Solving Quantified Boolean Formulas

by

Horst Samulowitz

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Copyright © 2007 by Horst Samulowitz

Abstract

Solving Quantified Boolean Formulas

Horst Samulowitz

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2007

Many real-world problems do not have a simple algorithmic solution and casting these problems as search problems is often not only the simplest way of casting them, but also the most efficient way of solving them.

In this thesis we will present several techniques to advance search-based algorithms in the context of solving quantified boolean formulas (QBF). QBF enables complex real-world problems including planning, two-player games and verification to be captured in a compact and quite natural fashion. We will discuss techniques ranging from straight forward pre-processing methods utilizing strong rules of inference to more sophisticated online approaches such as dynamic partitioning.

Furthermore, we will show that all of the presented techniques achieve an essential improvement of the search process when solving QBF. At the same time the displayed empirical results also reveal the orthogonality of the different techniques with respect to performance. Generally speaking each approach performs well on a particular subset of benchmarks, but performs poorly on others. Consequently, an adaptive employment of the available techniques that maximizes the performance would result in further improvements. We will demonstrate that such an adaptive approach can pay off in practice, by presenting the results of using machine learning methods to dynamically select the best variable ordering heuristics during search.

Contents

1 Introduction **1**

1.0.1 Contributions Of This Work 8

1.0.2 Structure of this thesis 9

2 Solving QBF with Search **12**

2.1 Introduction 12

2.2 Quantified Boolean Formulas 13

2.3 Search-Based QBF Solver 20

2.3.1 DPLL for QBF 21

2.3.2 Learning 24

2.3.3 Space Complexity 32

2.3.4 Extensions of the Basic Framework 33

2.4 Reduction to SAT 36

2.4.1 Variable Elimination for QBF 37

2.4.2 Resolve and Expand 37

2.4.3 Symbolic Quantifier Elimination 39

2.4.4 Skolemization 41

2.4.5 SAT Sampling 45

2.5 Experimental Evaluation 45

2.6 Conclusions 46

3 Extended Binary Resolution **48**

3.1 Introduction 48

3.2 Hyper Binary resolution 49

3.3 Hyper Binary Resolution in QBF 50

3.4 Preprocessor 55

3.5 Dynamic Employment 56

3.6 Empirical Results 60

3.6.1 Performance of the Preprocessor 60

3.6.2 Impact of Preprocessing 63

3.6.3 Impact of Dynamic Application 69

3.6.4 Detailed Performance Analysis of 2clsQ 70

3.6.5 Filtering out instances best solved by variable elimination 73

3.7 Related Work 78

3.8 Extensions of Preprocessing 80

3.9 Conclusion 82

4 Using SAT in QBF **83**

4.1 Introduction 83

4.2 SQBF 84

4.3 Integration of SAT and QBF 88

4.4 Formal Results 90

4.5 Empirical Results 91

4.5.1 Benchmark Settings 91

4.5.2 Discussion 92

4.6 Conclusions 96

5 Dynamically Partitioning **98**

5.1 Introduction 98

5.1.1 Partitioning QBF 99
5.1.2 Partitioning for a Search Based QBF Solver 100
5.1.3 Quantifier Trees 102
5.2 Learning with Partitioning 102
5.2.1 Clause Learning 103
5.2.2 Cube Learning 104
5.2.3 Triggering Learnt cubes 112
5.3 Soundness and Completeness 114
5.4 Implementation 116
5.5 Experimental Results 118
5.5.1 2clsQ vs. 2clsP 118
5.5.2 2clsP vs. Other solvers 120
5.5.3 State of the art solver 121
5.6 Conclusions 123

6 Adaptive Search **125**

6.1 Introduction 125
6.2 Dynamic Prediction 127
6.2.1 An Adaptive Search-Based QBF Solver 128
6.2.2 Heuristics 130
6.2.3 Feature Choice 132
6.2.4 Classification 135
6.3 Experimental Evaluation 136
6.3.1 Variable Elimination vs. Search 137
6.3.2 Predicting Heuristics 140
6.4 Conclusions and Future Work 144

7 Conclusions and Future Work **146**

7.1 Summary 146

7.2 Conclusions From This Work 147

7.3 Future Work 149

Bibliography **152**

Chapter 1

Introduction

Modeling and solving real-world problems are two cornerstones of *Artificial Intelligence* (AI). In particular, the sub-area of *Automated Reasoning* tackles these challenges by developing modeling languages to encode real-world problems as well as algorithms to solve them. Here we focus on approaches that are mainly based on mathematical logic that have already proven successful on a wide range of real-world applications such as planning, verification, and robotics (see e.g., [50]).

There exists a range of distinct modeling languages that differ in an essential fashion. The key difference between these languages is there expressiveness. For instance, propositional logic [24] has limited abilities to represent real-world problems while first-order logic [24] provides more powerful features and properties that allow it to represent world-knowledge more compactly [72].

However, the level of expressiveness also correlates with the efficiency of reasoning within the corresponding modeling language. For example, it is a well-known fact that classical logical entailment is undecidable in the first-order case [24]. In contrast, reasoning within less expressive propositional logic is not only decidable [24], but also computationally much more feasible [63, 50]. Consequently, the challenge is to determine the right balance between the two measures within the given problem domain in order

to obtain appropriate representational power along with sufficiently efficient inference. There exist various approaches that deal with this problem, but none of them is able to neutralize the trade-off between expressiveness and efficiency (see e.g., [65], [89]).

The fundamental steps underlying automated reasoning are summarized in Figure 1.1. The domain-specific problem instance (e.g., planning, scheduling, games) is encoded into a modeling language (e.g., propositional or first-order logic). An inference procedure which operates on this language solves the original problem in its new representation and returns a solution.

