation is precise with respect to dependencies by D^{std} but might achieve less compaction than the exact graph. We observed that X-paths by Definition 3.4.15 on page 48 often connect multiple variables. Hence it is not necessary to compute that information from scratch for each pair (x, y) of variables to check if $(x, y) \in D^{\text{std}}$. Instead, given a PCNF ψ, we first determine all connections between existential variables which are relevant for X-paths in terms of the connection forest of ψ. The connection forest is used to complete the construction of the graph by inserting missing dependency and auxiliary edges.

Although our presented algorithm construct only an approximation of the augmented, compressed dependency graph for D^{std}, experimental results on benchmarks from QBF evaluations show that the approximation is compact and that is can be computed efficiently in practice.

Chapter 5

QDPLL and Dependency Schemes

5.1 Introduction

In Chapter 3 we argued that the linear quantifier prefix of QBFs in PCNF might be a severe drawback for QBF solvers. The quantifier prefix naturally induces a linear ordering on the variables. This ordering restricts the set of possible assignment a search-based QBF solver can consider, for example. As pointed out in Example 3.3.6 on page 34, there might be an exponential gap between the run time of solvers relying on the quantifier prefix and solvers which carry out a more sophisticated dependency analysis.

In order to alleviate the limitations of quantifier prefixes, we introduced dependency schemes as a framework for dependency analysis in PCNFs in Section 3.4. Due to Proposition 3.4.5 on page 51, the linear quantifier prefix of PCNFs constitutes the worst of all dependency schemes. In contrast to quantifier prefixes, the use of dependency schemes in general allows a QBF solver to profit from additional freedom to assign variables. This way, the policy of assigning variables strictly from "left to right" in prefix ordering can be overcome.

In this chapter, we combine dependency schemes with search-based QBF solvers relying on QDPLL, which we briefly introduced in Section 2.3.1. The goal is to enable QBF solvers to profit from dependency information which is more refined than what can be obtained from quantifier prefixes. Although we focus on PCNF and search-based QBF solving by QDPLL, our approach is relevant for non-PCNF formulae and variable elimination as well. It turns out that the combination of dependency schemes and QDPLL does not require to change the overall structure of the algorithm.

Throughout QBF literature, QDPLL has been described based on the linear quantifier prefix. Since the prefix gives rise to the trivial dependency scheme by Definition 3.4.13, we can regard the original QDPLL algorithm [30] to be combined with that particular dependency scheme. We generalize that combination of QDPLL and the prefix-based trivial dependency scheme to arbitrary dependency schemes.

Apart from theoretical aspects of dependency schemes and QDPLL, we are interested in *efficient* practical applications. It is crucial to check for dependencies between variables within certain parts of QDPLL such as decision making or clause learning. In order to carry out such checks efficiently, we apply compact dependency graphs which we introduced in Section 4.2. These graphs are general and not limited to representations of particular dependency schemes.

We provide an experimental evaluation of QDPLL with dependency schemes. Our QDPLL-based QBF solver DepQBF [84] integrates dependency schemes as compact dependency graphs. We analyze the costs of moving from simple dependency information like linear quantifier prefixes of PCNFs or quantifier trees to more general dependency schemes. The results of this analysis give insights into practical applicability. Further, we evaluate the dynamic effects on QDPLL when combined with various dependency schemes. For that purpose, we implemented a common framework for compact dependency graphs in DepQBF. In addition to the standard dependency scheme D^{std}, this framework is able to represent the trivial dependency scheme D^{triv} arising from linear quantifier prefixes and D^{tree} given by quantifier trees as well. We compare the effects of D^{triv}, D^{tree} and D^{std} on the performance of QDPLL. Our experiments indicate that, despite increased overhead, the combination of QDPLL with D^{std} outperforms QDPLL relying on D^{triv} and D^{tree}. These results motivate the use of general dependency schemes in QDPLL.

The combination of QDPLL with dependency schemes is closely related to an approach which aims at exploiting tree-based quantifier structure within QDPLL [62]. However, our work generalizes observations made in [62] from quantifier trees to *arbitrary* dependency schemes.

In addition to combinations of QDPLL with dependency schemes, another contribution of this chapter is a description of clause and cube learning, also called constraint learning. From the theoretical point of view, learning methods for QDPLL were presented independently in related work [58, 78, 133, 134]. From a high level perspective, QDPLL with constraint learning resembles the well-known and widely applied DPLL algorithm for propositional logic (SAT). Implementations of DPLL are available in popular open source SAT solvers like MiniSAT [42] or PicoSAT [18]. However, implementations of clause learning for SAT cannot directly be ported to QBF. Therefore, we aim at providing an integrated view of constraint learning *as implemented in our QDPLL-based, open source solver DepQBF*.

We give an overview of QDPLL with constraint learning in Section 5.2. Starting with *boolean constraint propagation for QBF (QBCP)* in Section 5.3, we point out how to profit from dependency schemes within QDPLL. Applications of QBCP infer additional assignments from given ones, which reduces the need for explicit *branching* during the search process. As noted above, the freedom for branching, also called *decision making*, in QDPLL is limited by the linear ordering of the quantifier prefix. In order to overcome these limitations, we apply arbitrary dependency schemes for decision making in Section 5.4. We describe constraint learning in Section 5.6 and conclude with experimental results in Section 5.7.

5.2 QDPLL with Constraint Learning

In the following sections, we informally describe QDPLL with *conflict-directed clause* and *solution-directed cube learning*, called *constraint learning*, based on the presentation in [134]. Learning methods for QBF were published independently in [58, 78, 133, 134]. We consider the details of constraint learning in Section 5.6 below. For a description of DPLL with clause learning for SAT, we refer to [36, 117], for example.

5.2.1 Basics

From a very simplistic point of view, QDPLL with constraint learning successively generates (partial) assignments of the variables in a given PCNF ψ. These assignments correspond to paths in an assignment tree as defined in Section 2.2.1 which is implicitly constructed. The algorithm terminates if either a satisfying assignment tree was found or if it was proved that no such assignment tree exists. Note that assignment trees are never explicitly represented in QDPLL. Instead, only the paths in such trees are enumerated by means of assignments to variables. Although QDPLL is often presented as a recursive algorithm, it is typically implemented iteratively.

The ordering in which variables are assigned must follow the structure of dependencies in the PCNF. This corresponds to the requirement stated in Definition 2.2.3 that assignments along paths in assignment trees must be ordered. Information on dependencies can be drawn from the quantifier prefix or, as we point out later in this chapter, from more sophisticated dependency schemes computed for the PCNF. Informally, QDPLL checks the truth value of ψ under the current assignment A, that is $\psi[A]$. If the truth value can not yet be determined, then A is extended with further assignments. Otherwise, assignment A is analyzed to find out which parts of it were responsible for the resulting truth value. Depending on the outcome of that analysis, a learnt clause or learnt cube is constructed and added to ψ.

Definition 5.2.1. A *cube* is a conjunction $C_i := (l_1 \wedge \ldots \wedge l_{k_i})$ over literals. The *empty cube* is an empty conjunction and is denoted by $\emptyset$.

Similar to clauses, we assume that a cube neither contains multiple nor complementary literals of a variable, nor truth constants $\top$ and $\perp$.

Definition 5.2.2. A *constraint* is either a clause or a cube.

After a learnt constraint has been added to ψ, QDPLL retracts certain assignments in A in order to continue the search in a different search space. Thus another path of the assignment tree is entered. Informally, the purpose of constraint learning is to prevent the generation of assignments which do not contribute to overall progress of the search. Further, QDPLL with constraint learning can produce a *proof* which allows to verify its result using independent proof checkers. We briefly address proofs in Section 5.2.4 below. We may think of QDPLL with constraint learning like presented in this chapter as an algorithm which guides the search for a proof by successive generation of assignments. For DPLL-based solvers with clause learning, such view was suggested in [70], for example.

QDPLL with constraint learning does not directly operate on a given PCNF $\psi := Q_1B_1 \dots Q_nB_n.\ \phi$ but on a formula with specific structure. The idea is to extend the CNF-part ϕ of ψ in order to represent learnt clauses and learnt cubes in addition to original clauses.

Definition 5.2.3 ([134]). Given a PCNF

$$\psi := Q_1B_1 \dots Q_nB_n.\ \phi$$

with CNF-part ϕ. Let $\phi_{OCL} := \phi$. The QBF

$$\psi' := Q_1B_1 \dots Q_nB_n.\ (\phi_{OCL} \wedge \phi_{LCL}) \vee \phi_{LCU}$$

is represented as *augmented CNF (ACNF)*, where ϕ_{OCL} and ϕ_{LCL} are conjunctions over original and learnt clauses, respectively, and ϕ_{LCU} is a disjunction over learnt cubes.

Alternatively, a formula in ACNF is called *extended QBF* [58]. Given a PCNF ψ, the ACNF ψ' by Definition 5.2.3 is satisfiable if and only if ψ is satisfiable. This is due to properties of learnt clauses and learnt cubes which are generated by *Q-resolution*. In Section 5.6 below, we address Q-resolution and the construction of learnt constraints. The semantics of PCNFs based on assignment trees from Section 2.2.1 can be extended to ACNFs. In Definition 2.2.4 on page 13, assignments along paths in satisfying assignment trees must satisfy all the clauses of a PCNF. Additionally, an ACNF is satisfied if at least one cube is satisfied under the assignment along a path.

During the search, learnt clauses and cubes are added to ϕ_{LCL} and ϕ_{LCU}, respectively. Initially, subformulae ϕ_{LCL} and ϕ_{LCU} are empty. Different from the approach in [134], we do *not* consider to learn constraints containing complementary literals (called *long-distance resolution*). Instead, our description of the learning procedure is based on [61]. We consider constraint learning in detail in Section 5.6 below.

```
State qdpll ()
  while (true)
    State s = qbcp ();
    if (s == UNDET)
      // Make decision.
      v = select_dec_var ();
      assign_dec_var (v);
    else
      // Conflict or solution.
      // s == UNSAT or s == SAT.
      btlevel = analyze_leaf (s);
      if (btlevel == INVALID)
        return s;
      else
        backtrack (btlevel);
```

```
DecLevel analyze_leaf (State s)
  R = get_initial_constraint (s);
  // s == UNSAT: 'R' is empty clause.
  // s == SAT: 'R' is sat. cube...
  // ..or new cube from assignment.
  while (!stop_res (R))
    p = get_pivot (R);
    R' = get_antecedent (p);
    R = constraint_res (R, p, R');
  add_to_formula (R);
  return get_asserting_level (R);
```

Figure 5.1: Pseudo-code of QDPLL with conflict-directed clause and solution-directed cube learning [58, 78, 134]. Code blocks are indicated by indentation level.

5.2.2 Generation of Assignments

Figure 5.1 shows a high-level pseudo-code of QDPLL with constraint learning. The algorithm consists of two major parts. Function `qdpll` successively generates assignments as described above and checks the truth value of the given ACNF ψ to be evaluated. Constraint learning is performed in function `analyze_leaf`.

An important approach is the *propagation* of *implications*, which is carried out in function `qbcp`. Given an ACNF ψ and an assignment A, an implication is an additional assignment to a variable which is not yet part of A and which can be inferred from the formula $\psi[A]$. Thus QDPLL considers the formula $\psi[A]$ under the current assignment A each time A was modified. Propagation is the iterative inference of implications. This process in QDPLL is called *quantified boolean constraint propagation (QBCP)*. We explicitly distinguish QBCP from *boolean constraint propagation (BCP)* which is commonly applied in SAT solvers because QBCP and BCP typically rely on different inference rules. In Section 5.3 below, we consider common QBF-specific inference rules implemented in `qbcp`. QDPLL is complete without the rules in QBCP, that is it always terminates even if no implications are inferred. However, in general QBCP is able to improve the performance of QDPLL. Apart from propagating implications, `qbcp` evaluates the truth value of ψ under the current assignment A.

Definition 5.2.4. Given an ACNF ψ and an assignment A, the *formula* $\psi[A]$ is obtained similarly as defined in Section 2.2.1 by substituting truth constants for literals of assigned variables and performing simplifications. Additionally, *empty constraints* are handled as follows. If there is a clause

$C \in \psi[A]$ such that $C[A] = \emptyset$ then C is replaced by $\bot$ in $\psi[A]$. If there is a cube $C \in \psi[A]$ such that $C[A] = \emptyset$ then C is replaced by $\top$ in $\psi[A]$. All truth constants that were introduced this way are eliminated subsequently in $\psi[A]$.

Definition 5.2.5. Given an ACNF ψ and a (partial) assignment A, the *state* of ψ under A is defined as follows. If $\psi[A] = \bot$ then A is a *conflicting* assignment, also called a *conflict*, and ψ is *falsified* under A. If $\psi[A] = \top$ then A is a *satisfying* assignment, also called a *solution*, and ψ is *satisfied* under A. If A is neither conflicting nor satisfying then A is an *inconclusive* assignment and ψ is *undetermined* under A. We may drop "... under A" from definitions if assignment A is clear from the context.

Definition 5.2.5 is based on the current assignment only and ignores the QBF-specific rule of *constraint reduction* to be introduced in Section 5.3. Given an assignment, constraint reduction allows to ignore certain literals of universal variables in clauses and existential variables in cubes. This way, it might be be found out earlier than by Definition 5.2.5 that some assignment A cannot be part of a PCNF-model of ψ (see also Definition 5.3.7 below).

Function `qbcp` in Figure 5.1 propagates implications until saturation and checks the truth value of the ACNF under the current assignment A. If A is neither conflicting nor satisfying (`s == UNDET`) then some variable x is assigned as the next *decision* in function `select_dec_var`. This process is called *decision making* or *branching*. Different from decision making in DPLL for SAT, the choice of variables for decisions is not arbitrary in QDPLL. With respect to the current assignment A, a variable x can be assigned as decision only if all variables where x depends on are assigned in A already. For example, when relying on the quantifier prefix, then x can be assigned as decision only if all variables which are differently quantified to the left of x are assigned. This policy corresponds to decision making in classical descriptions of QDPLL like [31, 57]. In Section 5.4 below, we consider decision making by means of arbitrary dependency schemes which are represented by compact dependency graphs. Note that assignments made as implications by inference rules during QBCP do not have to respect dependencies.

Decision Levels and Trail Levels

Decisions are numbered ascendingly by *decision levels*, starting at decision level one. Once a variable x with decision level $dl(x)$ has been selected by `select_dec_var`, the value to be assigned is typically chosen according to certain heuristics in `assign_dec_var`. Given the current assignment A, assigning x to *true* or *false* produces a new, extended assignment $A' := A \cup \{x\}$ or $A' := A \cup \{\neg x\}$, respectively. All implications y that can be inferred from the current assignment A' are propagated in turn by `qbcp`, where

$dl(y) := dl(x)$. Hence, implications get the same decision level as the most recent decision or decision level zero if no decision has been made before.

In addition to decision levels, every single assignment of a variable, regardless of whether it was made as a decision or implication, is numbered chronologically starting from zero in the order of assignments made. This numbering is called the *trail level* $tl(x)$ of an assigned variable x. Different from decision levels, the trail level of every assigned variable is unique. We regard assignments $A := \{l_0, l_1, \ldots, l_{m-1}\}$ to be sorted chronologically by trail levels, where i is the trail level of assignment $l_i \in A$. Trail levels are related to the following property of assignments generated by QDPLL.

Definition 5.2.6. Given an assignment $A := \{l_0, l_1, \ldots, l_{m-1}\}$ sorted ascendingly by trail levels 0 to $m - 1$ and a dependency scheme D. Assignment A is *admissible with respect to* D if and only if, for every assignment $l_i \in A$ made as decision, the following holds: given the set $V_{<i} := \{x \mid x = v(l_j), l_j \in A, j < i\}$ of variables which were assigned before l_i in A, every variable where the decision variable $v(l_i)$ depends on by D must have been assigned before $v(l_i)$ in A, that is $D^{-1}(v(l_i)) \subseteq V_{<i}$.

Note that assignments generated in QDPLL as shown in Figure 5.1 are *always* admissible by Definition 5.2.6 with respect to the dependency scheme D that was used throughout all parts of QDPLL. This property also applies to classical descriptions of QDPLL where decisions have to be made depending on the quantifier prefix, as pointed out above. Actually, admissible assignments in classical QDPLL can be described by setting $D := D^{\text{triv}}$ in Definition 5.2.6. This way, QDPLL implicitly constructs assignment trees which correspond exactly to the original Definition 2.2.1 on page 11.

Further, admissible assignments are closely related to relaxed orderings of paths in PCNF-models by Proposition 3.4.1 on page 46. Given a dependency scheme D, the assignments along every path in a PCNF-model of a formula must respect dependencies in D. If two variables are independent then they *can* be assigned as decisions in arbitrary relative order along the paths, which increases the freedom for decision making. We consider the generation of admissible assignments in the context of arbitrary dependency schemes by QBCP and decision making in Sections 5.3 and 5.4.

5.2.3 Constraint Learning

Decision making as described above extends the current assignment A if A is neither conflicting nor satisfying. Otherwise, the ACNF ψ is either satisfied or falsified under A. This situation corresponds to a leaf in the assignment tree implicitly constructed by QDPLL. The current state of ψ under A is caused by at least one clause or cube in ψ.

Definition 5.2.7. Given an ACNF ψ, let $C \in \psi$ be a clause or cube and A be a (partial) assignment. The *state* of C under A is defined as follows. If

$C[A] = \bot$ then C is *falsified* under A. If $C[A] = \top$ then C is *satisfied* under A. Otherwise, if C is neither falsified nor satisfied, then C is *undetermined* under A. We may drop "... under A" from definitions if assignment A is clear from the context.

With respect to Definition 5.2.5, falsified clauses and satisfied cubes are related to satisfying and falsifying assignments, respectively. This relation is due to the structure of ACNFs by Definition 5.2.3. Similar to Definition 5.2.5, the effects of constraint reduction are ignored in Definition 5.2.7. We present an adapted variant in Section 5.3.4 below.

Proposition 5.2.1 ([134]). *Given an ACNF ψ and a (partial) assignment A. There is a clause $C \in \psi$ such that C is falsified under A if and only if A is conflicting. If there is a cube $C \in \psi$ such that C is satisfied under A then A is satisfying.*

Note that it is not possible to have a falsified clause and a satisfied cube under the same assignment in an ACNF. This is due to the way how learnt constraints are generated as described in Section 5.6.1.

Given an assignment A which is conflicting or satisfying, certain assignments to variables in A have to be retracted in order to continue the search in a different search space. The process of retracting assignments is called *backtracking*. The notion of "backtracking" is often associated with a chronological way of retracting assignments: the truth value of the most recently assigned decision variable x is flipped and implications with trail levels larger than the one of x are retracted.

Different from that, in modern implementations of QDPLL and DPLL with learning, like in Figure 5.1, in general assignments are retracted non-chronologically. That is, depending on the learning procedure, the effects of assignments that have been made earlier than the most recent decision could be undone. This policy is called *non-chronological backtracking* or *backjumping*, see also [36, 59, 117], for example. For simplicity, we use "backtracking" to denote the non-chronological variant.

Further, in contrast to the classical notion of backtracking, the truth values of decision variables are not explicitly flipped if the learning procedure always generates *asserting* clauses and cubes. Modern SAT and QBF solvers relying on (Q)DPLL like [18, 42, 54, 84, 93] only generate asserting clauses. Informally, a learnt constraint C is asserting if QBCP is able to infer an implication from C after backtracking to a certain decision level called *asserting level*. We refer to [36, 118, 132], for example, for further details related to the generation of asserting clauses in SAT solvers. In Section 5.6 below, we focus on the generation of asserting constraints in QDPLL.

Learnt constraints are generated in function `analyze_leaf` in Figure 5.1. The idea is to apply *Q-resolution* to derive new learnt constraints, called *resolvents*, with respect to the current conflicting or satisfying assignment. Q-resolution is the QBF-specific variant of the resolution operation for propositional logic. Resolution is also the core of clause learning in SAT solvers. In addition to clauses, cubes are learnt in QDPLL based on *term resolution* steps [61] involving cubes which have been learnt previously. Term resolution for cubes can be regarded to be dual to Q-resolution for clauses. For simplicity, we use "Q-resolution" to denote either Q-resolution over clauses or term resolution over cubes. We introduce Q-resolution formally in Section 5.6 below. According to our experimental results in Section 5.7, Q-resolution is able to generate shorter learnt constraints in the context of the standard dependency scheme D^{std} compared to the prefix ordering of PCNFs.

Learning from Conflicts

If the current assignment A is conflicting (`s == UNSAT`) then the ACNF contains at least one falsified clause. Function `get_initial_constraint` returns a reference to one such clause R which is called *initial constraint.* Given the initial constraint, new clauses are constructed using Q-resolution. In function `constraint_res`, clause R is resolved with the *antecedent clause* R' of an existential variable p which occurs in R and which was assigned as an implication in the current assignment A (see also Section 5.3.2 below). Variable p is the *pivot* of the current Q-resolution step and function `get_antecedent` returns its antecedent clause. The selection of pivots from the current constraint R by function `get_pivot` is not arbitrary. It depends on the current assignment. Care has to be taken in order to avoid the generation of constraints which contain complementary literals. We address the details of pivot selection in Section 5.6.1. Function `stop_res` stops the resolution process if the current clause R is asserting. The generated clause R is added to the set ϕ_{LCL} of learnt clauses in the ACNF. Function `get_asserting_level` computes the asserting level d such that, after backtracking to d, QBCP can infer a new implication from R.

Learning from Solutions

If the current assignment A is satisfying (`s == SAT`) then either the ACNF contains at least one satisfied learnt cube or all clauses in the ACNF are satisfied. The idea is to produce a learnt cube by Q-resolution starting from an initial constraint. If there is a satisfied learnt cube R in the ACNF then R is the initial constraint. Otherwise, an initial constraint is computed by selecting a subset $A' \subseteq A$ of the assignment A such that A' is satisfying.

Definition 5.2.8 ([134])**.** Given an ACNF $\psi := Q_1B_1 \dots Q_nB_n.\ (\phi_{OCL} \wedge \phi_{LCL}) \vee \phi_{LCU}$ and a (partial) assignment A as generated by QDPLL like in Figure 5.1. A *cover set* $A' \subseteq A$ is a subset of A such that $A' \cap C_i \neq \emptyset$ for all original clauses $C_i \in \phi_{OCL}$.

A cover set A' satisfies at least one literal in each *original* clause. Due to properties of ACNFs, it is not necessary to consider learnt constraints in Definition 5.2.8 to make A' satisfying. Note that there might be multiple cover sets for a given assignment A and set of original clauses.

The cover set A' constitutes the initial constraint R. Apart from obtaining initial constraints, learnt cubes are generated dually to learnt clauses. Q-resolution is applied to antecedent *cubes* of *universal* pivot variables. The process stops if the current cube R is asserting and R is added to the set ϕ_{LCU} of learnt cubes in the ACNF. Finally, the asserting level is computed and returned to algorithm `qdpll`.

After backtracking to the asserting level d and unassigning all variables with decision levels larger than d in function `backtrack`, QBCP is carried out again. A new implication is inferred by `qbcp` from the previously generated learnt constraint, possibly in addition to further implications.

5.2.4 Q-Resolution Proofs

Algorithm `qdpll` in Figure 5.1 terminates (`btlevel == INVALID`) if and only if Q-resolution produces an empty clause or an empty cube in function `analyze_leaf`. In this case, the learning procedure of QDPLL completed a Q-resolution proof of unsatisfiability or satisfiability of the given ACNF.

Definition 5.2.9 (terminology adapted from Section 2.3 of [9])**.** Given an ACNF $\psi := Q_1B_1 \dots Q_nB_n.\ (\phi_{OCL} \wedge \phi_{LCL}) \vee \phi_{LCU}$. The sequence $P := (C_1, C_2, \dots, C_m)$ of clauses is a *Q-resolution proof of unsatisfiability* of ψ if $C_m = \emptyset$ and $C_i \in (\phi_{OCL} \cup \phi_{LCL})$ for all $C_i \in P$ such that if $C_i \in \phi_{LCL}$ then C_i can be derived by Q-resolution from two clauses $C_j, C_k \in P$ where $j, k < i$.

Definition 5.2.10 (terminology adapted from Section 2.3 of [9])**.** Given an ACNF $\psi := Q_1B_1 \dots Q_nB_n.\ (\phi_{OCL} \wedge \phi_{LCL}) \vee \phi_{LCU}$. The sequence $P := (C_1, C_2, \dots, C_m)$ of cubes is a *Q-resolution proof of satisfiability* of ψ if $C_m = \emptyset$ and $C_i \in \phi_{LCU}$ for all $C_i \in P$ such that C_i is either a cover set or C_i can be derived by Q-resolution from two cubes $C_j, C_k \in P$ where $j, k < i$.

For simplicity, we write "proof" if the result of QDPLL is clear from the context. Note that in proofs P by Definitions 5.2.9 and 5.2.10 we do not assume any particular order for the generation of the constraints. That is, for $C_i \in P$ and $C_j \in P$ where $i < j$, the learning procedure of QDPLL as in Figure 5.1 might have derived C_j before C_i.

Proposition 5.2.2. *Given an ACNF ψ. If QDPLL with constraint learning as shown in Figure 5.1 derives the empty constraint then there is a proof by Definitions 5.2.9 or 5.2.10, respectively.*

After termination of QDPLL with constraint learning, a proof P for an ACNF by Proposition 5.2.2 can trivially be obtained from the sets of clauses or cubes, respectively: let $P := \phi_{OCL} \cup \phi_{LCL}$ for unsatisfiable ACNFs or $P := \phi_{LCU}$ for satisfiable ones. However, not all constraints in $\phi_{OCL} \cup \phi_{LCL}$ or ϕ_{LCU} might be necessary to derive the empty constraint.

Proposition 5.2.2 asserts the correctness of QDPLL with learning based on the correctness of Q-resolution [27, 61, 134]. The other direction, which amounts to completeness, also holds. If a PCNF is (un)satisfiable then QDPLL with learning will derive the empty constraint.

A comprehensive treatment of work related to proofs for QBFs is out of scope of this work. An overview of the topic is given in [94]. Proofs by Definitions 5.2.9 and 5.2.10 allow to verify the result of QDPLL based on the underlying proof system of Q-resolution. For example, if a formula is unsatisfiable and QDPLL returns a proof P then it can be checked whether the empty clause indeed can be derived from the clauses in P. This approach was considered in [130] and in a variant of the QuBE solver [94]. However, the proof P does not contain information on concrete values that have to be chosen for universal variables of a PCNF in order to explain unsatisfiability with respect to assignment trees. Dually, values for existential variables in satisfiable PCNFs explain satisfiability. Representations of values for variables are called *certificates*. As an alternative to merely checking Q-resolution proofs, Skolem functions [121] were suggested to represent certificates of a formula [11, 74]. A uniform approach combining Q-resolution proofs and certificates allows to construct a certificate for a PCNF from a given Q-resolution proof of (un)satisfiability [8]. This approach was implemented and evaluated using our QDPLL-based solver DepQBF [95, 105].

In addition to QDPLL with learning as shown in Figure 5.1, we implemented optimizations in our QBF solver DepQBF [84] presented in Section 5.6.2. These approaches are inspired by modern SAT solvers. For example, learnt constraints are heuristically deleted to speed up QBCP. Instead of backtracking to the asserting level, the solver periodically retracts *all* assignments made so far and thus restarts the search process. Our experimental analysis in Section 5.7 demonstrates the potential of these optimizations.

At the beginning of this chapter, we argued that QDPLL with constraint learning could profit from dependency schemes which refine the trivial dependency scheme given by the quantifier prefix. However, when considering Figure 5.1, dependency information is not explicitly used, although it is crucial to respect dependencies within QDPLL as pointed out.

In the following sections, we take a closer look on the core parts such as QBCP, decision making and constraint learning. We point out how these parts can profit from more refined dependency information than quantifier prefixes. Thereby, we obtain a generalization of QDPLL with constraint learning to *arbitrary* dependency schemes. The overall picture of the algorithm does not change. We consider a dependency scheme as an additional parameter passed to QDPLL. Compact dependency graphs from the previous chapter allow for efficient applications of the generalization of QDPLL.

5.3 QBCP

In this section, we focus on the details of *quantified boolean constraint propagation (QBCP)* which is carried out in function qbcp in Figure 5.1. Given an ACNF and an assignment A, the purpose of QBCP is to infer implications, that is additional assignments not yet being part of A by certain inference rules. This way, the number of decisions can be reduced which in turn might increase the performance of QDPLL. We report on the results of a related experiment in Section 5.7.

We introduce *universal* and *existential reduction*, also called *constraint reduction*, and rules for the detection of *unit literals* and *pure literals*. This set of rules is standard in QBF literature and it is part of QBCP in most QDPLL-based solvers [30, 52, 54, 60, 84]. In state-of-the-art SAT solvers, typically only unit literal detection is carried out in boolean constraint propagation (BCP). Apart from a variant based on tree-like dependency structure [62], usually QBCP is defined with respect to the linear ordering of quantifier prefixes, that is the trivial dependency scheme D^{triv}. We point out how to generalize the rules applied in QBCP from quantifier prefixes to *arbitrary* dependency schemes. Experimental results presented in Section 5.7 show that QBCP based on dependency schemes which refine D^{triv} infers more implications than classical variants based on D^{triv}.

In the following, we assume that we are given the ACNF ψ for a PCNF to be solved by QDPLL with constraint learning like in Figure 5.1. Further, we consider $\psi[A]$ at an arbitrary point of time during the solving process with respect to the current assignment A. That is, learnt constraints might have been added to the ACNF. Originally, the rules of constraint reduction and detection of unit and pure literals were introduced for PCNF. However, we focus on ACNF as the representation used in practice.

5.3.1 Constraint Reduction

As pointed out in Section 5.2.3 above, Q-resolution is applied to generate learnt constraints in QDPLL. Different from propositional resolution as used in SAT solvers, resolvents obtained by Q-resolution in general can be simplified further by deleting certain literals. Actually, Q-resolution as originally introduced for clauses [27] differs from propositional resolution exactly in that additional reduction step.

Recall notation $\prec$ and $\prec_D$ from Definition 3.4.7 on page 45. Further, by Definition 5.2.4, $\psi[A]$ is the ACNF ψ simplified under the assignment A by fully eliminating truth constants.

Definition 5.3.1. Given an ACNF ψ, a constraint $C \in \psi$ and $Q \in \{\forall, \exists\}$, $L_Q(C) := \{l \in C \mid q(l) = Q\}$.

Definition 5.3.2 ([27, 30, 61]). Given an ACNF

$$\psi := Q_1 B_1 Q_2 B_2 \ldots Q_n B_n.\ (\phi_{OCL} \wedge \phi_{LCL}) \vee \phi_{LCU},$$

a constraint $C \in \psi$ such that there is no variable x with $\{x, \neg x\} \subseteq C$ and a dependency scheme D defining the relation $\prec_D$. Let $Q := \forall$ and $\overline{Q} := \exists$ if C is a clause and $Q := \exists$ and $\overline{Q} := \forall$ if C is a cube. The application of *constraint reduction* to C with respect to D produces the constraint

$$CR_D(C) := C \setminus \{l \in L_Q(C) \mid \forall l' \in L_{\overline{Q}}(C) : l \not\prec_D l'\}.$$

We may write CR instead of CR_D if D is clear from the context. Constraint reduction is generalized from constraints to ACNFs:

$$CR(\psi) \text{ is the ACNF such that if } C \in \psi \text{ then } CR(C) \in CR(\psi).$$

Additionally, the quantifier prefix of $CR(\psi)$ is simplified by removing quantified occurrences of variables which do no longer occur in the quantifier free part of $CR(\psi)$ after reduction.

Constraint reduction by Definition 5.3.2 operates on clauses and cubes. Originally, this operation was introduced explicitly for clauses [27] under the names *universal reduction* or *forall reduction*. Later, it was extended to cubes [61], where reduced constraints were referred to as *minimal*. Implicitly, constraint reduction appears in early descriptions of QDPLL like [30] where literals are not removed but ignored in order to detect unit literals. We consider unit literal detection in Section 5.3.2 below.

Note that the dependency scheme D in Definition 5.3.2 is arbitrary. Actually, early definitions of constraint reduction took into account the ordering of the quantifier prefix, which is D^{triv} in our framework. Therefore, Definition 5.3.2 generalizes constraint reduction from quantifier prefixes to arbitrary dependency schemes.

Informally, constraint reduction removes literals from a constraint C which are irrelevant to satisfy or falsify C, depending on whether C is a clause or a cube. In the following, we assume that D is an arbitrary but fixed dependency scheme applied in constraint reduction by Definition 5.3.2. We prove the correctness of constraint reduction with respect to D separately for reductions of clauses and cubes in Theorems 5.3.1 and 5.3.2 below.

Theorem 5.3.1 (see also Theorem 1 in [61]). *Given an ACNF*

$$\widehat{Q}.\ (\phi_{OCL} \wedge \phi_{LCL} \wedge C) \vee \phi_{LCU}$$

where C is a clause and $\widehat{Q}$ is the quantifier prefix. Constraint reduction on clause C produces a model-equivalent ACNF:

$$\widehat{Q}.\ (\phi_{OCL} \wedge \phi_{LCL} \wedge C) \vee \phi_{LCU} \equiv_m \widehat{Q}'.\ (\phi_{OCL} \wedge \phi_{LCL} \wedge CR(C)) \vee \phi_{LCU},$$

where $\widehat{Q}'$ is obtained from $\widehat{Q}$ by deleting superfluous quantifiers.

$$\begin{tikzcd}
& r \arrow[dl] \arrow[dr] & & & & & r \arrow[dl] \arrow[dr] & \\
\neg y \arrow[d] & & y \arrow[d] & & & \neg y \arrow[d] & & y \arrow[d] \\
x & & \neg x & & & x & & x
\end{tikzcd}$$

Figure 5.2: The two possible refuting assignment trees by Definition 5.3.3 for the unsatisfiable PCNF $\psi := \exists y \forall x.\ (x \vee \neg y) \wedge (\neg x \vee y) \wedge (\neg x \vee \neg y)$.

By Theorem 5.3.1, constraint reduction on clauses in an ACNF preserves models. Every model of the original ACNF is also a model of the reduced one. In fact, reduced clauses are also satisfied by the assignments along the paths in models, that is satisfying assignment trees, of the original ACNF. However, in general it is crucial for the correctness of constraint reduction not to reduce constraints containing complementary literals.

Example 5.3.1. Given the satisfiable PCNF $\psi := \forall x.\ (x \vee \neg x)$. Constraint reduction produces the empty clause from $(x \vee \neg x)$, thus $\mathit{CR}(\psi)$ is unsatisfiable.

In order to allow for simpler correctness proofs, we introduce a particular variant of assignment trees which allows to explain *unsatisfiability* of ACNFs. Thus the existence of a satisfying assignment tree can be *refuted*.

Definition 5.3.3. Given an unsatisfiable ACNF ψ, a *refuting assignment tree* T of ψ is defined dually to a satisfying assignment tree by Definitions 2.2.3 and 2.2.4. Nodes which assign universal variables in T do not have siblings whereas there is exactly one sibling for nodes which assign existential variables. The assignments along every path in T all falsify the CNF-part and the cubes in ψ. Assignments along paths in refuting assignment trees are ordered with respect to the quantifier prefix or, as noted above in Definition 5.2.6, by some arbitrary dependency scheme.

A refuting assignment tree represents a sufficient selection of values for the universal variables in an ACNF such that *no* assignment to the depending existential variables satisfies either all the clauses or at least one cube. Note that, similar to satisfying assignment trees by Definition 2.2.4, refuting assignment trees are not necessarily unique. Figure 5.2 shows an example.

Proof of Theorem 5.3.1. Let $\psi := \widehat{Q}.\ (\phi_{OCL} \wedge \phi_{LCL} \wedge C) \vee \phi_{LCU}$ and $\psi' := \widehat{Q}'.\ (\phi_{OCL} \wedge \phi_{LCL} \wedge \mathit{CR}(C)) \vee \phi_{LCU}$. Thus ACNF ψ' was obtained from ψ by reducing clause C. We show that ψ and ψ' are model-equivalent.

If ψ is unsatisfiable then there is a refuting assignment tree T of ψ. By Definition 5.3.3, the CNF-part of ψ and all cubes are falsified by each assignment along the paths in T. Since the clause $\mathit{CR}(C) \in \psi'$ has fewer literals than $C \in \psi$, $\mathit{CR}(C)$ is also falsified under every assignment in T.

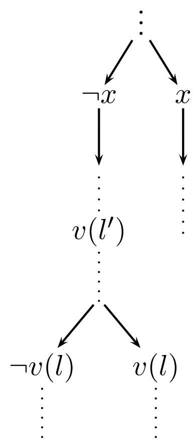


Figure 5.3: Part of a satisfying assignment tree m which illustrates the proof of Theorem 5.3.1. Variable x is universal and irrelevant. Variable $v(l)$ is universal and was reduced by constraint reduction from clause C to obtain clause $CR(C)$. A literal of the existential variable $v(l')$ satisfies both C and $CR(C)$, since $v(l')$ was assigned before $v(l)$ along the path in m.

The other constraints in ψ' are the same as in ψ. Therefore, T is a refuting assignment tree of ψ' as well and hence ψ' is unsatisfiable.

If ψ is satisfiable, let m be a satisfying assignment tree of ψ, that is $m \models \psi$. Assume that all assignments along the paths in m are complete and admissible with respect to D where D is the dependency scheme that was applied for constraint reduction. We show that also $m \models \psi'$.

By definition of satisfying assignments trees, assignments along paths in m must satisfy the quantifier-free part of ψ. We consider all assignments along paths at the leaves of m. If such an assignment satisfies a cube in ψ then that cube is also satisfied in ψ' since constraint reduction did not affect cubes in ψ. The same holds for all clauses in ψ' other than $CR(C)$ since they were not affected by constraint reduction.