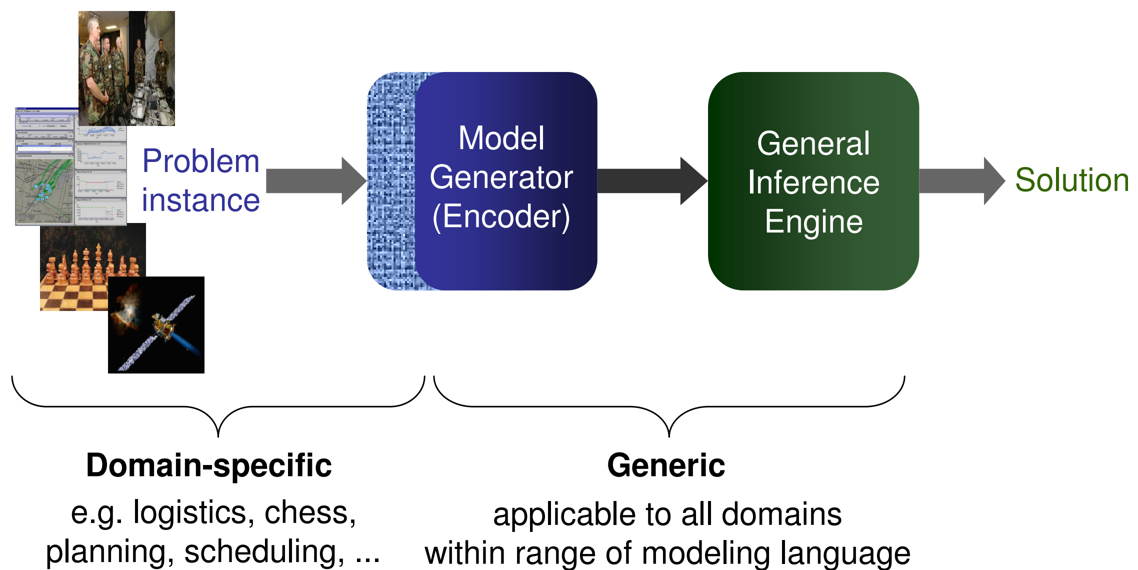


Figure 1.1: The fundamental steps underlying automated reasoning. The domain-specific problem instance is encoded into a modeling language. An inference procedure which operates on this model solves the original problem in its new representation and returns a solution. [Credit: A. Sabharwal, B. Selman, *Beyond Traditional SAT Reasoning*, AAAI 2007 Tutorial]

As a matter of a fact, propositional logic, with its relatively low representational power is for now one of the most successful modeling languages in AI. This is mainly due to advances in propositional inference in the last 50 years. The problem of reasoning within propositional logic is known as the Satisfiability Problem (SAT).

More precisely, SAT is the problem of deciding if there exists an assignment for variables in a propositional formula that makes the formula true. For more details and a thorough background on SAT the reader is referred to, for instance, [63] or [50].

One of the first algorithms presented in 1960 to solve SAT was the Davis-Putnam algorithm (DP) [31] which is based on variable elimination. Nowadays the search-based algorithm presented two years later, called DPLL [30], is still the state-of-the-art framework for solving SAT. Of course, the original DPLL algorithm has been extended in many essential ways. A few of these extensions are clause learning, branching heuristics, intelligent backtracking, and parallelization. These techniques have a significant impact on the performance of a SAT solver. Again for a summary of these techniques the reader is referred to, e.g., [42], [63], [50]. In addition the development of data structures tailored for SAT solving has had an enormous impact on the performance of SAT solvers (e.g., watched literals [73]).

SAT was the first problem shown to be NP-complete [104] and is therefore intractable in general. Since then (1971) there has been a large amount of research on algorithms for solving the satisfiability problem [42, 50]. Moreover it has been shown that there are many problem instances modeled within SAT originating from real-world domains that can be solved extremely efficiently. In fact, in the last two decades the complexity of problem instances that have been modeled, and solved in SAT has increased in a rather dramatic fashion. While in 1980 the problem instances that could be solved consisted of approximately 100 variables and 300 constraints, modeled real-world problems in 2007 contain up to $1,000,000$ variables and about $5,000,000$ constraints (see e.g.,[63, 50]).

Figure 1.2 gives an intuitive insight into the complexity of different problem domains.

In general, a SAT solver is faced with $2^{|Variables|}$ combinations of possible complete assignments to the variables in the given problem instance. The figure shows in an approximate fashion how different problem domains relate to this basic measure of complexity. The applications shown (e.g., formal verification, planning) in the figure reflect only a subset of the wide range of applications that exist for SAT. Clearly, there is a demand to be able to solve very complex real-world problems. Again the reader is referred to the rich literature on SAT and its applications (e.g.,[63, 50, 69, 42, 88, 103, 108, 62, 18, 17, 16]). This strong demand for efficient solvers and the already achieved success are the reasons that SAT solvers improved that drastically and are still the focus of several research groups around the world.

While these achievements of SAT are not only very impressive but also extremely useful in practice, it appears to be the case that SAT reaches its limits in real-world modeling and solving when the problem domain becomes more complex (e.g., involves uncertainty, multi-agent scenarios, etc.). In fact, Figure 1.2 illustrates this downside of SAT as well: the size of the propositional encoding required by practical applications becomes enormous. In other words, SAT is not sufficiently expressive to model complex problems compactly. In general, problems that are in a higher complexity class than SAT (NP-complete) such as general AI planning or two-player games like Reversi (both PSPACE-complete [50, 57]) cannot be compactly represented in SAT. Ali et al. [1] make the limits of the propositional encoding more apparent. They show that the propositional encoding of some fault-diagnosis problem requires space that is beyond feasible memory bounds. We return to this specific example later on in this section. Now we first present a modeling language that is more expressive than pure propositional logic.