Consider the clauses C and $CR(C)$ and, without loss of generality, assume that exactly one universal literal l was removed from C to obtain $CR(C)$. We can assume that, among all variables occurring in $CR(C)$, variable $v(l)$ was assigned last on the current path in m since all assignments are admissible. Figure 5.3 illustrates the situation. Further, due to Definition 5.3.2, there is no other variable v'' in C such that $v(l) \prec v''$. Therefore, clause C must be satisfied by some other existential literal l' because $v(l)$ is universal: otherwise, considering the sibling of node $v(l)$ which assigns the opposite truth value, C would no longer be satisfied. Since literal l' is existential, it cannot be removed by constraint reduction and therefore l' will also satisfy $CR(C)$. □

Theorem 5.3.2. *Given an ACNF*

$$\widehat{Q}.\ (\phi_{OCL} \wedge \phi_{LCL}) \vee (\phi_{LCU} \vee C)$$

where C is a cube and $\widehat{Q}$ is the quantifier prefix. Constraint reduction on cube C produces a model-equivalent ACNF:

$$\widehat{Q}.\ (\phi_{OCL} \wedge \phi_{LCL}) \vee (\phi_{LCU} \vee C) \equiv_m \widehat{Q}'.\ (\phi_{OCL} \wedge \phi_{LCL}) \vee (\phi_{LCU} \vee CR(C)),$$

where $\widehat{Q}'$ is obtained from $\widehat{Q}$ by deleting superfluous quantifiers.

Proof. The proof is similar to the one of Theorem 5.3.1 and dual arguments apply. Let ψ and ψ' be defined like in the proof above. That is ACNF ψ' was obtained from ψ by reducing cube C.

If ψ is satisfiable then so is ψ'. Let m be a satisfying assignment tree of ψ, that is $m \models \psi$. If an assignment along a path in m satisfies all clauses in ψ then it also satisfies the clauses in ψ' because clauses were not affected by constraint reduction. The same applies to cubes other than C in ψ. If an assignment along a path in m satisfies the cube $C \in \psi$ then all literals in C are satisfied. Thus, that assignment also satisfies the reduced cube $CR(C) \in \psi'$ which has fewer literals than C. Thus also $m \models \psi'$.

If ψ is unsatisfiable then there is a refuting assignment tree T. The CNF-part and all cubes in ψ are falsified with respect to each assignment along the paths in T. Since clauses were not affected by constraint reduction, the CNF-part of ψ' is falsified as well. This also holds for all cubes in ψ other than C. Consider cubes C and $CR(C)$. Similar to the proof of Theorem 5.3.1, we assume that exactly one existential literal l was reduced and that assignments in T are complete and admissible. Cube C must be falsified by some other literal l' in addition to l because otherwise C would be satisfied when $v(l)$ is assigned the opposite truth value in the sibling node of $v(l)$. Hence $CR(C)$ is also falsified after removing l by constraint reduction. □

Different from Definition 5.3.2, in QDPLL constraint reduction is applied to the formula $\psi[A]$ under the current assignment A. We consider the practice of constraint reduction in QDPLL as part of QBCP in Section 5.3.4.

Given an ACNF ψ, a *partial* assignment A and a constraint $C \in \psi$, C might be empty after constraint reduction in $CR(\psi[A])$ but not in $\psi[A]$. For example, if C is a clause then assignment A *cannot* be extended to a solution in this case. In fact we have $CR(\psi[A])[A] = \bot$ by Definition 5.2.4. That is, constraint reduction allows to detect conflicting or satisfying assignments implicitly by empty clauses and empty cubes, respectively. According to that, we refine Definition 5.2.5 by Definition 5.3.7 below.

As pointed out, constraint reduction was originally introduced with respect to the linear ordering of variables by quantifier prefixes. By Definition 5.3.2, we obtained a generalization to arbitrary dependency schemes.

In the forthcoming sections of this chapter, we consider how QDPLL might benefit from dependency schemes which are more refined than the trivial one arising from quantifier prefixes. For example, constraint reduction might produce shorter constraints.

Proposition 5.3.1. *Given two dependency schemes D and D' such that $D \subseteq D'$. Let CR_D and $CR_{D'}$ denote constraint reduction with respect to D and D', respectively. Given a constraint C, $CR_D(C) \subseteq CR_{D'}(C)$.*

As pointed out below, constraint reduction is a crucial part of QBCP. Due to Proposition 5.3.1, using more refined dependency schemes for constraint reduction might influence unit and pure literal detection positively.

5.3.2 Unit Literal Detection

We consider a common rule in QBCP which allows to infer *forced* assignments to variables from a given ACNF. For example, if there is a clause in a CNF which consists of only one existential literal, then that clause can only be satisfied by assigning the variable of the literal accordingly. Such literals are called *unit literals.* The rule for the detection of unit literals allows to identify forced assignments in an ACNF during QBCP. In general, unit literals can appear in clauses as well as in cubes of an ACNF.

In state-of-the-art SAT solvers, unit literals are detected during BCP. Classical definitions of unit literal detection for QBF [31] typically differ from the rule applied in SAT solvers. That difference is due to a combination of constraint reduction and unit literal detection into one single rule. We introduce unit literal detection for ACNFs separately from constraint reduction. Our definition corresponds exactly to the rule which is common for SAT solving, as far as clauses are concerned. That is, we focus on constraints in ACNFs which contain *exactly* one literal. In Section 5.3.4 below, we explicitly combine constraint reduction and unit literal detection in QBCP. This way, we obtain the same effects as the original definition of unit literal detection [31] which implicitly includes constraint reduction.

Definition 5.3.4 (see also [30, 61, 134], for example)**.** Given an ACNF ψ and constraint $C \in \psi$. Let $Q := \exists$ if C is a clause and $Q := \forall$ if C is a cube. Constraint C is *unit* if and only if $C = (l_1)$ and $q(v(l_1)) = Q$, where l_1 is called a *unit literal.* Given a constraint C, *unit literal detection*

$$UL(C) := \{l\}$$

collects the assignment $\{l\}$ from C if $C = (l_1)$ and C is unit, where $l := l_1$ if C is a clause and $l := \neg l_1$ if C is a cube. Constraint $C = \mathit{ante}(l)$ is the *antecedent constraint* of assignment $\{l\}$. Otherwise, if C is not unit then $UL(C) := \emptyset$ is the empty assignment. Unit literal detection is extended from individual constraints to sets of constraints in an ACNF ψ, that is to original clauses, learnt clauses and cubes:

$$UL(\psi) := \bigcup_{C \in \psi} UL(C).$$

In order to show the correctness of unit literal detection, proofs apply as given in [31, 61], for example. This is possible because, as pointed out above, our variant from Definition 5.3.4 does not take constraint reduction into account. Therefore, it can be considered a special case of those variants which combine unit literal detection and constraint reduction.

Unit literal detection by Definition 5.3.4 infers assignments from unit literals in an ACNF. Note that, in addition to clauses like in SAT solving, also cubes can be unit. Informally, a unit cube $C := (l_1)$ asserts that there exists a *full* assignment A' which assigns the variable of l_1 such that the formula is satisfied under A'. Assignment A' was used during cube learning to generate cube C. Unit literal detection on cubes assigns the variable of l_1 in such a way to prevent QDPLL from generating A' again. In general, assignments drawn from unit clauses prevent QDPLL from generating assignments which are obviously falsifying. Dually, obvious solutions that have been found already are prevented by assignments from unit cubes.

We briefly introduced the notion of antecedent constraints in the context of constraint learning in Section 5.2.3 and in Figure 5.1 above. In fact, antecedents used for Q-resolution during learning correspond exactly to antecedents by Definition 5.3.4, which is pointed out in Section 5.6 below.

Unit literal detection during QBCP in QDPLL always interprets the given ACNF ψ under the current assignment A. That is, different from Definition 5.3.4, $UL(\psi[A])$ is computed instead of $UL(\psi)$. Further, the rule is applied iteratively in QBCP. Given an ACNF ψ, first units are detected on ψ. ACNF ψ is simplified under the resulting assignment $UL(\psi)$ and unit are detected again on $\psi[UL(\psi)]$, as pointed out in Example 5.3.2 below.

5.3.3 Pure Literal Detection

In addition to unit literals, QBCP detects variables which either have only positive or only negative literals left in the constraints of an ACNF. Such literals are called *pure* or *monotone* [30, 60]. A truth value can be assigned depending on the quantifier type of the variable. Assignments due to pure literals are not forced because, in contrast to unit literals, they are not necessary to satisfy particular clauses, for example. Therefore, detection of pure literals is usually not carried out in state-of-the-art SAT solvers. However, assignments due to pure literals can produce additional unit literals in clauses and cubes of an ACNF, as illustrated in Example 5.3.2 below.

In order to introduce pure literals formally, we adapt the definition of occurrences of a variable, which was given in Section 2.1.2, to ACNF. Given an ACNF ψ and a literal l, $O(l) := \{C \mid C \in \psi, l \in C\}$ is the set of *occurrences* of l, that is the set of all constraints including clauses and cubes in ψ which contain literal l.

Definition 5.3.5 ([30])**.** Given an ACNF ψ, a literal l where $O(l) \neq \emptyset$ and $O(\neg l) = \emptyset$ is *pure* in ψ. The operation of *pure literal detection*

$$PL(\psi) := \bigcup \{l'\}$$

collects assignments $\{l'\}$ such that there is a literal l which is pure in ψ where $l' := l$ if $q(l) = \exists$ and $l' := \neg l$ if $q(l) = \forall$.

Correctness of pure literal detection was proved in [30] with respect to clauses in a PCNF only. However, the same arguments apply to our definition as well since including learnt cubes in the definition of the set of occurrences $O(l)$ imposes an additional restriction.

Pure literal detection by Definition 5.3.5 assigns existential variables x such that all clauses where x occurs are satisfied and literals of x are satisfied in cubes. Dually, assignments of universal variables falsify cubes and falsify literals in clauses. Therefore, assignments by pure literal detection can make constraints unit, as pointed out in Example 5.3.2 below.

Different from unit literal detection by Definition 5.3.4, there are no antecedent constraints associated with assignments by pure literal detection. As pointed out in Section 5.6 below, constraint learning relies on antecedent constraints of unit literals and does not have to deal with assignments by pure literals in any special way.

5.3.4 Putting It All Together

QBCP as described informally in Section 5.2.2 and in Figure 5.1 applies the combination of constraint reduction and detection of unit and pure literals. We call assignments that are inferred during QBCP *implications*, regardless of whether they are due to unit or pure literal detection.

Above, we introduced constraint reduction separately from unit literal detection, which is different from classical definitions. When looking at Definitions 5.3.2, 5.3.4 and 5.3.5, it turns out that only constraint reduction by Definition 5.3.2 is involved with dependency schemes. As done in Proposition 5.3.1, our separate definition allows to point out the benefits of combining constraint reduction with arbitrary dependency schemes. Apart from this generalized variant of constraint reduction, our definition of QBCP introduced below is equivalent to variants where constraint reduction and unit literal detection are combined into one single rule. Since we proved the correctness of constraint reduction with respect to arbitrary dependency schemes in Section 5.3.1 above, the correctness of QBCP when relying on that generalized variant follows right away.

Definition 5.3.6. Let ψ be an ACNF and A an assignment.

- $\mathit{QBCP}^0(\psi[A]) := \psi'[A \cup A']$ where $A' := (\mathit{UL}(\psi') \cup \mathit{PL}(\psi'))$ and $\psi' := \mathit{CR}(\psi[A])$.
- $\mathit{QBCP}^i(\psi[A]) := \mathit{QBCP}^0(\mathit{QBCP}^{i-1}(\psi[A]))$ for natural number $i > 0$.
- $\mathit{QBCP}(\psi[A]) := \mathit{QBCP}^i(\psi[A])$ for the smallest natural number i such that $\mathit{QBCP}^i(\psi[A]) = \mathit{QBCP}^{i+1}(\psi[A])$.

QBCP by Definition 5.3.6 infers additional assignments from a given one in QDPLL. For an ACNF ψ and the current assignment A generated by QDPLL, first constraint reduction is applied to $\psi[A]$ to obtain the reduced formula ψ'. Implications A' by unit and pure literal detection are inferred from ψ'. Extending the current assignment A with the set of implications A' yields a new assignment $A \cup A'$ which is used to simplify ψ' further. This process is repeated until no more implications can be found.

Constraint reduction is crucial in QBCP. The deletion of literals from constraints can enable the detection of both new unit and new pure literals. Due to Proposition 5.3.1, the use of more refined dependency schemes in constraint reduction can produce shorter constraints. This in turn might allow to identify more unit and pure literals. Further, as noted in Section 5.3.1, constraints might become empty exclusively due to constraint reduction. This property allows to generalize the state of a formula under an assignment by Definition 5.2.5 to QBCP including constraint reduction.

Definition 5.3.7. Given an ACNF ψ and a (partial) assignment A, the *state* of ψ under QBCP and A is defined as follows. If $\mathit{QBCP}(\psi[A]) = \bot$ then A is a *conflicting* assignment, also called a *conflict*, and ψ is *falsified* under A. If $\mathit{QBCP}(\psi[A]) = \top$ then A is a *satisfying* assignment, also called a *solution*, and ψ is *satisfied* under A. If A is neither conflicting nor satisfying then A is an *inconclusive* assignment and ψ is *undetermined* under A.

Notation $\mathit{QBCP}(\psi[A]) = \bot$ and $\mathit{QBCP}(\psi[A]) = \top$ is well-defined due to Definition 5.2.4. Empty constraints produced by constraint reduction are replaced by truth constants which in turn are eliminated in the formula $\mathit{QBCP}(\psi[A])$. Function `qbcp` in Figure 5.1 typically implements QBCP and determines the state of the formula according to Definitions 5.3.6 and 5.3.7, respectively. Proposition 5.2.1 is adapted to QBCP accordingly.

It is desirable to maximize the number of implications during QBCP. In general, the more implications are inferred, the fewer decisions have to made by QDPLL. Each of the three rules applied in Definition 5.3.6 contributes to the inference of implications, as illustrated in the following example.

Example 5.3.2. Given the ACNF ψ which, for simplicity, only contains clauses:

$$\psi := \exists e_1 \forall a_2 \exists e_3, e_4.\ (e_1 \vee a_2 \vee e_3 \vee e_4) \wedge (e_1 \vee a_2 \vee \neg e_4) \wedge (\neg e_1 \vee e_3 \vee \neg e_4) \wedge (\neg a_2 \vee \neg e_3).$$

Assume that the current assignment is $A := \{e_4\}$. We consider ψ under A:

$$\psi[A] = \exists e_1 \forall a_2 \exists e_3.\ (e_1 \vee a_2) \wedge (\neg e_1 \vee e_3) \wedge (\neg a_2 \vee \neg e_3).$$

Neither the unit nor the pure literal rule is applicable to $\psi[A]$. Constraint reduction with respect to D^{triv} on $\psi[A]$ deletes the universal literal a_2 from clause (e_1, a_2) to obtain the reduced clause (e_1):

$$CR(\psi[A]) = \exists e_1 \forall a_2 \exists e_3.\ (e_1) \wedge (\neg e_1 \vee e_3) \wedge (\neg a_2 \vee \neg e_3).$$

Variable a_2 has only one negative occurrence left and is pure in $CR(\psi[A])$. Pure literal detection assigns variable a_2 to *true* to obtain the new assignment $A' := \{a_2\}$. Further, clause (e_1) is unit in $CR(\psi[A])$ and the new assignment A' is extended by setting variable e_1 to *true*, which yields $A' := \{a_2, e_1\}$. No further implications can be inferred from $CR(\psi[A])$. Formula $CR(\psi[A])$ is simplified using the new, current assignment $A \cup A'$:

$$CR(\psi[A])[A \cup A'] = \exists e_3.\ (e_3) \wedge (\neg e_3).$$

Assume that unit literal detection infers the assignment $A'' := \{e_3\}$ from $CR(\psi[A])[A \cup A']$. Simplifications under A'' produce the empty clause:

$$CR(\psi[A])[A \cup A'][A''] = \bot.$$

QBCP without constraint reduction could miss the inference of both unit and pure literals. In Example 5.3.2 above, variable a_2 becomes pure only after constraint reduction was applied. Similarly, the clause (e_1, a_2) becomes unit only by constraint reduction. Dual effects on cubes can be observed in general. Further, pure literal detection is crucial for QBCP to detect additional unit clauses. In Example 5.3.2, clause $(\neg a_2, \neg e_3)$ is unit only after variable a_2 becomes pure. Dually, cubes can become unit by assigning existential variables by pure literal detection. Therefore, none of the three rules of constraint reduction, unit and pure literal detection can be omitted from QBCP without harming its ability to infer implications.

Note that, due to lack of decisions, new assignments generated by unit and pure literal detection in QBCP according to Definition 5.3.6 are trivially admissible by Definition 5.2.6 with respect to the same dependency scheme D that was used for constraint reduction (we require that one and the same D is used in all parts of QDPLL). As noted, variables assigned by unit and pure literal rule during QBCP get trail levels in chronological fashion. Additionally, we distinguish assignments to variables as follows.

Definition 5.3.8. Given an assignment A generated by QDPLL, the *assignment mode* of a literal $l \in A$ assigned by A is $am(l) := D$ if l was assigned as decision, $am(l) := U$ if l was assigned by unit literal rule and $am(l) := P$ if l was assigned by pure literal rule.

Definition 5.3.9. Given a decision level $i \in \{1, 2, \ldots, n\}$, the unique variable x with $dl(x) = i$ and $am(x) = D$ is the *decision variable at decision level i*.

Implementation

Different from Definition 5.3.6, in our QBF solver DepQBF QBCP is not implemented recursively but iteratively. Literals reduced by constraint reduction with respect to the current assignment are never explicitly deleted from constraints but disabled instead. This way, reduced literals do not have to be introduced again during backtracking.

Additionally, the application of constraint reduction is merged with unit literal detection. For that purpose, the approach of two-literal watching [93], which is commonly applied in modern SAT solvers, can be extended to QBF [52]. The following description was published previously in [84].

Two unassigned literals l_1 and l_2 are watched in each constraint C under the following restrictions. If C is a clause, then either (1) $q(l_1) = q(l_2) = \exists$ or (2) $q(l_1) = \forall$, $q(l_2) = \exists$ and $l_1 \prec_D l_2$ where D is the dependency scheme applied in QDPLL. Otherwise, C is a cube and either (1) $q(l_1) = q(l_2) = \forall$ or (2) $q(l_1) = \exists$, $q(l_2) = \forall$ and $l_1 \prec_D l_2$.

If variable x is assigned in QDPLL then the watchers of all constraints C where a literal of x is watched will be updated under the restrictions stated above. That is, watchers must be set such that either case (1) or case (2) holds. Whenever a current watcher already satisfies clause C (or falsifies cube C) then no update is made. If a clause (cube) C contains exactly one unassigned existential (universal) literal then C is unit. If clause (cube) C does not contain unassigned existential (universal) literals then C is falsified (satisfied). During watcher update of C, constraint reduction is applied on the fly to C by ignoring any universal (existential) literals in C which are unassigned or *false* (*true*). Dependency checking during watcher updates, which is needed in case (2) only, can be carried out efficiently as described in Section 5.5 below.

Pure literal detection in QBCP considers the given ACNF ψ under the current assignment A. Since learnt constraints have to be taken into account by Definition 5.3.5, it might become expensive to search for pure literals in ψ explicitly if the number of learnt constraints becomes large. In order to overcome this problem in implementations, constraints can be *watched* [52]. A literal of variable x is not pure as long as x has at least one positive and one negative occurrence in $\psi[A]$. For each variable a, one negative and one positive occurrence $C_{\neg x}$ and C_x of x is watched, respectively. Whenever either $C_{\neg x}$ or C_x are satisfied (or falsified, if $C_{\neg x}$ or C_x are cubes) under a modified assignment $A' \neq A$ then new watched constraints are selected for x. If this is not possible because x does not have positive or negative occurrences in $\psi[A']$ then x is pure.

In addition to constraint watching, learnt constraints could be ignored during pure literal detection. That is, only the set ϕ_{OCL} of original clauses of the ACNF $\psi := (\phi_{OCL} \wedge \phi_{LCL}) \vee \phi_{LCU}$ is taken into consideration. Variables might be pure with respect to ϕ_{OCL} but not with respect to ψ if learnt constraints are included. This approach might produce *spurious pure literals* [60], which have to be treated in a special way whenever they are involved in the detection of new unit literals or empty constraints. In our QBF solver DepQBF we implemented both constraint watching and spurious pure literals [84]. We refer to [52, 60, 84] for further details.

As noted above, assignments made during QBCP are always admissible. In order to guarantee admissible assignments not only by QBCP but also by decision making, that is in entire QDPLL, variable dependencies have to be taken into account as follows.

5.4 Decision Making

In classical descriptions of QDPLL [30] which rely on the variable ordering given by the quantifier prefix, decisions have to be made with respect to that linear ordering. That is, function `select_dec_var` in Figure 5.1 must select decision variables "from left to right", which corresponds to the natural ordering by variable dependencies in the quantifier prefix. We extend this approach from the prefix-based trivial dependency scheme to arbitrary ones. Thereby, QDPLL can benefit from more freedom in decision making because the linear prefix ordering is relaxed, as shown in Example 3.1.3 on page 25. Instead of the linear prefix ordering, decisions are made with respect to the partial ordering defined by the dependency scheme.

The requirement of respecting variable dependencies in decision making is necessary in QDPLL to generate assignments which are admissible by Definition 5.2.6. This way, the assignment tree implicitly constructed by QDPLL complies with semantics of PCNF (and also ACNF) from Section 2.2 and, when generalized to arbitrary dependency schemes, by Proposition 3.4.1. Assignment which are not admissible might yield unsound results, as pointed out in Example 3.1.1 on page 24.

Implications assigned during QBCP are exempted from the requirements stated in Definition 5.2.6. Assignments to some variable x by unit or pure literal rule can always be made even if not all variables where x depends on are assigned already. Note that Definitions 5.3.4 and 5.3.5 do not refer to dependency schemes. Therefore, the extension of an admissible assignment by implications detected in QBCP will again produce an admissible assignment. The problem of generating admissible assignments in QDPLL is limited to decision making only. In the following, we assume that an arbitrary dependency scheme D is applied for QBCP and decision making.

QDPLL as illustrated in Figure 5.1 generates assignments starting from the empty assignment $A := \emptyset$, which is trivially admissible with respect to D. We have to ensure that every decision selected by function `select_dec_var` results in a new admissible assignment. From a theoretical point of view, we adopt Definition 5.2.6 to obtain a sufficient condition as follows.

Definition 5.4.1. Given a dependency scheme D, an admissible assignment A generated by QDPLL and a variable x. Let $V_A := \{z \mid z = v(l), l \in A\}$ be the set of variables assigned in A. A variable x is a *decision candidate* with respect to D and A if and only if $D^{-1}(x) \subseteq V_A$. If x is a decision candidate then x is *enabled* under A, otherwise x is *disabled.*

By Definition 5.4.1, every variable y where a decision candidate x depends on must be assigned already. Checking the condition explicitly might be expensive in practice if the set $D^{-1}(x)$ is large.

Therefore, our goal is to maintain the *set DC of decision candidates* incrementally based on the current assignment A. Initially, $A = \emptyset$ and all variables x where $D^{-1}(x) = \emptyset$ are decision candidates. For example, in terms of quantifier prefixes likes in classical QDPLL, this situation corresponds to the left-most quantifier block in an ACNF. Each additional assignment to a variable *potentially* enables some variable x. Variable x is enabled if all variables in $D^{-1}(x)$ are assigned. When it comes to decision making, function `select_dec_var` must select variables from the set DC only. If assignments are retracted during backtracking in function `backtrack`, then a candidate $x \in DC$ must be removed from this set if there is at least one variable in $D^{-1}(x)$ which was unassigned, thus disabling x.

In order to maintain the set DC of decision candidates incrementally, we present an approach relying on the augmented compressed dependency graph $G_{\approx,\subseteq}(D)$ for D by Definition 4.2.8 on page 60. First, the idea is not to check the condition of decision candidates by Definition 5.4.1 explicitly for each variable, but to focus on equivalence classes of variables as part of graph $G_{\approx,\subseteq}(D)$ instead. Second, the set DC of decision candidates does not have to be updated each time the current assignment A is modified. We want to check the condition only if modifying A potentially changes the set DC. For that purpose, equivalence classes are applied as follows.

Given variables x and x', if $D^{-1}(x) = D^{-1}(x')$ then also $x \approx_\uparrow x'$ by Definition 4.2.4. Since x and x' depend on the same set of variables, x becomes a decision candidate if and only if x' becomes one. Thus, the whole class $[x]_\uparrow$ becomes enabled.

Further, if some variable y is assigned then that single assignment does *not* enable some variable $x \in D(y)$ if there is an unassigned variable y' with $y' \neq y$ where x depends on. Consider the classes $[y]_\downarrow$ induced by relation $\approx_\downarrow$ according to Definition 4.2.3, which checks equality of the sets $D(y)$ of dependencies. A variable x in $D(y)$ *might* become enabled as soon as all variables in the class $[y]_\downarrow$ are assigned. Note that x might depend on other variables in addition to $[y]_\downarrow$. Thus having all variables in class $[y]_\downarrow$ assigned is a necessary but not a sufficient condition to enable new variables in $D(y)$. In practice, we count the number of assigned variables in classes $[y]_\downarrow$.

In addition to equivalence classes given by relations $\approx_\uparrow$ and $\approx_\downarrow$, subset edges in the graph $G_{\approx,\subseteq}(D)$ can be used to reduce the amount of work that has to be carried out in order to update the set DC of decision candidates. In the following, we describe how to maintain the set DC incrementally and lazily based on modifications of the current assignment A by QBCP, decision making and backtracking. For that purpose, we assume that the dependency scheme D that is used in QDPLL is represented as the augmented compressed dependency graph $G_{\approx,\subseteq}(D)$ by Definition 4.2.8.

5.4.1 Maintaining Decision Candidates

Each time some variable y is assigned during QBCP or decision making, potentially some other variable $x \in D(y)$ is enabled. QDPLL starts with the empty assignment $A = \emptyset$ and the set $DC := \{x \mid D^{-1}(x) = \emptyset\}$ of variables which do not depend on other ones as decision candidates. Given the current assignment A and the current set DC of decision candidates, assume that variable y is unassigned in A. Let A' be a new assignment obtained from A by assigning y. We present an algorithm to obtain the new set DC' of decision candidates with respect to A' based on the graph $G_{\approx,\subseteq}(D)$, which includes subset edges.

1. Let $DC' := DC$.
2. Given the assigned variable y, consider the class $[y]_\downarrow$ of variables which have the same set of dependencies as y.
3. If there is an unassigned variable y' in class $[y]_\downarrow$ then no additional variables are enabled after y was assigned. No additional work has to be done.
4. Otherwise, all variables in class $[y]_\downarrow$ are assigned. Consider the set $REF := \{[x]_\uparrow \mid ([y]_\downarrow, [x]_\uparrow) \in E_d\}$ of classes which are referenced from class $[y]_\downarrow$ by a dependency edge and possibly from other classes as well. These classes have to be checked as follows.
5. For all classes $[x]_\uparrow \in REF$:
   - 5.1. If class $[x]_\uparrow$ is referenced by a dependency edge from a class $[y']_\downarrow$ such that $[y']_\downarrow$ contains an unassigned variable then continue with the next class in REF. No additional work has to be done.
   - 5.2. Otherwise, given the class $[x]_\uparrow$, all variables in $D^{-1}(x)$ are assigned and the set DC is updated as follows.
   - 5.3. For all classes $[x']_\uparrow$ which are reachable from $[x]_\uparrow$ over zero or more subset edges:
     - 5.3.1. If class $[x']_\uparrow$ is referenced by a dependency edge from a class $[y'']_\downarrow$ such that $[y'']_\downarrow$ contains an unassigned variable, then continue with the next class. No additional work has to be done.

5.3.2. Otherwise, given the class $[x']_\uparrow$, all variables in $D^{-1}(x)$ are assigned and the new set DC' is updated by $DC' := DC' \cup \{[x']_\uparrow\}$.

6. Finally, set DC' contains all decision candidates with respect to the new assignment A'.

The algorithm described above keeps track of the number of unassigned variables in a class with respect to relation $\approx_\downarrow$. The only situation where potentially any work has to be done is when that number becomes zero due to assigning variable y. In this case, at least the class $[y]_\downarrow$ does not prevent other variables in the set $D(y)$ from becoming enabled. Note that if assigning y enables any variable x then $x \in D(y)$. Therefore, it is sufficient to check the set $D(y)$ for new decision candidates. Classes $[x']_\uparrow$ of variables in the set $D(y)$ are visited by traversing subset edges in graph $G_{\approx,\subseteq}(D)$. The traversal implicitly performs a check whether, given a visited class $[x']_\uparrow$, all variables in $D^{-1}(x')$ are assigned. If so then the whole class $[x']_\uparrow$ is enabled.

As an optimization, subset edges in graph $G_{\approx,\subseteq}(D)$ allow to skip all successors of visited classes $[x']_\uparrow$ in step 5.3 if it was found out that $[x']_\uparrow$ is not enabled. In this case, class $[x']_\uparrow$ is referenced by at least one class which contains an unassigned variable. If there is a class $[x'']_\uparrow$ which is reachable from $[x']_\uparrow$ by subset edges then $[x'']_\uparrow$ is (implicitly) referenced by the same unassigned variable. Thus the traversal can stop at class $[x']_\uparrow$.

If assignments are retracted in function `backtrack` in Figure 5.1 then decision candidates in the set DC will be disabled. However, as noted above similarly for enabling variables, not every single variable y which is unassigned will disable decision candidates. Unassigning some variable y from the current assignment A produces the new assignment $A' := A \setminus \{y\}$. Variables from the set $D(y)$ which are decision candidates under A potentially are disabled under A'. Note that if unassigning y disables any variable x then $x \in D(y)$. Consider the class $[y]_\downarrow$ containing all variables which have the same set of dependencies as y. If there is another variable $y' \in [y]_\downarrow$ which is unassigned already under the old assignment A then unassigning y cannot disable additional variables since $D(y) = D(y')$. Like for enabling variables, we assume that retracting assignments and disabling decision candidates is done for each unassigned variable separately. Only if y is the first variable to be unassigned in $[y]_\downarrow$ we have to check if variables in $D(y)$ are disabled. In this case, all classes $[x']_\uparrow$ of variables in the set $D(y)$ are visited by traversing subset edges in graph $G_{\approx,\subseteq}(D)$. Analogously to the optimization described above, classes $[x']_\uparrow$ and all of their successors by subset edges can be skipped if $[x']_\uparrow$ was already disabled before y was unassigned.

Example 5.4.1. Consider graph on the left in Figure 5.4. Assigning x_3 does not enable any new decision candidates since x_4 is still unassigned in the class $[\forall x_3, x_4]_\downarrow$. After x_4 was assigned, the class $[\forall x_3, x_4]_\downarrow$ does not

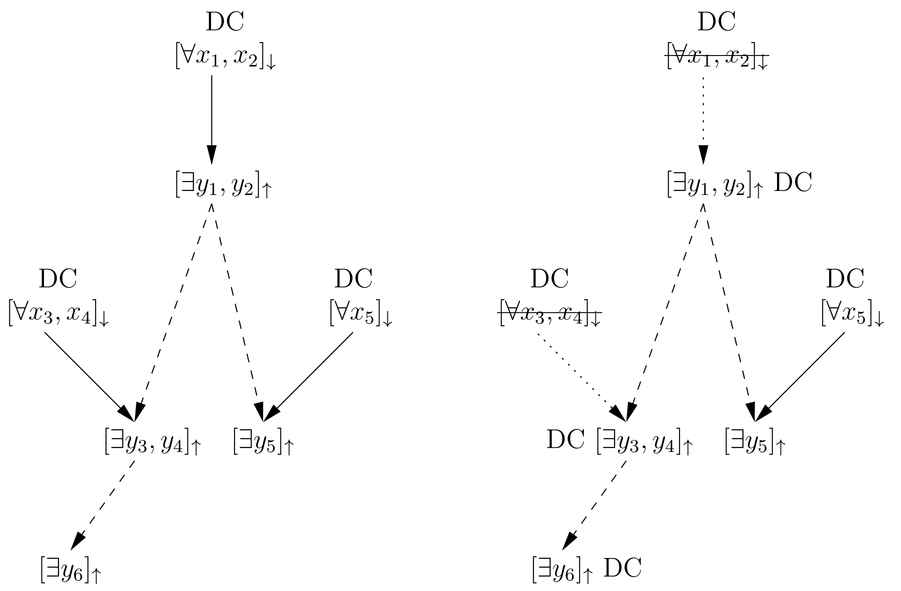

DC
$[\forall x_1, x_2]_\downarrow$
$[\exists y_1, y_2]_\uparrow$
DC
$[\forall x_3, x_4]_\downarrow$
DC
$[\forall x_5]_\downarrow$
$[\exists y_3, y_4]_\uparrow$
$[\exists y_5]_\uparrow$
$[\exists y_6]_\uparrow$
DC
~~$[\forall x_1, x_2]_\downarrow$~~
$[\exists y_1, y_2]_\uparrow$ DC
DC
~~$[\forall x_3, x_4]_\downarrow$~~
DC
$[\forall x_5]_\downarrow$
DC $[\exists y_3, y_4]_\uparrow$
$[\exists y_5]_\uparrow$
$[\exists y_6]_\uparrow$ DC

Figure 5.4: Illustration of a compressed augmented dependency graph $G_{\approx,\subseteq}(D)$ by Definition 4.2.8 for an arbitrary dependency scheme D and PCNF. Examples 5.4.1 and 5.4.2, point out enabling and disabling of variables. For simplicity, not all edges are shown. Solid and dashed edges represent dependency and subset edges, respectively. Dotted edges in the graph on the right indicate that the respective dependency edge is disabled due to assigned variables. Classes labeled with "DC" are decision candidates.

contain assigned variables anymore. Class $[\exists y_3, y_4]_\uparrow$ is not directly referenced by a dependency edge from another class. However, there is an implicit dependency edge by the subset edge from class $[\exists y_1, y_2]_\uparrow$ to $[\exists y_3, y_4]_\uparrow$. Class $[\exists y_1, y_2]_\uparrow$ is referenced by a dependency edge from class $[\forall x_1, x_2]_\downarrow$ which has unassigned variables. This dependency edge refers to $[\exists y_3, y_4]_\uparrow$ as well implicitly by the subset edge. Therefore, none of class $[\exists y_3, y_4]_\uparrow$ and its successors by subset edges is enabled. Due to subset edges, it is sufficient to find out that the parent class $[\exists y_1, y_2]_\uparrow$ of class $[\exists y_3, y_4]_\uparrow$ is not enabled. No other classes in the graph have to be checked. Further, assigning x_1 does not enable new decision candidates since x_2 is still unassigned in the class $[\forall x_1, x_2]_\downarrow$. After x_2 was assigned, the class $[\exists y_1, y_2]_\uparrow$ is enabled since it is not referenced by dependency edges from classes containing unassigned variables. Class $[\exists y_5]_\uparrow$ is not enabled due to the reference from class $[\forall x_5]_\downarrow$. All successors $[\exists y_3, y_4]_\uparrow$ and $[\exists y_6]_\uparrow$ of $[\exists y_1, y_2]_\uparrow$ by subset edges are now enabled, which is shown in the graph on the right.

Example 5.4.2. Consider graph on the right in Figure 5.4. Variables x_1, x_2, x_3 and x_4 are assigned, hence all variables except the class $[\exists y_5]_\uparrow$ are decision candidates. During backtracking, decision candidates are disabled as follows. Unassigning x_3 disables the class $[\exists y_3, y_4]_\uparrow$ which is referenced from class $[\forall x_3, x_4]_\downarrow$ by a dependency edge. All successors of $[\exists y_3, y_4]_\uparrow$ by subset edges, in this case only $[y_6]_\uparrow$, are disabled as well. Unassigning x_4 does not incur any additional work since the class $[\forall x_3, x_4]_\downarrow$ already contains an unassigned variable. It is not necessary to traverse subset edges again.

In our solver DepQBF, decision candidates are maintained as described above based on the *approximated* dependency graph $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$ for the standard dependency scheme D^{std} by Definition 4.5.4 on page 73. In our implementation, the graph $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$ neither contains transitive subset edges nor transitive dependency edges. In order to avoid additional overhead during QBCP, the function for enabling variables is called only right before a decision is made in function `select_dec_var` in Figure 5.1. The effects of all assignments that were made since the most recent decision are then taken into account one after the other to update the set DC. Our experimental results presented in Section 5.7 show that this way decision candidates can be maintained efficiently.

The set DC of decision candidates with respect to the current assignment and the dependency scheme applied in QDPLL contains all variables which can be assigned as decisions. In SAT solving, where at any point of time in QDPLL all variables are decision candidates, several branching heuristics have been suggested to select a variable from the set DC as next decision. We referred to related work in Section 2.3.1 on page 16. Branching heuristics from the domain of SAT solving basically can be extended to QBF by restricting the selection to set DC. In our QBF solver DepQBF [84], we implemented variable state independent decaying sum heuristic (VSIDS) [93] similarly to the MiniSAT solver [42].

5.5 Dependency Checking

Constraint reduction by Definition 5.3.2 shortens constraints with respect to dependencies between variables occurring in the constraint. Due to applications in QBCP, a slow implementation of constraint reduction will likely have a negative effect on the overall run time of QBCP. In fact, the vast majority of assignments in QDPLL are due to implications in QBCP.[1] We conclude that constraint reduction is performed frequently during QBCP and we aim at efficient application in practice.

[1]For example, on 372 formulae solved by DepQBF out of total 568 formulae from QBFEVAL'10[100], on average 19.08 implications were detected per decision during QBCP in function qbcp in Figure 5.1. On the benchmark set from in QBFEVAL'08 (3326 formulae), 88% of total assignments in DepQBF were implications, with 59% unit literals and 29% pure literals [84].

With respect to the trivial dependency scheme D^{triv}, variable y depends on variable x if $x < y$ by prefix ordering. This way, dependencies can easily be checked by comparing the levels of quantifier blocks. However, this is no longer possible if more refined dependency schemes $D \subseteq D^{\text{triv}}$ are applied.

In order to check for dependencies with respect to an arbitrary dependency scheme D, we consider a similar approach as for decision making described in the previous section.

Let D be a dependency scheme and $G_{\approx,\subseteq}(D)$ be the compressed augmented dependency graph for D by Definition 4.2.8. We assume that the graph $G_{\approx,\subseteq}(D)$ does not contain dependency edges which are implicitly represented by subset edges. This also applies to our implementation of the graph-based representation of the standard dependency scheme D^{std} in DepQBF (see Section 4.5.3 for an illustration). Given two variables x and y, we can check if $x \prec_D y$ as follows.

1. Find the classes $[x]_\downarrow$ and $[y]_\uparrow$ of x and y, respectively.
2. Consider all ancestors $[y']_\uparrow$ of $[y]_\uparrow$ which are reachable over zero or more subset edges in $G_{\approx,\subseteq}(D)$:
    2.1. If ancestor $[y']_\uparrow$ is referenced by a dependency edge from class $[x]_\downarrow$ then $x \prec_D y$.
3. At this point, all ancestors of $[y]_\uparrow$ have been checked and hence $x \not\prec_D y$.