Quantified boolean formulas (QBF) are a powerful generalization of the satisfiability problem in which variables are allowed to be universally as well as existentially quantified. The ability to nest universal and existential quantification in arbitrary ways makes QBF considerably more expressive than SAT. Due to alternating quantification QBF is for

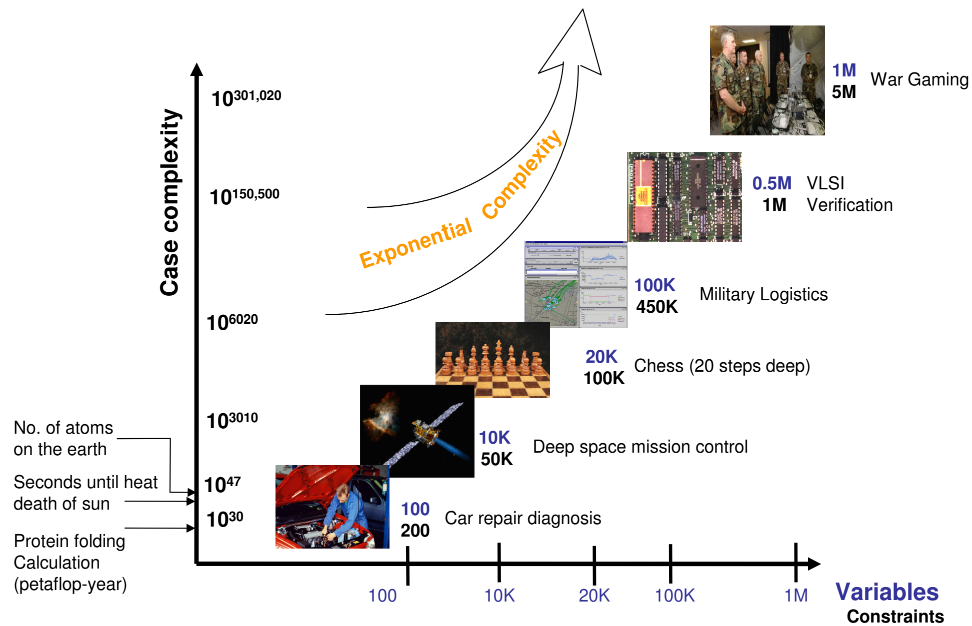


Figure 1.2: The case complexity of different real-world application. Shown are rough estimates for propositional reasoning in the corresponding problem domain. [Credit: Kumar, DARPA, *Computer World Magazine* and A. Sabharwal, B. Selman, *Beyond Traditional SAT Reasoning*, AAAI 2007 Tutorial]

example able to model adversarial settings quite naturally and compactly (e.g., two-player games like chess, multi-agent scenarios). In contrast, SAT is not able to deal with adversarial reasoning naturally and effectively since it belongs to a higher complexity class than SAT. In general it is the case that any NP problem can be compactly encoded in SAT while QBF allows us to compactly encode any PSPACE problem: QBF is PSPACE-complete [102].

This expressiveness opens a much wider range of potential application areas for a QBF solver, including areas like automated planning (particularly conditional planning),

non-monotonic reasoning, electronic design automation, scheduling, model checking and verification (e.g., fault-diagnosis), strategic decision making, multi-agent scenarios, problem domains with incomplete or probabilistic knowledge, see e.g., [85, 84, 37, 36, 84, 32, 1, 52, 101, 15, 60, 61, 68, 82].

The difficulty, however, is that QBF is in practice a much harder problem to solve than SAT. (It is also harder theoretically, assuming that PSPACE $\neq$ NP). One indication of this practical difficulty is the fact that current QBF solvers are typically limited to problems that are about 1-2 orders of magnitude smaller than the instances solvable by current SAT solvers (1000's of variables rather than 100,000's) (see e.g.,[63], [50]).

Nevertheless, this limitation in the size of the instances solvable by current QBF solvers is somewhat misleading. In particular, many problems have a much more compact encoding in QBF than in SAT due to its greater expressiveness.

In order to further illustrate the impact of the increased expressiveness available in QBF, we briefly touch on an example application in the area of formal verification as discussed in [61]. The following example illustrates the fact that QBF is more succinct than SAT. In bounded model checking (BMC) the correctness of a given system—normally a finite state machine—is verified. In general, it is tested if a created system complies with the desired design rules. In particular, it is checked if for instance an undesired state S_u is reachable in k steps from the initial state S_0 ($R_k(S_0, S_u)$) given the transition relations $T(S_i, S_j)$ between two states of the system. The classical SAT-based BMC encoding can be formalized in the following way [61] [15]:

$$R_k(S_0, S_u) = \exists S_1, S_2, ..S_{k-1} : I(S_0) \wedge F(S_u) \wedge \bigwedge_{i=0}^{k-1} T(S_i, S_{i+1}), \tag{1.1}$$

In Equation 1.1, S_i denote possible states in the system and $I(S)$ (respectively $F(S)$) checks if S is the initial (respectively final) state. Hence, the reachability analysis is performed by unrolling the transition relation k times and determining the truth value of the propositional part of Equation 1.1. As [61] points out the number copies of the

transition relation T is identical to the depth k of the reachability analysis. The bound k is incremented iteratively so that in each additional iteration the analysis verifies if one of the final states is reachable in one more step. Since in each iteration the number of transition relations is increased by the corresponding iteration depth the number of copies of the transition relations grows in size. Hence, when performing a complete verification the SAT solver is faced with an exponential number of copies of the transition relations. Clearly, the size of the SAT encoding can be enormous, and in fact this growth in the encoding is the main limitation in many application domains. With QBF the same problem can be modeled in the following way [61]:

$$R_k(S_0, S_u) = \exists S_1, S_2, ..S_{k-1} : I(S_0) \wedge F(S_u) \wedge \forall U, V(\bigvee_{i=0}^{k-1}(U \leftrightarrow S_i) \wedge (V \leftrightarrow S_{i+1})) \implies T(U, V) \tag{1.2}$$

It is apparent from Equation 1.2 that the transition relations only appear once in the QBF encoding regardless of the number of iterations. Only the term $(U \leftrightarrow S_i) \wedge (V \leftrightarrow S_{i+1})$ has to be added in each iteration. Since the transition relations require the largest amount of space in the encoding the savings with respect to the size of the encoding are quite essential. In fact, it has been shown that due to the increase of representational power this way of modeling bounded model checking is able to reduce the memory requirements drastically in practice [61] [60].

In the similar problem domain of fault diagnosis an innovative application of QBF to hardware debugging in electrical circuits is given by [1] [52] as mentioned earlier. Analogous to the previous example the encoding in QBF does not require the explicit unrolling of the complete circuit for each additional time step iteration during debugging as the SAT encoding does. However, [52] do not only show that the QBF encoding of the problem is many times smaller than an equivalent SAT encoding, but they are also able to outperform SAT solvers on the SAT encoding while using a QBF solver on the QBF encoding [52]. Results like this demonstrate the potential of QBF and the importance of continuing to improve QBF solvers.

Finally, it is also worthwhile to point out that the research on QBF is a much younger and less active area than SAT. Research on QBF solvers only started about a decade ago. Only recently have more researchers become involved with this new modeling language and its inference engines. While SAT modeling and solving has reached a level at which it is commercially used in a wide range of industries, QBF research has still ways to go in order to achieve a similar impact.

1.0.1 Contributions Of This Work

In this thesis we present a range of novel approaches that make progress toward an efficient and practically applicable QBF solver. Referring back to Figure 1.1 the work presented here is only concerned with improving the inference engine. Recently, the topic of how to encode real-world problems both efficiently and effectively in QBF received more attention and several new insights and techniques have been proposed (see, e.g., [2]).

We mainly focus on several extensions of the DPLL algorithm for QBF. While the core of this thesis concentrates on novel solving techniques based on inference, the last part introduces a machine learning method that shows that it is possible to automatically decide in which context to apply which solving technique in order to achieve the best performance.

All the inference procedures presented in this thesis were able to improve state of the art in QBF solving. While we provide the reader with an extensive amount of benchmarking results in order to illustrate the impact of the introduced techniques, the performance of our proposed techniques has been also independently verified by the QBF competition [49]. In fact, a combination of the approaches and insights that we present in Chapter 3 and 4 were able to rank first, second, and third in the international competition of 2006. The preprocessing strategy we present in Chapter 3 is especially effective and the impact of preprocessing seems to be much stronger than with SAT.

We show in Chapter 4 that the dynamic employment of SAT as a look-ahead technique

within QBF solving can be extremely effective. This approach, based on the relaxation of the underlying problem instance, integrates key technologies—especially learning—incorporated in SAT solving very tightly within a QBF solver.

Similar observations also hold for the results achieved by dynamically partitioning QBF during solving that we present in Chapter 5. While in SAT there exists no empirical evidence that the effort of determining if the problem falls into independent components appears to be worthwhile, the benchmark results by our partitioning-based solver are much more encouraging.

Besides the empirical results we also present a number of new theoretical results. For instance, in Chapter 5 we show how learning in a partitioned-based solver can be accomplished soundly. In general, we show that all our extensions retain the soundness and completeness properties of the underlying DPLL algorithm.

In addition to these previously mentioned inference techniques we also present results on adaptive search in the context of QBF. In fact, the work discussed in Chapter 6 is the first one reported in this area. We show that it is possible to extract features out of QBF instances that are sufficiently strong to discriminate between problem instances that differ in their structural properties. In the static context we present a linear classifier that is able to predict which QBF solver to employ in order to achieve the best performance. In addition, we show that these structural properties are altered during the solving process and that it is possible to automatically adjust the inference engine to take these changes into account by altering the solving strategy. Again, we can demonstrate that this technique improves our ability to solve QBF.

1.0.2 Structure of this thesis

This thesis falls naturally into seven parts, which are relatively independent:

- Solving QBF with Search (Chapter 2),

- Preprocessing and solving QBF with strong rules of inference (Chapter 3),
- Relaxation of QBF by employing a SAT solver during QBF solving (Chapter 4),
- Dynamic partitioning during QBF solving (Chapter 5),
- Employing machine learning techniques to automatically adapt the solving strategy (Chapter 6),
- Assessment of the work done in this thesis and ideas for future work (Chapter 7).

Chapter 2 forms the main body of the thesis, and is devoted to introducing the necessary background as well as some novel definitions and properties of QBF. Reading Chapter 2 sets the appropriate context required to enable a complete understanding of the different approaches that we present in the subsequent chapters. Furthermore, this chapter provides several novel insights on QBF and its relation to search. In addition, we also discuss related work in this chapter.

Based on the information and insights provided in Chapter 2 the remaining chapters can be read completely independently of each other.

Chapter 3, Chapter 4, and Chapter 5 are concerned with new inference methods in the context of QBF. In Chapter 3 a version of extended binary clause reasoning is employed in order to process QBF instances in a static as well as a dynamic fashion. The underlying idea is to perform more reasoning in order to achieve a better performance in QBF solving.

An algorithm that utilizes a SAT solver by relaxing the original QBF formula is discussed in Chapter 4. The SAT solver functions as a powerful look ahead technique during QBF solving. Finally, Chapter 5 tries to dynamically divide the underlying problem instance and to achieve better performance by tackling the sub-problems independently instead of solving the complete problem as a whole.