5.6 Constraint Learning

In the previous sections, we considered the generation of assignments in QDPLL as shown in Figure 5.1 by means of decision making and QBCP. We generalized these parts to arbitrary dependency schemes. As the final step of combining QDPLL with arbitrary dependency schemes, we address constraint learning in this section. Similarly to QBCP, it turns out that the classical approach of constraint learning can be adapted by applying the generalized variant of constraint reduction from Definition 5.3.1.

As noted in Section 5.2.3 above, constraint learning in QDPLL relies on Q-resolution, the QBF-specific variant of resolution [27, 108]. Given a PCNF ψ, Q-resolution is a sound and complete approach to solve ψ [27]. PCNF ψ is unsatisfiable if and only if the empty clause can be derived from ψ by Q-resolution. Otherwise, if the set of all possible Q-resolvents that can be derived from ψ does not contain the empty clause then ψ is satisfiable. In contrast to that, the application of Q-resolution for constraint learning in QDPLL does *not* focus on the generation of all possible Q-resolvents. Instead, Q-resolvents are derived selectively with respect to the current conflicting assignment that was enumerated by QDPLL. Thus we may think of constraint learning in QDPLL as a heuristic application of Q-resolution guided by the generation of assignments. A similar view was presented for propositional resolution in SAT solvers [70].

In the following, we present constraint learning for QDPLL as implemented in our solver `DepQBF`. Function `analyze_leaf` in Figure 5.5 shows a high-level workflow. We focus on practical aspects such as the selection of pivot variables to generate asserting constraints. We begin with a formal definition of Q-resolution.

Definition 5.6.1 ([26, 27, 61, 134])**.** Let C_1 and C_2 either be two clauses or two cubes such that, for some variable v, $v \in C_1, \neg v \in C_2$ and $q(v) = \exists$ if C_1 and C_2 are clauses and $q(v) = \forall$ otherwise. The *tentative Q-resolvent* of C_1 and C_2 on *pivot variable* v is the constraint

$$C_1 \otimes C_2 := (CR(C_1) \cup CR(C_2)) \setminus \{v, \neg v\}.$$

If the tentative Q-resolvent $C_1 \otimes C_2$ contains complementary literals, that is $\{x, \neg x\} \subseteq C_1 \otimes C_2$ for some variable x, then no *Q-resolvent* exists. Otherwise, the *Q-resolvent*

$$C := CR(C_1 \otimes C_2)$$

is obtained by an additional application of constraint reduction to $C_1 \otimes C_2$.

Although Q-resolution is originally defined for clauses, we extend the definition to cubes for simplicity. Q-resolution on cubes is also called *consensus* [134]. Note that by Definition 5.6.1 the pivot variable must be existential if clauses are resolved and universal otherwise.

As far as clauses are concerned, the application of constraint reduction is the *only* difference between Q-resolution by Definition 5.6.1 and propositional resolution as applied in SAT solvers. In fact, Q-resolution is incomplete if constraint reduction is not applied [26] as pointed out in the following example.

Example 5.6.1 (see also Section 23.5 in [26] for a related example)**.** Given the unsatisfiable PCNF $\psi := \exists x \forall y.\ (x \vee y) \wedge (\neg x \vee \neg y)$. If constraint reduction is omitted in Q-resolution then the empty clause cannot be derived from ψ since the tentative resolvent $(x \vee y) \otimes (\neg x \vee \neg y) = (y \vee \neg y)$ on x is tautological.

Due to constraint reduction, as pointed out in Example 5.3.1 above, in general it is crucial for the correctness of Q-resolution to prevent complementary literals in Q-resolvents.

Example 5.6.2. Given the satisfiable PCNF $\psi := \forall x \exists y.\ (x \vee \neg y) \wedge (\neg x \vee y)$. If the requirement that tentative Q-resolvents must not contain complementary literals is dropped from Definition 5.6.1, then Q-resolution can derive the empty clause from ψ: $(x \vee \neg y) \otimes (\neg x \vee y) = (x \vee \neg x)$ and finally $CR((x \vee \neg x)) = \emptyset$.

```
DecLevel analyze_leaf (State s)
  R = get_initial_constraint (s);
  // s == UNSAT: 'R' is empty clause.
  // s == SAT: 'R' is sat. cube...
  // ..or new cube from assignment.
  while (!stop_res (R))
    p = get_pivot (R);
    R' = get_antecedent (p);
    R = constraint_res (R, p, R');
  add_to_formula (R);
  return get_asserting_level (R);
```

Figure 5.5: Function `analyze_leaf` from Figure 5.1. Assuming that the current assignment A generated by decision making and QBCP is conflicting (`s == UNSAT`), initially R is a clause which is falsified (`get_initial_constraint`). Q-resolution generates a new learnt clause by successively resolving the antecedent clauses R' (`get_pivot`) of unit literals in the current clause R with R. The process stops if clause R is asserting (`stop_res`) and QDPLL backtracks to the asserting level where the new learnt clause R triggers an implication (`get_asserting_level`).

We assume that formulae do not contain clauses or cubes with complementary literals. This way, incorrectness of constraint reduction and Q-resolution as pointed out in Examples 5.3.1 and 5.6.2 is avoided.

Note that dependency schemes affect Q-resolution only with respect to constraint reduction. Due to Proposition 5.3.1, the use of more refined dependency schemes potentially produces shorter resolvents. This property motivates combinations of constraint learning for QDPLL based on more refined dependency schemes. Altogether, this adds to the benefits that can be drawn from dependency schemes in QBCP and decision making as argued in Sections 5.3.4 and 5.4 above.

Recall that we apply *one and the same* dependency scheme D in all parts of QDPLL, that is constraint reduction in QBCP, decision making and constraint learning to be discussed below.

5.6.1 Generation of Learnt Constraints

In the following, we describe the generation of learnt constraints in QDPLL. Our description largely relies on insights presented in [61, 134], but we also address the implementation of our solver DepQBF [84]. Thereby, we include practical aspects of constraint learning. For simplicity, we confine the presentation of constraint learning to the generation of learnt clauses. Apart from initialization, the generation of learnt cubes is entirely dual to clause learning. We refer to Section A.1.1 in the appendix for related definitions.

Initialization - Clause Learning

Given the current ACNF ψ, where learnt constraints might have been added previously, and the current admissible assignment $A := \{l_1, l_2, \ldots, l_n\}$ generated by QBCP which is sorted by trail levels $1, 2, \ldots, n$. Assume that A is conflicting by Definition 5.3.7, that is $QBCP(\psi[A]) = \psi'[A] = \bot$. Figure 5.5 shows function `analyze_leaf` from Figure 5.1 again. Note that by assumption `s == UNSAT` in the figures. Since A is conflicting, there must be at least one clause $R \in \psi$ such that R either is empty (thus falsified) under assignment A, that is $R[A] = \bot$, or R became empty by constraint reduction in $QBCP(\psi[A])$, that is $CR(R[A]) = \emptyset$. If there are multiple such clauses in ψ then let R be an arbitrary one. Clause R could be an original or a learnt clause. Starting from R, clause learning produces a new learnt clause C such that, after backtracking to a specific decision level, QBCP can infer an additional implication from C by the unit literal rule.

In practice, function `get_initial_constraint` in Figure 5.5 selects R according to the requirements stated above. Clause R is used as initial clause for the process of generating a new learnt clause by Q-resolution according to Definition 5.6.1. Antecedent clauses R' of unit literals assigned in R by Definition 5.3.4 are successively resolved with R to obtain a new clause. In order to carry out constraint reduction efficiently during Q-resolution, a similar approach as described in Section 5.5 can be applied. If the current resolvent R is *asserting*, that is a new implication can be inferred from R after backtracking, then the process stops. Clause R is added to the set of learnt clauses and QDPLL backtracks to the *asserting level* of R. We consider asserting clauses and asserting levels in Section 5.6.1 below.

Initialization - Cube Learning

Assume that the current assignment A is satisfying by Definition 5.3.7, that is $QBCP(\psi[A]) = \psi'[A] = \top$ (`s == SAT` in Figure 5.5). Either there is a cube $R \in \psi$ which became empty (thus satisfied) by QBCP or all clauses in ψ are satisfied under A. The former case is dual to an empty clause under a conflicting assignment A and function `get_initial_constraint` select one such cube R to generate a new learnt cube according to Figure 5.5.

If all clauses in ψ are satisfied under A then a cover set $A' \subseteq A$ is generated from A by Definition 5.2.8. Function `get_initial_constraint` constructs a new cube R' containing all the literals in A' and returns the reduced cube $R := CR(R')$. Although cube R is not necessarily a cover set, Proposition 5.2.1 still holds due to the correctness of constraint reduction by Theorem 5.3.2. The cube to be learnt is generated from R dually to clause learning, which is described below.

Pivot Selection

In modern SAT solvers like PicoSAT or MiniSAT [18, 42], for example, function `get_pivot` in Figure 5.5 typically selects pivot variables from variables in the current clause R in reverse trail ordering. Hence the variable which was assigned as unit literal most recently in clause R is selected. According to that policy, a learnt clause related to the *first unique implication point (1UIP)* in the *implication graph*, which is associated with the current assignment A, is obtained. Informally, an implication graph represents implications in A as a directed acyclic graph [36, 117, 118, 132]. To the best of our knowledge, there is no comprehensive theoretical framework of implication graphs and 1UIPs in the context of QBF. We concentrate on practical aspects in our description of learning for QDPLL.

The strategy to select pivot variables in reverse trail ordering like in clause learning for SAT cannot directly be applied to QBF. If that strategy was applied then, due to combinations of constraint reduction and the unit literal rule, universal literals which were reduced from antecedent clauses by QBCP might produce tautological resolvents during Q-resolution (function `constraint_res` in Figure 5.5). The dual problem can occur in cube learning if complementary existential literals are introduced into the resolvent.

One approach to tackle the problem of resolvents containing complementary literals is to integrate them into the theoretical and practical framework of QDPLL with constraint learning. The resulting variant of Q-resolution is called *long-distance resolution* [133]. The rules for unit literal detection and constraint reduction have to be adapted accordingly.

As an alternative, resolvents containing complementary literals can be prevented by selecting other pivot variables whenever Q-resolution on the previously selected pivot would introduce complementary literals into the resolvent [61]. We implemented this approach in our solver DepQBF. Our presentation below relies on the algorithm sketched in Figure 6 in [61]. Additionally, we address pitfalls related to bad selections of pivot variables.

Let A be the current assignment and R be the current resolvent in the loop in Figure 5.5. At any time during the generation of a learnt clause, all existential literals in R are falsified by A. Additionally, there might be unassigned universal literals in R which are reducible by constraint reduction. Thus clause R is falsified under A, that is $QBCP(R[A]) = \perp$. The pivot variable is selected with respect to A and R as follows.

1. Let R be the current resolvent (line `p = get_pivot (R)` in Figure 5.5).

2. Let $P(R) := \{y \mid y = v(l), l \in R, q(l) = \exists, am(l) = U\}$ be the set of existential variables in R which were assigned by the unit literal rule. Set $P(R)$ contains all *potential* pivot variables to be selected.

3. Select $p \in P(R)$ such that p has the maximum trail level of variables in $P(R)$, that is $tl(p) = max(\{tl(y) \mid y \in P(R)\})$.

4. If the tentative resolvent $R \otimes R'$ of R and the antecedent $R' = \mathit{ante}(p)$ of p does *not* contain complementary literals, then variable p is the pivot for the next application of Q-resolution (line `R = constraint_res (R, p, R')` in Figure 5.5).

5. Otherwise, if $\{x, \neg x\} \subseteq (R \otimes R')$ for some variable $x \in R$ then an alternative pivot is selected as follows.

   5.1. Let x be one of the variables which occur both positively and negatively in the tentative resolvent $(R \otimes R')$.

   5.2. Let $P'(R) := P(R) \setminus \{y \mid y \in P(R), x \not\prec y\} \setminus \{y \mid y \in P(R), \{x', \neg x'\} \subseteq R \otimes R'' \text{ where } R'' = \mathit{ante}(y)\}$. In addition to the restrictions on set $P(R)$ as defined above, the set $P'(R)$ contains only existential variables which depend on variable x. Further, the resolvent of R and the antecedent of variables in $P'(R)$ does not contain complementary literals.

   5.3. Select $p \in P'(R)$ such that p has the maximum trail level of variables in $P'(R)$, that is $\mathit{tl}(p) = \mathit{max}(\{\mathit{tl}(y) \mid y \in P'(R)\})$.

   5.4. Variable p is the pivot for the next application of Q-resolution (line `R = constraint_res (R, p, R')`).

Note that complementary literals in resolvents are always due to universal variables which were reduced by constraint reduction in the current resolvent and the antecedent. Therefore, this problem cannot occur during clause learning in SAT solvers since all variables are existential. Dually in cube learning, complementary existential literals might be introduced.

The idea behind the two-phase selection process described in the algorithm above is to always pick the existential variable p which was assigned as unit literal most recently. Only if resolution on p would introduce complementary literals of some variable x then an alternative pivot is selected. In this case, the goal is to resolve out existential variables which prevent the literal of x in the current resolvent from being removed by constraint reduction. Therefore, it is necessary to focus on variables which depend on x with respect to the dependency scheme that is used in QDPLL. During the generation of a learnt constraint, complementary literals which were removed by constraint reduction might be introduced again into the current resolvent. In the worst case, an exponential number of intermediate resolvents has to be produced until the final learnt constraint is obtained [50].

For cube learning, pivot selection is dual to clause learning. The goal is to select universal variables assigned by the unit literal rule, where the introduction of complementary literals into the resolvent has to be avoided analogously. We refer to the appendix in for related algorithms.

The algorithm presented above is in essence equivalent to Figure 6 in [61], where it was proved that an asserting clause can always be generated according to that algorithm (see Lemma 4 in [61]). Note that in our presentation, we generalize dependency checking by quantifier prefixes as in [61] to arbitrary dependency schemes. We conjecture that the proof can be adapted in a similar way.

Stop Criteria and Asserting Levels

After a suitable pivot variable p has been selected, the current resolvent R is resolved with the antecedent R' of p (line `R = constraint_res (R, p, R')` in Figure 5.5). The resolution process stops if the current resolvent is asserting according to the current assignment A (function `stop_res`). See also Example 5.6.3 below.

Definition 5.6.2 ([53, 133])**.** Given the current resolvent R. Let $m := max(\{d \mid d = dl(y), y = v(l), l \in R, q(l) = \exists\})$ be the largest decision level of existential variables in R. Clause R is *asserting* if and only if:

1. There is exactly one literal $l \in R$ with $q(l) = \exists$ such that $dl(v(l)) = m$. Let $v_a := v(l)$.
2. The decision variable at decision level m is existential.
3. Variables where v_a depends on must be assigned where the decision levels must be smaller than $m = dl(v_a)$, that is:
$$\forall y \in \{y \mid y = v(l), l \in R, y \prec v_a\} : dl(y) < m.$$

If R is asserting then variable v_a is *asserted* by R.

Different from the definition of asserting clauses in [133], Condition 2 is not included in [53]. Asserting clauses are generated in DepQBF with respect to Definition 5.6.2. For cube learning, the conditions of asserting cubes are dual. We refer to Definition A.1.1 in the appendix.

As pointed out in Section 5.2.3, QDPLL by Figure 5.1 does not explicitly flip the value of decision variables. Instead, every learnt constraint R is constructed in such way that QBCP can infer an additional unit literal from R after backtracking. Given the current conflicting assignment A and the asserting clause R that was learnt according to Definition 5.6.2, QDPLL backtracks to the *asserting level* with respect to R defined as follows.

Definition 5.6.3. Given an asserting clause R by Definition 5.6.2 and the variable v_a asserted by R.

1. Let $DL_{\exists}(R) := \{d \mid d = dl(y), y = v(l), l \in R, q(l) = \exists, y \neq v_a\}$ be the set of decision levels of existential variables in R, excluding v_a.
2. Let $DL_{\prec}(R) := \{d \mid d = dl(y), y = v(l), l \in R, y \prec v_a\}$ be the set of decision levels of variables in R where v_a depends on.

3. The *asserting level* d_a with respect to R is $d_a := max(DL_{\exists}(R) \cup DL_{\prec}(R))$.

Given the current conflicting assignment A, asserting clause R and the asserting level d_a by Definition 5.6.3, QDPLL retracts all assignments which have decision levels *larger* than d_a during backtracking, yielding a new assignment $A' \subseteq A$. The assignment l, where $l \in R$ is the literal of variable v_a in R, can now be inferred from R by unit literal detection because R is unit under A' with constraint reduction, that is $CR(R[A']) = (l)$. Due to Definition 5.6.3, universal literals of variables where v_a depends on are falsified by A' in R. All other universal literals are removed by constraint reduction from the clause $R[A']$. For cube learning, the definition of asserting levels is dual. We refer to Definition A.1.2 in the appendix. Definitions 5.6.2 and 5.6.3 are illustrated by the following example.

Example 5.6.3. Given the quantifier prefix $\exists x \forall a \exists y \forall b \exists z$ and the clause $C := (x(1) \vee a(4) \vee y(3) \vee b(3) \vee z(3))$, where numbers in parenthesis indicate decision levels. We are using the prefix ordering for dependency analysis.

Clause C is not asserting since variables y and z have decision level 3, which is the largest decision levels of existential variables in C.

Clause $C := (x(1) \vee a(4) \vee y(3) \vee b(3) \vee z(2))$ is not asserting since variable a, where y depends on, is assigned at a decision level larger than y.

Clause $C := (x(1) \vee a(2) \vee y(3) \vee b(3) \vee z(2))$ is asserting since the asserted variable y is the only variable at the largest decision level of all existential variables. Further, variable a, where y depends on, is assigned at a decision level smaller than y.

Given the asserting clause $C := (x(10) \vee a(15) \vee y(30) \vee b(25) \vee z(20))$, where variable y is asserted by C. Decision level $d_a := 20$ is the asserting level with respect to R. Variable b is ignored for the computation of d_a since y does not depend on b.

Given the asserting clause $C := (x(10) \vee a(20) \vee y(30) \vee b(25) \vee z(10))$, where variable y is asserted by C. Decision level $d_a := 20$ is the asserting level with respect to R. Variable a is taken into account for the computation of d_a since y depends on b.

Note that clause (cube) learning does *not* treat literals of universal (existential) variables in the current resolvent R in any special way. For example, assume that there is a literal l of an unassigned universal variable in clause R which prevents R from being asserting. The criterion of asserting clauses by Definition 5.6.2 (Condition 3) makes sure that the variable $v(l)$ of l is eventually reduced by constraint reduction during forthcoming Q-resolution steps.

Constraint Learning and Dependency Schemes

It is important to note that the general workflow of constraint learning as illustrated by Figure 5.5 is independent of the dependency scheme that is used in QDPLL. This applies to the high-level view of QDPLL with constraint learning shown in Figure 5.1 as well. Actually, the only difference between our description and classical ones *implicitly* relying on the trivial dependency scheme D^{triv} [61, 134] is the generalization to arbitrary dependency schemes. This affects dependency checking for pivot selection, stop criteria and the computation of asserting levels. Actually, we may think of a dependency scheme as an additional parameter that is passed to QDPLL. The generalization of constraint learning to arbitrary dependency schemes presented in this section completes the combination of QDPLL with dependency schemes.