In Chapter 6 a novel framework to solving QBF is developed that combines a search-based QBF solver with machine learning techniques. The employment of machine learning techniques allows us to automatically adapt the solving strategy and further improve our ability to solve QBF.

Chapter 7 summarizes and assesses the work presented in this thesis and further ideas for future work are presented.

Chapter 2

Solving QBF with Search

2.1 Introduction

In this chapter we introduce the basics required to enable a complete understanding of the different approaches to solve quantified boolean formulas (QBFs) in the subsequent chapters. In addition we survey several techniques that have been applied to tackle the problem of solving quantified boolean formulas. An outline of this chapter is as follows.

First, the necessary background is provided. While presenting the required notations and definitions at the beginning of this chapter we also cover the semantics of QBF in more detail. Based on these fundamentals we present a framework for a generic time-exponential search-based QBF solver based on DPLL [30]. We present in detail how search-based QBF solvers relate to the semantics of QBF. Given this understanding of a elementary search-based solver we move our focus to a discussion of learning techniques in the context of QBF. In particular, we concentrate on solution learning which is a newly introduced technique in QBF solving. We also provide a novel definition of solution learning in order to explain how solution learning is embedded within a search-based QBF solver. We close the section on backtracking search with a discussion on several other extensions of this basic framework that have been developed to improve the basic

algorithm.

Subsequently we address the class of QBF solvers that solve QBF by reducing it to SAT. These techniques are mainly based on variable elimination [31] and skolemization [100].

Furthermore we briefly discuss the classical trade off between time and space in the context of QBF solving. We close with some conclusions.

2.2 Quantified Boolean Formulas

A quantified boolean formula has the form $\vec{Q}.F$, where F is a propositional formula and $\vec{Q}$ is a sequence of quantified variables ($\forall x$ or $\exists x$). We require that no variable appear twice in $\vec{Q}$ and that all variables in F appear in $\vec{Q}$ (i.e., F contains no free variables). In this thesis we restrict our attention to propositional formula F expressed in CNF. The main motivation of such a restriction is that most QBF solvers (including ours) are restricted to CNF inputs.

A **quantifier block** qb of $\vec{Q}$ is a maximal contiguous subsequence of $\vec{Q}$ where every variable in qb has the same quantifier type. We order the quantifier blocks by their sequence of appearance in $\vec{Q}$: $qb_1 \leq qb_2$ iff qb_1 is equal to or appears before qb_2 in $\vec{Q}$. Each variable x in F appears in some quantifier block $qb(x)$, and the ordering of the quantifier blocks imposes an order on the variables. For two variables x and y we say that $x \leq_q y$ iff $qb(x) \leq qb(y)$. Note that the variables in the same quantifier block are unordered. The following definition summarizes the most important properties of a variable and its relations to other variables.

Definition 1

1. *For two variables x and y, $x \leq_q y$ if $qb(x) \leq qb(y)$ and $x <_q y$ if $qb(x) < qb(y)$.*

2. *Variable x is* ***universal*** *(**existential**) if its quantifier in $\vec{Q}$ is $\forall$ ($\exists$).*

3. *A variable x is* ***downstream*** *(**upstream**) of a set of variables V if (1) $x \notin V$ and (2) $\forall y. y \in V \implies y \leq_q x$ ($\forall y. y \in V \implies x \leq_q y$)). That is, x is not a member of V and appears no sooner (later) in the quantifier sequence $\vec{Q}$ than the last (first) quantifier block containing elements of V.*

4. *A variable x is* ***maximal*** *(**minimal**) in a set of variables V if (1) $x \in V$ and (2) $\forall y. y \in V \implies y \leq_q x$ ($\forall y. y \in V \implies x \leq_q y$). That is x is a member of V and appears in the highest (lowest) quantifier block amongst all variables of V.*

Each variable x generates two literals ℓ and $\neg\ell$. As a slight abuse of notation we often use a literal ℓ to refer to ℓ's variable. For example, when we say that ℓ is maximal in a set of variables V, we mean that ℓ's variable is maximal in V. Similarly, we might assert that ℓ is universal if ℓ's variable is universal, that $\ell_1 <_q \ell_2$ if ℓ_1's variable is $<_q$ than ℓ_2's variable, or that ℓ is added to a set of variables V if ℓ's variable is added to V.

For example, $\exists e_1 e_2. \forall u_1 u_2. \exists e_3 e_4. (e_1, \neg e_2, u_2, e_4) \wedge (\neg u_1, \neg e_3)$ is a QBF with the quantifier prefix $\vec{Q} = \exists e_1 e_2. \forall u_1 u_2. \exists e_3 e_4$ and the propositional formula F equal to the two clauses $(e_1, \neg e_2, u_2, e_4)$ and $(\neg u_1, \neg e_3)$. The quantifier blocks in order are $\exists e_1 e_2$, $\forall u_1 u_2$, and $\exists e_3 e_4$, and we have, e.g., that, $e_1 <_q e_3$, $u_1 <_q e_4$, u_1 is universal, e_4 is existential, e_4 is downstream of the set $\{u_2, e_3\}$, e_3 is maximal in the set $\{u_2, e_3\}$, and u_2 is upstream of the set $\{u_1, e_3, e_4\}$. The empty set of literals has by definition a quantifier level that is lower than any quantifier level of a variable contained in $\vec{Q}.F$ (e.g.,0).

A QBF instance can be reduced by assigning values to some of its variables. The **reduction** of a formula $\vec{Q}.F$ by a literal ℓ (denoted by $\vec{Q}.F|_\ell$) is the new formula $\vec{Q}'.F'$ where F' is F with all clauses containing ℓ removed and the negation of ℓ, $\neg\ell$, removed from all remaining clauses, and $\vec{Q}'$ is $\vec{Q}$ with the variable of ℓ and its quantifier removed. For example, $\forall xz. \exists y. (\neg y, x, z) \wedge (\neg x, y)|_{\neg x} = \forall z. \exists y(\neg y, z)$. Note that we also have that $\forall z. \exists y(\neg y, z)$ is equivalent to $\forall xz. \exists y(\neg y, z)$. That is if a variable does not appear in F its presence in $\vec{Q}$ is irrelevant. In addition given a set of literals T we also denote by $\vec{Q}.F|_T$ the reduction of F by all literals in T.

Furthermore, we also define the notion of **universal reduction** as introduced by [22]. A universal variable u is called a **tailing universal** in a clause c if for every existential variable $e \in c$ we have that $e <_q u$. The universal reduction of a clause c is the process of removing all tailing universals from c. The universal reduction of a QBF formula $\vec{Q}.F$ is the process of applying universal reduction to all of the clauses of F. As we will show later universal reduction produces an equivalent QBF formula.

For example, the QBF $\exists e_1 \forall u \exists e_2 (e_1, u) \wedge (u, e_2)$ can be universally reduced to the equivalent QBF $\exists e_1 \forall u \exists e_2 \exists (e_1) \wedge (u, e_2)$. Note that due to the CNF input format QBF instances always have a quantifier prefix that ends with a quantifier block of existentials after universal reduction has been performed as it is standard.

We call the application of unit propagation and universal reduction until closure **Q-propagation**, and denote by $QProp(\vec{Q}.F)$ the new formula that results from Q-propagation. In Q-propagation any universal reduction steps are always performed prior to any unit propagation steps: a unit clause containing only a universal variable should yield the empty clause rather than forcing the universal.

Semantics. A SAT model $\mathcal{M}_s$ of a CNF formula F is a truth assignment π to the variables of F that satisfies every clause in F. We denote the value of a variable v in π by $\pi(v)$. We have developed an obvious extension of SAT models for QBF[95].

Definition 2 *QBF Model $\mathcal{M}_q$ as a tree*

*A QBF model (**Q-model**) $\mathcal{M}_q$ of a quantified formula $\vec{Q}.F$ is a **tree** of truth assignments in which the root is the empty truth assignment, and every node n assigns a truth value to a variable of F not yet assigned by one of n's ancestors. The tree $\mathcal{M}_q$ is subject to the following conditions:*

1. *For every node n in $\mathcal{M}_q$, n has a sibling if and only if it assigns a truth value to a universal variable x. In this case it has exactly one sibling that assigns the opposite truth value to x. Nodes assigning existentials have no siblings.*

2. *Every path π in $\mathcal{M}_q$ (π is the sequence of truth assignments made from the root to a leaf of $\mathcal{M}_q$) must assign the variables in an order that respects $<_q$. That is, if n assigns x and one of n's ancestors assigns y then we must have that $y \leq_q x$.*

3. *Every path π in $\mathcal{M}_q$ must be a SAT model of F. That is π must satisfy the body of $\vec{Q}.F$.*

Thus a Q-model has a path for every possible setting of the universal variables of $\vec{Q}$, and each of these paths is a SAT model of F. We say that a QBF $\vec{Q}.F$ is QSAT iff it has a Q-model. The QBF problem is to determine whether or not $\vec{Q}.F$ is QSAT.

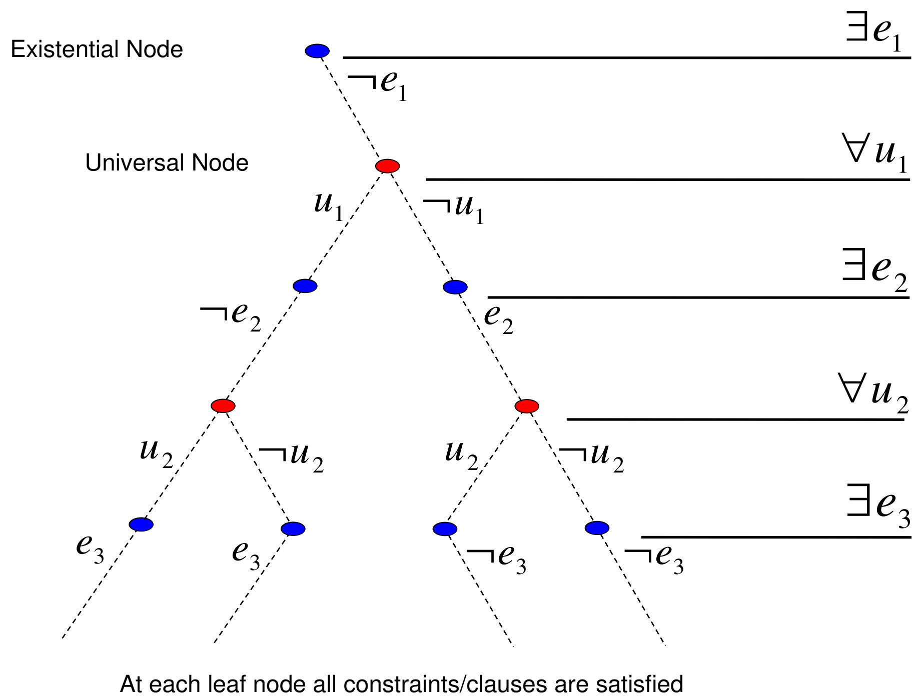


Figure 2.1: An QBF Model $\mathcal{M}_q$ of the formula $\exists e_1 \forall u_1 \exists e_2 \forall u_2 \exists e_3.F$ represented as a tree.