5.6.2 Optimizations

The basic framework of QDPLL with constraint learning as shown in Figure 5.1 does not differ from DPLL for SAT, apart from cube learning. Several optimizations have been suggested for DPLL which address the selection of decision variables or the management of learnt clauses, for example. We considered related work in Section 2.3.1. We refer to [84] for further details on the implementation of the following optimizations.

In DepQBF, learnt constraints in ϕ_{LCL} and ϕ_{LCU} are periodically *deleted.* Deletion is necessary to avoid overhead in QBCP. The idea is to keep those constraints which are considered to be important with respect to a certain heuristic criterion. Learnt clauses and cubes are kept in doubly linked lists. The idea is to let frequently used learnt constraints appear at the head of the lists by a simple move-to-front strategy. When deleting constraints, the lists are processed starting from the tail, thus removing least-recently used and possibly less important constraints.

In SAT solving, *restarting* the search has been found useful to improve performance [17, 65, 70, 109, 119]. Instead of backtracking to the asserting level as described in Section 5.6.1, the solver periodically retracts *all* assignments made so far and thus restarts the search process, while keeping learnt constraints. DepQBF implements a restart schedule inspired by PicoSAT [18] which is based on ideas from [15]. Different from typical restarts in SAT solvers, DepQBF restarts the search from the decision level of the most recently assigned universal decision variable. Like many SAT solvers, DepQBF combines restarts with assignment caching [103]. Decision variables are assigned a cached value, if possible. The cache of the values of variables is updated whenever assignments are made. In Section 5.7.2 below, we report the results of related experiments.

5.7 Experimental Results

In this chapter, we combined the classical QDPLL algorithm [30] with arbitrary dependency schemes. Actually, we extended QDPLL from the trivial dependency scheme, which is implicitly applied in its classical variant, to arbitrary ones. This way, QDPLL can profit from additional information on independence of variables in a given PCNF. Apart from decision making, constraint reduction is the central point where independence might have a positive influence. Since constraint reduction removes literals from constraints, we might detect more unit literals during QBCP and generate shorter learnt constraints, for example.

Given the theoretical framework of QDPLL with dependency schemes, we now focus on practical aspects. In addition to comparing the effects of different dependency schemes in QDPLL in the following section, we provide a general performance analysis of our solver DepQBF in Section 5.7.2. Thereby, we also briefly address the role of preprocessing.

5.7.1 QDPLL with Different Dependency Schemes

In this section, we evaluate the performance of QDPLL when combined with the three dependency schemes D^{triv}, D^{tree} and D^{std} as introduced in Section 3.4.3. We want to find out whether the theoretical advantages of more refined dependency schemes due to $D^{\text{std}} \subseteq D^{\text{tree}} \subseteq D^{\text{triv}}$ also show up in practice. The number of solved formulae or average run times indicate overall performance. Additionally, the number of backtracks, implications and decision candidates allow for a more fine-grain analysis of dynamic effects.

For experiments, we used a variant of our solver DepQBF.[2] The standard dependency scheme D^{std} is represented as the approximated dependency graph $\dot{G}_{\approx,\subseteq}(D^{\text{std}})$ by Definition 4.5.4 on page 73. For D^{triv} and D^{tree}, we implemented augmented compressed dependency graphs $G_{\approx,\subseteq}(D^{\text{triv}})$ and $G_{\approx,\subseteq}(D^{\text{tree}})$ by Definition 4.2.8 on page 60, where transitive dependency edges and subset edges were omitted. Actually, the graph $G_{\approx,\subseteq}(D^{\text{triv}})$ corresponds to a linear list of equivalence classes of variables.

Apart from the different dependency graphs for representing the dependency schemes, the implementation of QDPLL by Figure 5.1 in DepQBF is the same regardless of whether D^{triv}, D^{tree} or D^{std} is applied. This tight integration of QDPLL and dependency schemes allows for a direct comparison of the effects observed in practice. In order to construct one out of possibly many non-deterministic dependency graphs (trees) for D^{tree}, we adapted the approach from [12] to our framework.

Table 5.1 shows a comparison[3] of DepQBF with D^{triv}, D^{tree} and D^{std} on instances from previous QBF competitions [56]. Benchmarks include all structured formulae from *QBFEVAL'07*, *QBFEVAL'08* and from set *Herb-*

[2]Experimental data on the comparison of dependency scheme combined with QDPLL in Tables 5.1 to 5.3 was published in [85]. The publicly available DepQBF version 0.1 described in [84] is different and supports only D^{triv} and D^{std}.

[3]Setup for all experiments reported in Section 5.7.1, unless stated otherwise: Ubuntu Linux 9.04, Intel® Q9550@2.83 GHz, 3 GB/900 seconds memory and time limit.

QBFEVAL'08 (3326 formulae)					
	D^{triv}	D^{tree}	D^{std}	QuBE6.6-nopp	QuBE6.6
Solved	1223	1221	**1252**	1106	**2277**
Avg. Time	579.94	580.64	**572.31**	608.97	**302.49**
QBFEVAL'07 (1136 formulae)					
Solved	533	548	**567**	458	**734**
Avg. Time	497.12	484.69	**469.97**	549.29	**348.05**
Herbstritt (478 formulae)					
Solved	321	357	357	296	**395**
Avg. Time	316.06	248.20	**248.07**	357.52	**173.53**

Table 5.1: Performance comparison of DepQBF with quantifier prefixes (D^{triv}), quantifier trees (D^{tree}) and the standard dependency scheme (D^{std}). Statistics of the QDPLL-based solver QuBE6.6 [57] with and without preprocessing (QuBE6.6-nopp) are listed for reference.

stritt [56]. Average run times are given in seconds. The three versions of DepQBF do *not* apply preprocessing and differ only in the integrated dependency schemes, all other parts are *exactly* the same. For external reference, statistics of PCNF-based QuBE6.6 [57] with and without preprocessing (QuBE6.6-nopp) are listed.[4] Preprocessing in QuBE6.6 is based on approaches described in [55]. We evaluate the performance of DepQBF with and without preprocessing in Section 5.7.2 below.

We implemented dependency checking with respect to D^{tree} and D^{std} as described in Section 5.5. For D^{triv}, an optimal approach was applied. By comparing the levels of variables based on the quantifier prefix ordering, dependencies can be checked in constant time. An approach based on numberings of quantifier trees by depth-first search was applied in [62]. This way, dependency checking can be carried out in constant time for D^{tree} as well. However, we did not implement that approach but inspect the dependency graph of $G_{\approx,\subseteq}(D^{\text{tree}})$ explicitly. Although we expect additional overhead from dependency checking for D^{std} and D^{tree}, QDPLL with D^{std} is best on *QBFEVAL'07* and *QBFEVAL'08* and is slightly faster than D^{tree} on set *Herbstritt*. There is a large performance gap to QuBE6.6 which, differently from DepQBF, uses preprocessing. However, any version of DepQBF outperforms QuBE6.6-nopp, where preprocessing is disabled. Note that in our terminology QuBE6.6 uses D^{triv}.

A more detailed comparison of all three versions of DepQBF is shown in Table 5.2. Only formulae solved by *both* solvers ($\cap$) were considered. For

[4]Due to limited computational resources, we did not rerun the experiments for Table 5.1 with QuBE7.2, the current successor of QuBE6.6 by February 2012.

QBFEVAL'08 (solved only)						
	$D^{\text{triv}} \cap D^{\text{tree}}$		$D^{\text{triv}} \cap D^{\text{std}}$		$D^{\text{tree}} \cap D^{\text{std}}$	
solved	1172		1196		1206	
time	**23.15**	26.68	**23.73**	25.93	25.63	**22.37**
implied/assigned	90.4%	**90.7%**	88.6%	**90.5%**	90.9%	**92.1%**
backtracks	32431	**27938**	34323	**31085**	**25106**	26136
sat. cubes/sol.	1.8%	**2.9%**	1.8%	**2.6%**	**3.6%**	3.1%
learnt constr. size	157	**99**	150	**96**	102	**95**
QBFEVAL'07 (solved only)						
solved	501		513		537	
time	**31.22**	34.46	32.76	**32.66**	33.31	**28.33**
implied/assigned	89.0%	**89.2%**	87.7%	**89.5%**	89.9%	**91.9%**
backtracks	35131	**22334**	39906	**26362**	**21945**	22323
sat. cubes/sol.	4.0%	**10.0%**	4.0%	**9.5%**	**10.8%**	9.9%
learnt constr. size	150	**101**	134	**113**	100	**96**
Herbstritt (solved only)						
solved	312		308		348	
time	26.86	**19.28**	24.41	**19.28**	**20.46**	20.83
implied/assigned	96.6%	**97.4%**	96.2%	**97.4%**	97.4%	97.4%
backtracks	26565	**1329**	26733	**1482**	**1615**	1800
sat. cubes/sol.	0%	0%	0%	0%	0%	0%
learnt constr. size	**174**	306	**173**	323	**407**	410

Table 5.2: Comparing combinations of DepQBF with quantifier prefixes (D^{triv}), quantifier trees (D^{tree}) and the standard dependency scheme (D^{std}) on the intersection of solved formulae.

example, in section "$D^{\text{triv}} \cap D^{\text{std}}$", the left column reports statistics for D^{triv}, the right one for D^{std}. Average values are given for run time in seconds, ratio of implications among all assignments, number of backtracks, ratio of satisfied learnt cubes among all identified solutions and size (number of literals) of learnt constraints.

In general, D^{triv} is slightly faster on the *QBFEVAL* sets. However, D^{triv} yields more backtracks than D^{tree} and D^{std} on all sets. On set *Herbstritt*, the difference in this respect is a factor of up to 20. D^{tree} and D^{std}, both being more refined than D^{triv}, produce smaller learnt constraints on the *QBFEVAL* sets. Furthermore, D^{std} triggers more implications in QBCP on all sets and D^{triv} fewer satisfied learnt cubes. These effects can be attributed to unit literal detection and constraint reduction, which benefit from more refined dependency schemes.

The results from Table 5.2 indicate that moving from D^{triv} to more refined dependency schemes incurs run time overhead (except on set *Herbstritt*), but also allows QDPLL to produce shorter runs in terms of backtracks. Dependency checking is not a constant-time operation in general dependency DAGs used to represent D^{tree} and D^{triv}, for example. Instead, such DAGs are inspected explicitly as described in Section 5.5. Additionally, maintaining the set of decision candidates as described in Section 5.4 is more expensive than with respect to D^{triv}. As indicated in Table 5.1, QDPLL still seems to profit from using more refined dependency schemes such as D^{tree} and D^{std}.

In order to assess both the costs and benefits of combining dependency schemes with QDPLL in more detail, we carried out the following experiment. In addition to the dependency DAG which is used to represent the dependency scheme in QDPLL, called primary DAG G_1, another dependency scheme is represented by a secondary dependency DAG G_2. The secondary DAG G_2 is *independent* from G_1 and used *in parallel* for statistical computations only. The idea is to compare the effects of using different dependency schemes *dynamically*, that is during a run of QDPLL.

This setup allows to compute more fine-grain statistics than overall run time or number of backtracks, as listed in Tables 5.1 and 5.2. During a run of QDPLL, it is interesting to compare the numbers of decision candidates (DC) by Definition 5.4.1 with respect to G_1 and G_2 under the current assignment. These numbers are computed each time before a decision is made and reflect the degree of freedom resulting from more refined dependency schemes. For example, we expect D^{std} to allow more decision candidates than D^{triv} and D^{tree}. Apart from that, we want to measure the average costs of dependency checking and decision candidate maintenance for dependency DAGs resulting from different dependency schemes.

For G_1 and G_2, we compared D^{std} to D^{triv} and D^{tree} , where all four combinations were run to even out biased solver behaviour. Due to limited computational resources, we did not compare D^{triv} to D^{tree} and omitted *QBFEVAL'07*.

Table 5.3 shows the results of the experiments described above with a time out of 900 seconds. DepQBF maintains *two* dependency DAGs G_1 (primary) and G_2 (secondary) in parallel. For example, in section "$D^{\text{triv}} \ltimes D^{\text{std}}$", G_1 is obtained from D^{triv} (left column), G_2 from D^{std} (right column). Note that columns "D^{std}" in "$D^{\text{std}} \ltimes D^{\text{triv}}$" and "$D^{\text{std}} \ltimes D^{\text{tree}}$" are incomparable since G_2 influences run time, that is "$D^{\text{std}} \ltimes D^{\text{triv}}$" and "$D^{\text{std}} \ltimes D^{\text{tree}}$" may run at different speeds. We compare the numbers of decision candidates (set DC as introduced in Section 5.4) when using different dependency schemes. Each time before decision making, the size of the set DC under the current assignment is computed. Row "DC/d" shows the total sum of decision candidates over the total number of decisions in the benchmark set after at most 900 seconds run time.

As indicated for sets *QBFEVAL'08* and *Herbstritt*, the difference in DC

QBFEVAL'08 (3326 formulae)				
	$D^{\text{triv}} \ltimes D^{\text{std}}$		$D^{\text{std}} \ltimes D^{\text{triv}}$	
DC/d	13801.0	13801.6	11409.7	11409.0
DC-updt.	3.23	3.16	3.30	3.43
$\prec$	1	-	6.21	-
C-red.	1.18	-	535.62	-
Herbstritt (478 formulae)				
DC/d	21.3	26.55	20.14	20.13
Pan (384 formulae) $\cup$ *Sorting-Networks (84 formulae)*				
DC/d	75.81	93.87	117.50	109.66

QBFEVAL'08 (3326 formulae)				
	$D^{\text{tree}} \ltimes D^{\text{std}}$		$D^{\text{std}} \ltimes D^{\text{tree}}$	
DC/d	8932.5	8933.0	15625.6	15625.3
DC-updt.	3.38	3.37	3.30	3.36
$\prec$	7.15	-	6.26	-
C-red.	538.30	-	540.94	-
Herbstritt (478 formulae)				
DC/d	20.67	20.67	20.16	20.16
Pan (384 formulae) $\cup$ *Sorting-Networks (84 formulae)*				
DC/d	86.89	86.90	120.03	119.98

Table 5.3: Comparing the costs and benefits of D^{triv}, D^{tree} and D^{std} in DepQBF on the intersection of solved formulae.

statistics is very small in general, sometimes less than one candidate on average per decision. However, it seems that this is already enough for QDPLL with D^{std} to outperform D^{triv} and D^{tree} by Table 5.1. Further, *DC* statistics are also family-dependent, as shown by the results for sets *Pan* and *Sorting-Networks* in Table 5.3.

Additionally, we measured the costs of integrating dependency DAGs in QDPLL. Cost statistics in Table 5.3 (rows "*DC-updt.*", "$\prec$", "*C-red.*") are correlated to the number of equivalence classes of variables that have to be visited (pointer dereferences) when inspecting a dependency DAG. Such inspections are necessary for decision candidate management and dependency checking as described in Sections 5.4 and 5.5. Average costs are listed for (un)assigning an assignment as defined in Section 5.4 to update the set *DC* of decision candidates (*DC-updt.*), for constraint reduction as needed in QBCP and for the stop criterion during constraint learning ($\prec$), and separately for constraint reduction per Q-resolution operation (*C-red.*).

The latter are irrelevant for G_2 (“-”).

Average costs for dependency checking and (un)assigning variables for updating DC before decisions or during backtracking are small. This is due to equivalence classes in dependency DAGs. However, costs of constraint reduction during constraint learning are very high for D^{tree} and D^{std}. These effects are closely related to implementation. When using D^{triv}, all constraints C can be kept sorted according to prefix ordering, which allows for efficient constraint reduction. This was implemented in DepQBF with D^{triv} and is reflected by low costs in Table 5.3. In general, such an approach is impossible for arbitrary dependency schemes. Instead, we reduce constraints based on classes in the dependency DAG for D^{tree} and D^{std}. Classes are collected for all literals in a constraint C before reduction, where the size of C (particularly for cubes) can be large. This effort is included in the statistics shown in Table 5.3. Despite that overhead in D^{tree} and D^{std}, overall performance by Tab. 5.1 is still better than with D^{triv}.

5.7.2 General Performance Analysis

Experimental results in the previous sections illustrate that combining QDPLL with dependency schemes other than D^{triv} potentially increases the performance. Despite additional overhead for constructing and maintaining dependency DAGs, more refined dependency schemes such as D^{tree} and D^{std} might allow to detect more implications and produce shorter learnt constraints. We conclude that the use of an arbitrary dependency scheme D where $D \subset D^{\text{triv}}$ can pay off in practice, provided that the augmented compressed dependency graph $G_{\approx,\subseteq}(D)$ by Definition 4.2.8 on page 60 is implemented carefully.

In this section, we analyze the general performance of DepQBF with D^{std}.[5] We address the effects of optimizations described in Section 5.6.2 and compare DepQBF to other solvers, with and without preprocessing. Parts of the experimental results in this section were previously published in [21, 84, 86] and reported in related conference talks.

Improvements of QDPLL

Table 5.4 shows experimental results of DepQBF 0.1 with D^{std} where optimizations of QDPLL such as pure literal detection, restarts and assignment caching were disabled. The time limit was set to 900 seconds. DepQBF performs best if all of these techniques are enabled. Switching off restarts in DepQBF-nr affects the number of solved satisfiable instances negatively. Without assignment caching but with restarts (DepQBF-nc), the performance is even worse. DepQBF performs worst if neither restarts nor as-

[5]Setup for all experiments reported in Section 5.7.2, unless stated otherwise: Ubuntu Linux 9.04, Intel® Q9550@2.83 GHz, 7 GB/900 seconds memory and time limit.

QBFEVAL'10 (568 formulae)				
	Solved	*Avg. Time*	*SAT*	*UNSAT*
DepQBF	**370**	**337.10**	**165**	**205**
DepQBF-nr	360	352.33	154	206
DepQBF-nc	350	384.66	157	193
DepQBF-np	345	398.12	141	204
DepQBF-ncnr	340	400.24	147	193

Table 5.4: The influence of optimization such as restarts (disabled in DepQBF-nr), assignment caching (disabled in DepQBF-nc) and pure literal detection (disabled in DepQBF-np) on the performance of DepQBF 0.1.

QBFEVAL'10 (568 formulae) – with preprocessing				
	Solved	*Avg. Time*	*SAT*	*UNSAT*
Bloqqer + QxBF + DepQBF	468	197.31 (16.47)	224	244
Bloqqer + DepQBF	466	198.50 (7.69)	223	243
QuBE7.2	435	264.70 (–)	202	233
QxBF+ DepQBF	378	323.19 (7.21)	167	211
QBFEVAL'10 (568 formulae) – without preprocessing				
DepQBF	372	334.60	166	206
QuBE7.2-nopp	319	431.47	144	175
Nenofex	211	573.65	103	108
Quantor 3.0	203	590.15	99	104
squolem 2.02	124	708.80	53	71

Table 5.5: DepQBF and other solvers with and without preprocessing.

signment caching is enabled (DepQBF-ncnr). Pure literal detection, which is disabled in DepQBF-np, has a substantial negative influence on the number of solved satisfiable instances. This is interesting because modern SAT solvers typically do not detect pure literals. As argued in Section 5.3.4, pure literal detection during QBCP might cause additional constraints to become unit. As future work, we want to analyze the actual effects in detail.