An example for a Q-model represented as a tree given the underlying QBF $\exists e_1 \forall u_1 \exists e_2 \forall u_2 \exists e_3.(e_1 \vee u_1 \vee \neg e_2 \vee \neg e_3) \wedge (e_2 \vee u_2 \vee e_3)$ is shown in Figure 2.1. In the displayed tree each existential node has exactly one child node and each universal node has exactly two child nodes as

required by Definition 2. Furthermore, all constraints are satisfied at each leaf node of the tree. The quantifier prefix shown on the right-hand side of the figure emphasizes the ordering constraint the tree has to satisfy.

An alternate, and more standard way of defining QSAT is the following recursive definition (e.g.,[95]):

Definition 3 *Recursive Definition of QSAT*

1. *F is the empty set of clauses then $\vec{Q}.F$ is* true.
2. *F contains an empty clause then $\vec{Q}.F$ is* false.
3. *$\forall v \vec{Q}.F$ is* true *iff both $\vec{Q}.F|_v$ and $\vec{Q}.F|_{\neg v}$ are true.*
4. *$\exists v \vec{Q}.F$ is* true *iff at least one of $\vec{Q}.F|_v$ and $\vec{Q}.F|_{\neg v}$ is true.*

Note that since $\vec{Q}$ contains all of the variables of F, cases 3 and 4 must eventually reduce to instances of cases 1 and 2.

Proposition 1 *$\vec{Q}.F$ has a Q-Model iff it evaluates to true under the preceding recursive definition.*

Proof: The proof is by induction on the number of variables in the prefix. If $\vec{Q}$ has no variables then F can only be either a collection of empty clauses or it must be the empty set of clauses (all variables of F must appear in $\vec{Q}$). In the first case $\vec{Q}.F$ can have no Q-model since no path can satisfy F. In the second case a tree consisting of only an unlabelled root node is a Q-model for $\vec{Q}$. Assume that if $\vec{Q}.F$ has n variables then $\vec{Q}.F$ has a Q-model if and only if $\vec{Q}.F$ evaluates to true. Consider a formula $\exists v \vec{Q}.F$ that has $n+1$ variables.

"$\Leftarrow$": Assume that $\exists v \vec{Q}.F$ has a Q-model. Then by the definition of a Q-model, either $\vec{Q}.F|_v$ or $\vec{Q}.F|_{\neg v}$ has a Q-model, which consists of the left or right subtree of the Q-model of $\exists v \vec{Q}.F$. Therefore, by the induction hypothesis, either $\vec{Q}.F|v$ or $\vec{Q}.F|_{\neg v}$ evaluates to true. Therefore, by #4, $\exists v \vec{Q}.F$ evaluates to true.

"$\Rightarrow$": Assume that $\exists v\vec{Q}.F$ evaluates to true. Then by #4, either $\vec{Q}.F|_v$ or $\vec{Q}.F|_{\neg v}$ evaluates to true. Therefore, by the induction hypothesis, either $\vec{Q}.F|_v$ or $\vec{Q}.F|_{\neg v}$ has a Q-model $\mathcal{M}_q$. Let $\mathcal{M}'_q$ be the Q-model consisting of a root node whose single child is the root node of $\mathcal{M}_q$. Further we relabel the root node of $\mathcal{M}'_q$ with $v = true$ or $v = false$ depending on whether $\mathcal{M}_q$ is a Q-model of $\vec{Q}.F|_v$ or $\vec{Q}.F|_{\neg v}$. Clearly, $\mathcal{M}'_q$ is a Q-model for $\exists v\vec{Q}.F$.

Therefore, $\exists vQ.F$ has a Q-model if and only if it evaluates to true. The case with $\forall vQ.F$ is similar. Therefore, by induction, a formula $\vec{Q}.F$ has a Q-model if and only if it evaluates to true. ■

The advantage of our "tree-of-models" Definition 2 is that it makes two key observations more apparent. These observations can be used to prove the correctness of various techniques utilized in QBF solvers.

Lemma 1 *If F' is a formula with the same variables of F that has the same satisfying assignments (SAT models) as F then $\vec{Q}.F$ will have the same satisfying models (Q-models) as $\vec{Q}.F'$.*

Proof: $\mathcal{M}_q$ is a Q-model of $\vec{Q}.F$ iff each path in $\mathcal{M}_q$ is a SAT model of F iff each path is a SAT model of F' iff $\mathcal{M}_q$ is a Q-model of $\vec{Q}.F'$. ■

This observation allows us to transform F with any model preserving SAT transformation. Note that the transformation must be model preserving, i.e., it must preserve all SAT models of F. Simply preserving whether or not F is satisfiable is not sufficient.

A Q-model preserving (but not SAT model preserving) transformation that can be performed on $\vec{Q}.F$ is universal reduction which we defined earlier. Before proving this result we first make the following observation:

Lemma 2 *Let $\mathcal{M}_q$ be a Q-model for $\vec{Q}.F$. In addition, let F' denote the formula F with either (a) clauses removed or (b) literals from the variables of F added to the clauses. Then $\mathcal{M}_q$ is a Q-model for $\vec{Q}.F'$.*

Proof: For Case (a) we observe that along any path in $\mathcal{M}_q$ all clauses in F are satisfied. Since F' has less clauses than F, $\mathcal{M}_q$ is clearly a Q-model for $\vec{Q}.F'$. In Case (b) we note again that along any path in $\mathcal{M}_q$ all clauses are satisfied. All altered clauses in F' are less constraining than the clauses in F since these clauses in F' express weaker disjunctive conditions. Consequently, $\mathcal{M}_q$ is a Q-model for $\vec{Q}.F'$. ■

Now we show that universal reduction does not alter the set of Q-models.