Preprocessing and Other Solvers

DepQBF version 0.1 [84] using D^{std} took first place in QBFEVAL'10 [100]. The lower part of Table 5.5 shows statistics for an improved, non-public version of DepQBF and other solvers *without* preprocessing. In fact, QuBE is the only solver listed in Table 5.5 which applies preprocessing natively [55]. Both DepQBF and QuBE are search-based solvers relying on QDPLL with constraint learning whereas all other solvers eliminate variables. On the considered benchmark set, QDPLL outperforms variable elimination. With respect to QDPLL, DepQBF performs better than QuBE7.2-nopp where preprocessing was disabled. However, there is a huge performance gap between

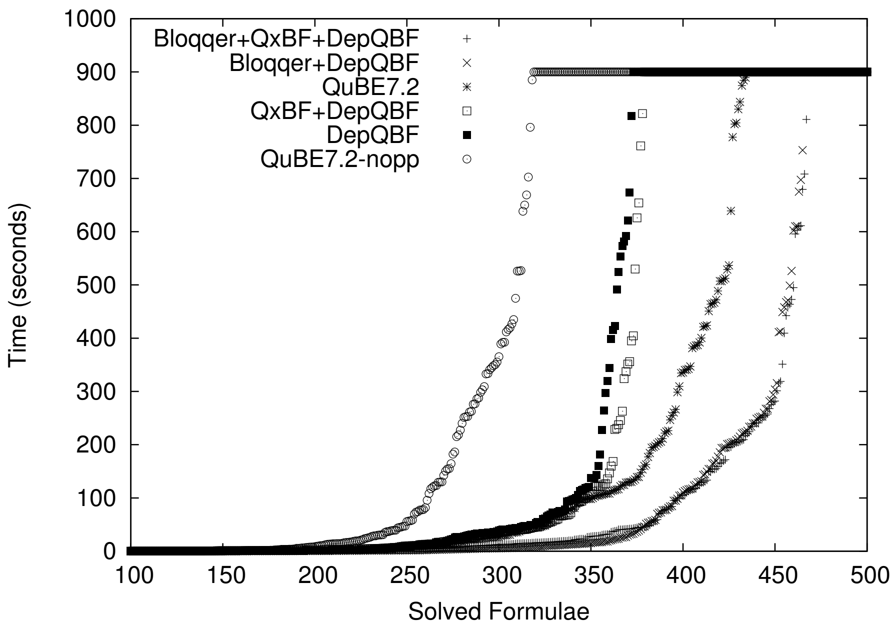


Figure 5.6: Sorted run times of selected solvers from Table 5.5.

solvers with and without preprocessing.

We considered preprocessing in the upper part of Table 5.5. Numbers in parentheses are average times spent on preprocessing by various approaches. QuBE7.2 performs substantially better if preprocessing is enabled, which includes resource bounded variable elimination and equivalence reasoning [55]. We were not able to determine the preprocessing time spent in QuBE7.2. As DepQBF does not include preprocessing natively, we applied two external tools. Bloqqer [21][6] combines blocked clause elimination [71] for QBF with approaches like resource bounded variable elimination and equivalence reasoning inspired from SAT solving [41]. QxBF [86][7] is a preprocessor which detects failed literals in a QBF. The version listed in Table 5.5 applies abstraction-based and SAT-based failed literal detection (see [86] for further details). We allowed at most 80 seconds to be spent in QxBF whereas Bloqqer uses static limits which are not based on time. Altogether, preprocessing and solving time was limited to 900 seconds.[8]

Combining DepQBF with Bloqqer yields a larger performance improvement than with QxBF. By first running Bloqqer and then QxBF, DepQBF solves only two instances more. However, if we consider the number of

[6]See also `http://fmv.jku.at/bloqqer/`.

[7]See also `http://fmv.jku.at/qxbf/`.

[8]Due to limited computational resources, we did not combine QuBE7.2 with Bloqqer and QxBF.

instances that were solved solely by preprocessing then it turns out that QxBF has a considerable positive effect. QxBF solves 30 and Bloqqer 148 instances. The combination Bloqqer + QxBF as applied in Table 5.5 solves 172 instances, which amounts to 36.7% of the 468 instances solved by DepQBF combined with Bloqqer and QxBF. With respect to the full set of 568 instances, 30% are solved *solely* by preprocessing.

The results in Table 5.5 point out that QDPLL without preprocessing as in Figure 5.1 on page 89 is by far not competitive. Further experimental evaluation is necessary to analyze the actual role and interplay of individual preprocessing approaches applied in QuBE7.2, Bloqqer and QxBF.

5.8 Summary

In this chapter, we considered combinations of QDPLL and dependency schemes. The classical variant of QDPLL implicitly relies on the quantifier prefix of the given PCNF, that is the trivial dependency scheme D^{triv}. We extended QDPLL to arbitrary dependency schemes. We may think of a dependency scheme D as an additional parameter of QDPLL. By analyzing the core parts of QDPLL, we figured out that the decision making, dependency checking and constraint reduction in QDPLL must be adapted from D^{triv} to arbitrary schemes D. Apart from that, the high-level workflow of the algorithm does not change. Hence the theoretical framework of dependency schemes introduced in Chapter 3 fits seamlessly into QDPLL, which is a state-of-the-art approach in QBF solving.

In general, it is important to apply one and the same dependency scheme D throughout all parts of QDPLL. Our generalization of QDPLL is independent from the actual dependency scheme. That is, approaches of decision making or dependency checking as presented in Sections 5.4 and 5.5 apply to arbitrary dependency schemes.

In our experimental analysis, QDPLL with the standard dependency scheme D^{std} outperformed combinations of QDPLL with D^{triv} and D^{tree}, which relies on quantifier trees obtained by mini-scoping. We represent dependency schemes as augmented compressed dependency graphs by Definition 4.2.8 on page 60. Despite additional overhead compared to D^{triv}, QDPLL with D^{tree} and D^{std} showed better overall performance.

Chapter 6

Summary and Outlook

The presence of universally and existentially quantified variables in quantified Boolean formulae (QBF) gives rise to dependencies between variables. If variable y depends on another variable x in some QBF ψ, then in general y must not be assigned before x during a semantical evaluation of ψ. Thus variable dependencies restrict the freedom of QBF solvers to assign values to variables. Violations of such restrictions might cause QBF solvers to produce incorrect results. This property is a crucial difference to propositional logic (SAT). SAT solvers are free to assign any variables, all of which are (implicitly) existentially quantified.

Variable dependencies are intrinsic to QBF. For example, given a QBF ψ in prenex CNF (PCNF), the classical QDPLL algorithm requires to assign variables "from left to right" in the ordering of the quantifier prefix of ψ. If variable x occurs to the left of some other, differently quantified variable y in the prefix, then y is *regarded* to depend on x. However, there are QBFs where dependencies of that kind are *spurious*. In this case, even if a QBF solver assigns y before x during semantical evaluation, the result returned by the solver will be correct. Thus spurious dependencies impose unnecessary restrictions to QBF solvers.

We considered dependency schemes as a means of analyzing variable dependencies in PCNFs. A dependency scheme D for a given PCNF ψ is a binary relation over the set of variables in ψ such that if $(x, y) \notin D$ then y does not depend on x. The *theoretical* framework of dependency schemes allows to figure out precisely all dependencies between variables, thereby entirely avoiding spurious ones. However, obtaining such optimal information on dependencies is at least as hard as QBF solving and therefore not feasible in practice.

A trade-off has to be made between optimality and efficiency of computation of dependency schemes. Consequently, dependency schemes which can be computed efficiently, that is in polynomial time, necessarily contain spurious dependencies. Dependency schemes can be compared with respect to the amount of independence that can be identified. We presented the tractable dependency schemes D^{triv}, D^{tree} and D^{std}. All three can be computed by analyzing the syntactic structure of a given PCNF. We pointed out that the standard dependency scheme D^{std} never contains more spurious dependencies than D^{tree} and D^{triv}, which are based on tree-like quantifier structure present in PCNFs and on the quantifier prefixes, respectively. Since $D^{\mathrm{std}} \subseteq D^{\mathrm{tree}} \subseteq D^{\mathrm{triv}}$, D^{std} potentially allows more freedom on a given PCNF than D^{tree} and D^{triv}. Differently from D^{tree}, D^{std} can be computed deterministically. Therefore, we combined D^{std} with search-based QBF solvers relying on QDPLL. This way, QDPLL can profit from additional information on independent variables.

In order to apply some tractable dependency scheme D in QDPLL, first D must be computed and also represented efficiently. Since D is a binary relation over the set V of variables in a given PCNF, D contains $|V| \cdot |V|$ pairs of variables in the worst case. Hence it might be prohibitive to represent D explicitly as a graph, for example, if the value of $|V|$ is large.

We presented a way to compute and represent the standard dependency scheme D^{std} efficiently. By partitioning the set of variables into equivalence classes with respect to dependency information, we obtained a compact dependency graph over classes rather than individual variables. Although our algorithms are tailored towards D^{std}, they might give insights into new algorithms to construct graphs for other dependency schemes. Further, the concept of compact dependency graphs based on equivalence classes can be used to represent arbitrary dependency schemes. Experimental analysis on instances from QBF evaluations showed that the time for computing and representing D^{std} by our approaches is negligible.

Relying on compact dependency graphs as a general way to represent dependency schemes efficiently, we considered to combine QDPLL with *arbitrary* dependency schemes. The original variant of QDPLL implicitly uses D^{triv} where variables must be assigned in the ordering of the quantifier prefix of the given PCNF. Thus combinations of QDPLL with dependency schemes are generalizations of classical QDPLL rather than novel approaches.

We described QDPLL and analyzed its core parts. It turned out that arbitrary dependency schemes can seamlessly be integrated into QDPLL without the need to change its high-level workflow. The crucial parts are maintenance of decision candidates and dependency checking. A decision candidate is a variable which, given the current assignment generated by QDPLL, can safely be assigned as decision variable without risking incorrect results. Maintaining decision candidates with respect to arbitrary dependency schemes generalizes the “left-to-right” policy of D^{triv} in classical QDPLL. Given two variables x and y, dependency checking finds out if $(x, y) \in D$, where D is the dependency scheme applied in QDPLL. This kind of information is necessary for constraint reduction and for the generation of asserting constraints during constraint learning. In general, one and the same dependency scheme D must be used in all parts of QDPLL.

We evaluated combinations of QDPLL with dependency schemes D^{triv}, D^{tree} and D^{std} in a variant of our QBF solver DepQBF. Experimental results on instances from QBF competitions show that QDPLL with D^{std} outperforms both D^{tree} and D^{triv}. However, the performance was only slightly better than with D^{tree}. Still we favour D^{std} because it can be constructed deterministically and is more refined than D^{tree}, that is $D^{\text{std}} \subseteq D^{\text{tree}}$.

We observed more implications, shorter learnt constraints and fewer backtracks for QDPLL with D^{std} or D^{tree} compared to D^{triv}. These observations clearly motivate further research related to dependency schemes which refine D^{std} such as the triangle [113] or quadrangle dependency scheme [51].

Despite promising experimental results, combining QDPLL with dependency schemes other than D^{triv} is not free of costs. Whereas checking if variable y depends on another variable x by D^{triv} is a constant time operation, in general dependency graphs have to be inspected explicitly for other dependency schemes. The approach presented in [62], which allows for dependency checking based on D^{tree} in contant time as well, cannot be applied for arbitrary dependency schemes. We presented approaches based on compact dependency graphs which allow to keep the overhead of dependency checking and decision candidate maintenance low.

It is important to note that dependency schemes are in fact not a new concept. Rather, they are inherent to QBF semantics. Implicitly, classical descriptions of QDPLL [30] apply the ordering of quantifier prefixes in given PCNFs. Such orderings correspond exactly to the trivial dependency scheme D^{triv}. The theoretical framework of dependency schemes [111, 112, 113] allows to generalize the classical QDPLL algorithm naturally from D^{triv} to arbitrary dependency schemes. There is a close interplay between dependency schemes and QBF semantics. Therefore, we believe that dependency schemes D which *strictly* refine D^{std}, that is $D \subset D^{\text{std}}$, have the potential to improve the state-of-the-art in QBF solving considerably.

Outlook

Although we pointed out the potential of using advanced dependency schemes, QDPLL still suffers from severe drawbacks. First, constraint learning is based on resolution over clauses or cubes, which is applied only heuristically. Constraint learning produces a constraint by selecting pivot variables for resolution with respect to the current assignment that was generated by QDPLL. Variable dependencies restrict the set of assignments that can be enumerated by QDPLL. Therefore, pivot selection heuristics in constraint learning are also restricted by variable dependencies. As illustrated by Example 3.3.6 on page 34, restricted constraint learning could miss short proofs which otherwise could be found with general resolution strategies.

Further, the selection of pivot variables in constraint learning is biased towards quantifier types. When generating a clause, all pivot variables are existential. This is due to the fact that clause learning relies on antecedents of existential unit literals detected in clauses during QBCP. Dually, learnt cubes are generated by resolutions over universal variables. Although these pivot-restricted variants of resolution are complete [27], we *conjecture* that allowing clause (cube) resolutions over universal (existential) variables could produce shorter proofs in certain cases. In general, it would be interesting to address proof theoretic properties of QDPLL with dependency schemes.

From the practical side, most QDPLL-based QBF solvers lack features for incremental solving. In contrast to that, support for incremental solving is common in modern SAT solvers like MiniSAT or PicoSAT, for example. In bounded model checking (BMC) [19] based on QBF [14, 73], a sequence $\psi_1, \psi_2, \ldots, \psi_n$ of PCNFs is solved where the structure of ψ_i is typically related to the previously solved instance ψ_{i-1}. A QBF solver could re-use information learnt from ψ_{i-1} in order to solve ψ_i. An approach for BMC of black box designs relying on a specific variant of incremental QBF solving was presented in [88], but we are not aware of a general-purpose incremental QBF solver. Further challenges arise from the use of dependency schemes other than D^{triv} in an incremental setting. So far it is not clear how to efficiently update dependency information which is represented by compact dependency graphs if clauses are added to or removed from an instance. Learnt constraints might have to be discarded if the underlying dependency scheme in QDPLL is modified.

We have not yet considered the role of dependency schemes in preprocessing like blocked clause elimination [21] or failed literal detection [86] for QBF. Dependency schemes might also give rise to smaller certificates if approaches based on resolution proofs are used [8, 95, 105]. In general, optimizations of DPLL which have been proved very effective in SAT solving could be ported to QDPLL for QBF. For example, as experimental results show, our variants of restarts and assignment caching inspired by SAT solvers improve the performance of our solver DepQBF considerably. Further promising approaches are clause minimization techniques like [122], to take a single example.

By the year 2012, there have been about 20 years of active research on practical SAT solving. Continuous improvements led to robust SAT solvers being capable of effectively tackling instances with thousands of variables and millions of clauses. In contrast to that, we are still waiting for the breakthrough of QBF. The success of SAT solving is driven both by the application side and the implementation side. We believe that similar synergy effects are necessary to boost QBF solving and applications. If we take the advent of the classical QDPLL algorithm [30] in the year 1998 as a starting point, then we have about 10 more years to bring QBF solving within a time span of 20 years to where SAT solving is now in the year 2012.

Appendix A

A.1 Cube Learning

A.1.1 Pivot Selection for Cubes

1. Let R be the current resolvent (line `p = get_pivot (R)` in Figure 5.5).

2. Let $P(R) := \{y \mid y = v(l), l \in R, q(l) = \forall, am(l) = U\}$ be the set of universal variables in R which were assigned by the unit literal rule. Set $P(R)$ contains all *potential* pivot variables to be selected.

3. Select $p \in P(R)$ such that p has the maximum trail level of variables in $P(R)$, that is $tl(p) = max(\{tl(y) \mid y \in P(R)\})$.

4. If the tentative resolvent $R \otimes R'$ of R and the antecedent $R' = ante(p)$ of p does *not* contain complementary literals, then variable p is the pivot for the next application of Q-resolution (line `R = constraint_res (R, p, R')` in Figure 5.5).

5. Otherwise, if $\{x, \neg x\} \subseteq (R \otimes R')$ for some variable $x \in R$ then an alternative pivot is selected as follows.

   (a) Let x be one of the variables which occur both positively and negatively in the tentative resolvent $(R \otimes R')$.

   (b) Let $P'(R) := P(R) \setminus \{y \mid y \in P(R), x \not\prec y\} \setminus \{y \mid y \in P(R), \{x', \neg x'\} \subseteq R \otimes R''$ where $R'' = ante(y)\}$. In addition to the restrictions on set $P(R)$ as defined above, the set $P'(R)$ contains only universal variables which depend on variable x. Further, the resolvent of R and the antecedent of variables in $P'(R)$ does not contain complementary literals.

   (c) Select $p \in P'(R)$ such that p has the maximum trail level of variables in $P'(R)$, that is $tl(p) = max(\{tl(y) \mid y \in P'(R)\})$.

   (d) Variable p is the pivot for the next application of Q-resolution (line `R = constraint_res (R, p, R')`).

Definition A.1.1 ([53, 133])**.** Given the current resolvent R. Let $m := max(\{d \mid d = dl(y), y = v(l), l \in R, q(l) = \forall\})$ be the largest decision level of universal variables in R. Clause R is *asserting* if and only if:

1. There is exactly one literal $l \in R$ with $q(l) = \forall$ such that $dl(v(l)) = m$. Let $v_a := v(l)$.
2. The decision variable at decision level m is universal.
3. Variables where v_a depends on must be assigned at decision levels smaller than $m = dl(v_a)$, that is $\forall y \in \{y \mid y = v(l), l \in R, y \prec v_a\} : DL(y) < m$.

If R is asserting then variable v_a is *asserted* by R.

Definition A.1.2. Given an asserting clause R by Definition A.1.1 and the variable v_a asserted by R.

1. Let $DL_\forall(R) := \{d \mid d = dl(y), y = v(l), l \in R, q(l) = \forall, y \neq v_a\}$ be the set of decision levels of universal variables in R, excluding v_a.
2. Let $DL_\prec(R) := \{d \mid d = dl(y), y = v(l), l \in R, y \prec v_a\}$ be the set of decision levels of variables in R where v_a depends on.
3. The *asserting level* d_a with respect to R is $d_a := max(DL_\forall(R) \cup DL_\prec(R))$.

A.2 Brief Biography

Personal Details

Name:	Florian Matthias Lonsing
Private Address:	Im Weideland 1 4060 Leonding Austria
Date and Place of Birth:	12th April 1983 in Linz, Austria

Research

Feb. 2008 – Feb. 2012	Research and Teaching Assistant at the Institute for Formal Models and Verification (FMV), Johannes Kepler Universität Linz. Research Interests: QBF, SAT, Formal Verification. Website: `http://fmv.jku.at/lonsing/`

Education

Feb. 2008 – Apr. 2012	Doctorate Computer Science, Johannes Kepler Universität Linz
Oct. 2005 – Feb. 2008	Master Computer Science, Johannes Kepler Universität Linz
Oct. 2002 – Oct. 2005	Bachelor Computer Science, Johannes Kepler Universität Linz
Okt. 2001 – Apr. 2002	Military Service
Sep. 1993 – Jun. 2001	Grammar School (*Allgemeinbildende Höhere Schule*) in Linz
Sep. 1989 – Jul. 1993	Primary School in Linz

Bibliography

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. The Transitive Reduction of a Directed Graph. *SIAM J. Comput.*, 1(2):131–137, 1972.

[2] B. Aspvall, M. F. Plass, and R. E. Tarjan. A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas. *Inf. Process. Lett.*, 8(3):121–123, 1979.

[3] G. Audemard, J. Lagniez, B. Mazure, and L. Sais. On Freezing and Reactivating Learnt Clauses. In Sakallah and Simon [110], pages 188–200.

[4] G. Audemard and L. Sais. A Symbolic Search Based Approach for Quantified Boolean Formulas. In Bacchus and Walsh [7], pages 16–30.

[5] G. Audemard and L. Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In C. Boutilier, editor, *IJCAI*, pages 399–404, 2009.

[6] A. Ayari and D. A. Basin. QUBOS: Deciding Quantified Boolean Logic Using Propositional Satisfiability Solvers. In M. Aagaard and J. W. O'Leary, editors, *FMCAD*, volume 2517 of *LNCS*, pages 187–201. Springer, 2002.

[7] F. Bacchus and T. Walsh, editors. *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *LNCS*. Springer, 2005.

[8] V. Balabanov and J. R. Jiang. Resolution Proofs and Skolem Functions in QBF Evaluation and Applications. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, volume 6806 of *LNCS*, pages 149–164. Springer, 2011.

[9] P. Beame, H. A. Kautz, and A. Sabharwal. Towards Understanding and Harnessing the Potential of Clause Learning. *J. Artif. Intell. Res. (JAIR)*, 22:319–351, 2004.

[10] E. Ben-Sasson, R. Impagliazzo, and A. Wigderson. Near-Optimal Separation of Treelike and General Resolution. *Electronic Colloquium on Computational Complexity (ECCC)*, 7(5), 2000.

[11] M. Benedetti. Extracting Certificates from Quantified Boolean Formulas. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI*, pages 47–53. Professional Book Center, 2005.

[12] M. Benedetti. Quantifier Trees for QBFs. In Bacchus and Walsh [7], pages 378–385.

[13] M. Benedetti. sKizzo: A Suite to Evaluate and Certify QBFs. In R. Nieuwenhuis, editor, *CADE*, volume 3632 of *LNCS*, pages 369–376. Springer, 2005.

[14] M. Benedetti and H. Mangassarian. QBF-Based Formal Verification: Experience and Perspectives. *JSAT*, 5:133–191, 2008.

[15] A. Bhalla, I. Lynce, J. T. de Sousa, and J. Marques-Silva. Heuristic-Based Backtracking Relaxation for Propositional Satisfiability. *Journal of Automated Reasoning (JAR)*, 35(1-3):3–24, 2005.

[16] A. Biere. Resolve and Expand. In H. H. Hoos and D. G. Mitchell, editors, *SAT (Selected Papers)*, volume 3542 of *LNCS*, pages 59–70. Springer, 2004.

[17] A. Biere. Adaptive Restart Strategies for Conflict Driven SAT Solvers. In Büning and Zhao [29], pages 28–33.

[18] A. Biere. PicoSAT Essentials. *JSAT*, 4(2-4):75–97, 2008.

[19] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In R. Cleaveland, editor, *TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[20] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[21] A. Biere, F. Lonsing, and M. Seidl. Blocked Clause Elimination for QBF. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *LNCS*, pages 101–115. Springer, 2011.

[22] M. L. Bonet, J. L. Esteban, N. Galesi, and J. Johannsen. On the Relative Complexity of Resolution Refinements and Cutting Planes Proof Systems. *SIAM J. Comput.*, 30(5):1462–1484, 2000.

[23] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[24] U. Bubeck. *Model-Based Transformations for Quantified Boolean Formulas*, volume 329 of *Dissertations in Artificial Intelligence*. IOS Press, 2010.

[25] U. Bubeck and H. Kleine Büning. Bounded Universal Expansion for Preprocessing QBF. In Marques-Silva and Sakallah [89], pages 244–257.

[26] H. Kleine Büning and U. Bubeck. Theory of Quantified Boolean Formulas. In Biere et al. [20], pages 735–760.

[27] H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for Quantified Boolean Formulas. *Inf. Comput.*, 117(1):12–18, 1995.

[28] H. Kleine Büning and T. Lettmann. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, New York, NY, USA, 1999.

[29] H. Kleine Büning and X. Zhao, editors. *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *LNCS*. Springer, 2008.

[30] M. Cadoli, A. Giovanardi, and M. Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *AAAI/IAAI*, pages 262–267, 1998.

[31] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An Algorithm to Evaluate Quantified Boolean Formulae and Its Experimental Evaluation. *J. Autom. Reasoning*, 28(2):101–142, 2002.

[32] P. Chatalic and L. Simon. Multi-resolution on Compressed Sets of Clauses. In *ICTAI*, pages 2–10. IEEE Computer Society, 2000.

[33] P. Chatalic and L. Simon. ZRES: The Old Davis-Putman Procedure Meets ZBDD. In D. A. McAllester, editor, *CADE*, volume 1831 of *LNCS*, pages 449–454. Springer, 2000.

[34] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *STOC*, pages 151–158. ACM, 1971.

[35] S. A. Cook and R. A. Reckhow. The Relative Efficiency of Propositional Proof Systems. *J. Symb. Log.*, 44(1):36–50, 1979.

[36] A. Darwiche and K. Pipatsrisawat. Complete Algorithms. In Biere et al. [20], pages 99–130.

[37] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[38] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, 1960.

[39] T. Boy de la Tour. An Optimality Result for Clause Form Translation. *J. Symb. Comput.*, 14(4):283–302, 1992.

[40] N. Dershowitz, Z. Hanna, and A. Nadel. A Clause-Based Heuristic for SAT Solvers. In Bacchus and Walsh [7], pages 46–60.

[41] N. Eén and A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In Bacchus and Walsh [7], pages 61–75.

[42] N. Eén and N. Sörensson. An Extensible SAT-Solver. In Giunchiglia and Tacchella [63], pages 502–518.

[43] U. Egly. On the Value of Antiprenexing. In F. Pfenning, editor, *LPAR*, volume 822 of *LNCS*, pages 69–83. Springer, 1994.

[44] U. Egly, M. Seidl, H. Tompits, S. Woltran, and M. Zolda. Comparing Different Prenexing Strategies for Quantified Boolean Formulas. In Giunchiglia and Tacchella [63], pages 214–228.

[45] U. Egly, M. Seidl, and S. Woltran. A Solver for QBFs in Nonprenex Form. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 477–481. IOS Press, 2006.

[46] U. Egly, M. Seidl, and S. Woltran. A solver for QBFs in negation normal form. *Constraints*, 14(1):38–79, 2009.

[47] U. Egly, H. Tompits, and S. Woltran. On Quantifier Shifting for Quantified Boolean Formulas. In *In Proceedings of the SAT-02 Workshop on Theory and Applications of Quantified Boolean Formulas (QBF-02*, pages 48–61, 2002.

[48] R. Feldmann, B. Monien, and S. Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In *AAAI/IAAI*, pages 285–290. AAAI Press / The MIT Press, 2000.

[49] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[50] A. Van Gelder. Contributions to the Theory of Practical QBF Solving. In *Pragmatics of SAT (POS) Workshop*, 2011.

[51] A. Van Gelder. Variable Independence and Resolution Paths for Quantified Boolean Formulas. In Lee [77], pages 789–803.

[52] I. P. Gent, E. Giunchiglia, M. Narizzano, A. G. D. Rowley, and A. Tacchella. Watched Data Structures for QBF Solvers. In Giunchiglia and Tacchella [63], pages 25–36.

[53] E. Giunchiglia, P. Marin, and M. Narizzano. Reasoning with Quantified Boolean Formulas. In Biere et al. [20], pages 761–780.

[54] E. Giunchiglia, P. Marin, and M. Narizzano. QuBE7.0. *JSAT*, 7(2-3):83–88, 2010.

[55] E. Giunchiglia, P. Marin, and M. Narizzano. sQueezeBF: An Effective Preprocessor for QBFs Based on Equivalence Reasoning. In Strichman and Szeider [125], pages 85–98.

[56] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas Satisfiability Library (QBFLIB), 2001. `http://www.qbflib.org`.

[57] E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability. In R. Goré, A. Leitsch, and T. Nipkow, editors, *IJCAR*, volume 2083 of *LNCS*, pages 364–369. Springer, 2001.

[58] E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for Quantified Boolean Logic Satisfiability. In *AAAI/IAAI*, pages 649–654, 2002.

[59] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for Quantified Boolean Logic satisfiability. *Artif. Intell.*, 145(1-2):99–120, 2003.

[60] E. Giunchiglia, M. Narizzano, and A. Tacchella. Monotone Literals and Learning in QBF Reasoning. In Wallace [128], pages 260–273.

[61] E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. *J. Artif. Intell. Res. (JAIR)*, 26:371–416, 2006.

[62] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantifier Structure in Search-Based Procedures for QBFs. *TCAD*, 26(3):497–507, 2007.

[63] E. Giunchiglia and A. Tacchella, editors. *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *LNCS*. Springer, 2004.

[64] E. I. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT-Solver. In *DATE*, pages 142–149. IEEE Computer Society, 2002.

[65] C. P. Gomes, B. Selman, and H. A. Kautz. Boosting Combinatorial Search Through Randomization. In *AAAI/IAAI*, pages 431–437, 1998.

[66] A. Goultiaeva and F. Bacchus. Exploiting Circuit Representations in QBF Solving. In Strichman and Szeider [125], pages 333–339.

[67] A. Goultiaeva and F. Bacchus. Exploiting QBF Duality on a Circuit Representation. In M. Fox and D. Poole, editors, *AAAI*. AAAI Press, 2010.

[68] A. Goultiaeva, V. Iverson, and F. Bacchus. Beyond CNF: A Circuit-Based QBF Solver. In Kullmann [76], pages 412–426.

[69] A. Haken. The Intractability of Resolution. *Theor. Comput. Sci.*, 39:297–308, 1985.

[70] J. Huang. The Effect of Restarts on the Efficiency of Clause Learning. In M. M. Veloso, editor, *IJCAI*, pages 2318–2323, 2007.

[71] M. Järvisalo, A. Biere, and M. Heule. Blocked Clause Elimination. In J. Esparza and R. Majumdar, editors, *TACAS*, volume 6015 of *LNCS*, pages 129–144. Springer, 2010.

[72] M. Järvisalo, T. A. Junttila, and I. Niemelä. Unrestricted vs. Restricted Cut in a Tableau Method for Boolean Circuits. *Ann. Math. Artif. Intell.*, 44(4):373–399, 2005.

[73] T. Jussila and A. Biere. Compressing BMC Encodings with QBF. *ENTCS*, 174(3):45–56, 2007.

[74] T. Jussila, A. Biere, C. Sinz, D. Kröning, and C. M. Wintersteiger. A First Step Towards a Unified Proof Checker for QBF. In Marques-Silva and Sakallah [89], pages 201–214.

[75] T. Jussila, C. Sinz, and A. Biere. Extended Resolution Proofs for Symbolic SAT Solving with Quantification. In Armin Biere and Carla P. Gomes, editors, *SAT*, volume 4121 of *LNCS*, pages 54–60. Springer, 2006.

[76] Oliver Kullmann, editor. *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *LNCS*. Springer, 2009.

[77] J. Ho-Man Lee, editor. *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *LNCS*. Springer, 2011.

[78] R. Letz. Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In U. Egly and C. G. Fermüller, editors, *TABLEAUX*, volume 2381 of *LNCS*, pages 160–175. Springer, 2002.

[79] M. D. T. Lewis, P. Marin, T. Schubert, M. Narizzano, B. Becker, and E. Giunchiglia. PaQuBE: Distributed QBF Solving with Advanced Knowledge Sharing. In Kullmann [76], pages 509–523.

[80] M. D. T. Lewis, T. Schubert, B. Becker, P. Marin, M. Narizzano, and E. Giunchiglia. Parallel QBF Solving with Advanced Knowledge Sharing. *Fundam. Inform.*, 107(2-3):139–166, 2011.

[81] F. Lonsing and A. Biere. Nenofex: Expanding NNF for QBF Solving. In Büning and Zhao [29], pages 196–210.

[82] F. Lonsing and A. Biere. A Compact Representation for Syntactic Dependencies in QBFs. In Kullmann [76], pages 398–411.

[83] F. Lonsing and A. Biere. Efficiently Representing Existential Dependency Sets for Expansion-based QBF Solvers. *ENTCS*, 251:83–95, 2009.

[84] F. Lonsing and A. Biere. DepQBF: A Dependency-Aware QBF Solver. *JSAT*, 7(2-3):71–76, 2010.

[85] F. Lonsing and A. Biere. Integrating Dependency Schemes in Search-Based QBF Solvers. In Strichman and Szeider [125], pages 158–171.

[86] F. Lonsing and A. Biere. Failed Literal Detection for QBF. In Sakallah and Simon [110], pages 259–272.

[87] P. Manolios and D. Vroon. Efficient Circuit to CNF Conversion. In Marques-Silva and Sakallah [89], pages 4–9.

[88] P. Marin, C. Miller, M. Lewis, and B. Becker. Verification of Partial Designs Using Incremental QBF Solving. In *DATE*. IEEE, 2012.

[89] J. Marques-Silva and K. A. Sakallah, editors. *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *LNCS*. Springer, 2007.

[90] A. R. Meyer and L. J. Stockmeyer. The Equivalence Problem for Regular Expressions with Squaring Requires Exponential Space. In *SWAT (FOCS)*, pages 125–129. IEEE Computer Society, 1972.

[91] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *DAC*, pages 272–277, 1993.

[92] M. N. Mneimneh and K. A. Sakallah. Computing Vertex Eccentricity in Exponentially Large Graphs: QBF Formulation and Solution. In Giunchiglia and Tacchella [63], pages 411–425.

[93] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535. ACM, 2001.

[94] M. Narizzano, C. Peschiera, L. Pulina, and A. Tacchella. Evaluating and Certifying QBFs: A Comparison of State-of-the-Art Tools. *AI Commun.*, 22(4):191–210, 2009.

[95] A. Niemetz. Extracting and Checking Q-Resolution Proofs from a State-Of-The-Art QBF Solver. Master's thesis, Johannes Kepler Universität, Linz, Austria, 2012 (to appear).

[96] A. Nonnengart and C. Weidenbach. Computing Small Clause Normal Forms. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 335–367. Elsevier and MIT Press, 2001.

[97] O. Olivo and E. Allen Emerson. A More Efficient BDD-Based QBF Solver. In Lee [77], pages 675–690.

[98] R. Paige and R. E. Tarjan. Three Partition Refinement Algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[99] G. Pan and M. Y. Vardi. Symbolic Decision Procedures for QBF. In Wallace [128], pages 453–467.

[100] C. Peschiera, L. Pulina, A. Tacchella, U. Bubeck, O. Kullmann, and I. Lynce. The Seventh QBF Solvers Evaluation (QBFEVAL'10). In Strichman and Szeider [125], pages 237–250.

[101] F. Pigorsch and C. Scholl. Exploiting Structure in an AIG Based QBF Solver. In *DATE*, pages 1596–1601. IEEE, 2009.

[102] F. Pigorsch and C. Scholl. An AIG-Based QBF-solver Using SAT for Preprocessing. In Sachin S. Sapatnekar, editor, *DAC*, pages 170–175. ACM, 2010.

[103] K. Pipatsrisawat and A. Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In Marques-Silva and Sakallah [89], pages 294–299.

[104] D. A. Plaisted and S. Greenbaum. A Structure-Preserving Clause Form Translation. *J. Symb. Comput.*, 2(3):293–304, 1986.

[105] M. Preiner. Extracting and Validating Skolem/Herbrand Function-Based QBF Certificates. Master's thesis, Johannes Kepler Universität, Linz, Austria, 2012 (to appear).

[106] QBFLIB. QDIMACS Standard v1.1, 2005. `http://www.qbflib.org/qdimacs.html`.

[107] S. Reimer, F. Pigorsch, C. Scholl, and B. Becker. Integration of Orthogonal QBF Solving Techniques. In *DATE*, pages 149–154. IEEE, 2011.

[108] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[109] V. Ryvchin and O. Strichman. Local Restarts. In Büning and Zhao [29], pages 271–276.

[110] K. A. Sakallah and L. Simon, editors. *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *LNCS*. Springer, 2011.

[111] M. Samer. Variable Dependencies of Quantified CSPs. In I. Cervesato, H. Veith, and A. Voronkov, editors, *LPAR*, volume 5330 of *LNCS*, pages 512–527. Springer, 2008.

[112] M. Samer and S. Szeider. Backdoor Sets of Quantified Boolean Formulas. In Marques-Silva and Sakallah [89], pages 230–243.

[113] M. Samer and S. Szeider. Backdoor Sets of Quantified Boolean Formulas. *Journal of Automated Reasoning (JAR)*, 42(1):77–97, 2009.

[114] H. Samulowitz, J. Davies, and F. Bacchus. Preprocessing QBF. In F. Benhamou, editor, *CP*, volume 4204 of *LNCS*, pages 514–529. Springer, 2006.

[115] C. E. Shannon. The Synthesis of Two Terminal Switching Circuits. *Bell System Technical Journal*, 28(1):59–98, 1949.

[116] J. P. Marques Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In P. Barahona and J. J. Alferes, editors, *EPIA*, volume 1695 of *LNCS*, pages 62–74. Springer, 1999.

[117] J. P. Marques Silva, I. Lynce, and S. Malik. Conflict-Driven Clause Learning SAT Solvers. In Biere et al. [20], pages 131–153.

[118] J. P. Marques Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[119] C. Sinz and M. Iser. Problem-Sensitive Restart Heuristics for the DPLL Procedure. In Kullmann [76], pages 356–362.

[120] A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logics. *J. ACM*, 32(3):733–749, 1985.

[121] T. Skolem. *Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit und Beweisbarkeit mathematischen Sätze nebst einem Theoreme über dichte Mengen.* Translation in: From Frege to Gödel, van Heijenoort, Harvard Univ. Press, 1971.

[122] N. Sörensson and A. Biere. Minimizing Learned Clauses. In Kullmann [76], pages 237–243.

[123] L. J. Stockmeyer. The Polynomial-Time Hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976.

[124] L. J. Stockmeyer and A. R. Meyer. Word Problems Requiring Exponential Time: Preliminary Report. In *STOC*, pages 1–9. ACM, 1973.

[125] O. Strichman and S. Szeider, editors. *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11 - July 14, 2010. Proceedings*, LNCS. Springer, 2010.

[126] R. E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM*, 22(2):215–225, 1975.

[127] G. S. Tseitin. On the Complexity of Derivation in Propositional Calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 1968.

[128] M. Wallace, editor. *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *LNCS*. Springer, 2004.

[129] C. Wrathall. Complete Sets and the Polynomial-Time Hierarchy. *Theoretical Computer Science*, 3(1):23 – 33, 1976.

[130] Y. Yu and S. Malik. Validating the Result of a Quantified Boolean Formula (QBF) Solver: Theory and Practice. In T. Tang, editor, *ASP-DAC*, pages 1047–1051. ACM Press, 2005.

[131] L. Zhang. On Subsumption Removal and On-the-Fly CNF Simplification. In Bacchus and Walsh [7], pages 482–489.

[132] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In *ICCAD*, pages 279–285, 2001.

[133] L. Zhang and S. Malik. Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. In L. T. Pileggi and A. Kuehlmann, editors, *ICCAD*, pages 442–449. ACM, 2002.

[134] L. Zhang and S. Malik. Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In P. Van Hentenryck, editor, *CP*, volume 2470 of *LNCS*, pages 200–215. Springer, 2002.