Lemma 3 *Universal reduction preserves the set of Q-models.*

Proof: Say that $v \in c$ is a tailing universal, then along any path π in any Q-Model $\mathcal{M}_q$ of $\vec{Q}.F$, c must be satisfied by π prior to v being assigned a value. Say not, then since v is universal, the prefix of π that leads to the assignment of v must also be the prefix of another path π' that sets v to false: but then π' will falsify c because at this point c is a unit clause containing only the universal variable v. Therefore $\mathcal{M}_q$ cannot be a Q-model of $\vec{Q}.F$. Hence every path π satisfies the universal reduction of c (and all other clauses in F), and thus $\mathcal{M}_q$ is also Q-model of $\vec{Q}.F'$ where F' is F with the tailing universal v removed from c. This process can be repeated to remove all tailing universals from all clauses of F.
On the other hand, it is the case that if $\vec{Q}.F'$ has a Q-model, then $\vec{Q}.F$ is true by the same Q-model as well. This follows directly from Lemma 2. ■

Before we conclude this section on the semantics of QBF we also show that the following additional properties hold:

Lemma 4 *Let e be an existential variable and $\mathcal{M}_q$ be a Q-model of $\vec{Q}.F|_e$. Then $\mathcal{M}_q$ can be extended to a Q-model $\mathcal{M}'_q$ of $\vec{Q}.F$.*

Proof: Since $\mathcal{M}_q$ is a Q-model of $\vec{Q}.F|_e$ each complete path in this model is a SAT solution of $F|_e$. Consequently, each clause c of F containing $\neg e$ is satisfied by some other variable in c along each path. All clauses of F not contained in or subsumed by $F|_e$ must contain e. We can extend $\mathcal{M}_q$ by adding the existential node of e at any position where it satisfies the quantifier ordering. The resulting Q-model $\mathcal{M}'_q$ thus satisfies every clause

of F along each path and is a Q-model for $\vec{Q}.F$. ∎

Lemma 5 *Let u be an universal variable and $\mathcal{M}_q$ be a Q-model of $\vec{Q}.F$. Then $\mathcal{M}_q$ contains a Q-model of $\vec{Q}.F|_u$.*

Proof: By Definition 2 the Q-model $\mathcal{M}_q$ covers both settings of the universal u. Consequently, $\vec{Q}.F|_u$ simplifies the original problem by fixing u to one particular truth value. At every appearance of a pair of siblings assigning u in the Q-tree of $\mathcal{M}_q$ we simply remove the sibling labeled by $\neg u$ and the subtree below it and replace the sibling labeled u by its children. The resulting tree is a Q-model of $\vec{Q}.F|_u$. ∎

Corollary 1 *If $\vec{Q}.F$ is true then so is $\vec{Q}.F|_u$ when u is a universal literal. If $\vec{Q}.F|_e$ is true for any existential literal e then so is $\vec{Q}.F$.*

The two semantic Definitions 2 and 3 also reflect the mode of operation utilized by a search based algorithm to solve QBF. Definition 3 captures the two main properties of QBF that must be accommodated by the search in a straight forward fashion. First, the search must solve both settings of every universal variable, and second the variable ordering followed during search must respect the ordering imposed by quantifier nesting. Conceptually, a search based algorithm traverses the search space trying to uncover a Q-model. The following section describes such an algorithm in more detail.

2.3 Search-Based QBF Solver

Most current QBF solvers, e.g., QuBE [46], Semprop [66], Quaffle [111], SSolve [39] are adaptations of the classic DPLL backtracking search algorithm originally developed for solving SAT [30]. In addition, there also exist approaches based on the original Davis-Putnam procedure [31] and Skolemization [100] that will be discussed in the subsequent section.

There are two main properties of QBF that must be accommodated by the search. First, the search must solve both settings of every universal variable, and second the

variable ordering followed during search must respect the ordering imposed by quantifier nesting. Both of these properties make QBF solving harder than SAT solving. The first property is intrinsic to QBF, and must be accommodated in some fashion by any QBF solver. The second property is, however, somewhat more tractable, and various attempts have been made to avoid the variable ordering constraint. In a subsequent chapter we will discuss a search-based solver that tries to decrease the impact of the strict variable ordering. In addition, the non-search based methods, discussed in the next section, can also be viewed as being an approach for decreasing the impact of the restrictive variable ordering of QBF.

In the remainder of this section we introduce a general framework for a search-based QBF solver. Then we will discuss the notion of solution learning. In addition, we review several other extensions to this framework.

2.3.1 DPLL for QBF

In this section we introduce a framework based on DPLL [30] that is close to current search-based QBF solvers. First, we start off with a basic DPLL based algorithm where the similarity to the recursive definition 3 is apparent. Furthermore, we describe how it deals with the two previously mentioned main requirements of QBF search. Then we move on to a DPLL framework that incorporates clause as well as solution learning.

DPLL works on the principle of assigning variables (reducing the theory by a variable setting), simplifying the formula to account for that assignment and then recursively solving the simplified formula.

The algorithm utilized in modern SAT solvers (e.g., [73], [35]) can be adapted to solve QBF. A simple recursive version of this algorithm called **QBF-DPLL-Basic** is shown in Algorithm 1.

1: ⟨bool *Result*⟩ **QBF-DPLL-Basic** $(\vec{Q}.F)$

{Return true if the theory is empty}

2: **if** *all clauses of* F *are satisfied* **then**

3: **return** *true*

{Return false if the theory contains a falsified clause}

4: **if** F *contains a falsified clause* **then**

5: **return** *false*

6: Pick v from the first quantifier block and let $\ell = v$ or $\neg v$

7: $Result$ = **QBF-DPLL-Basic** $QProp\big(\vec{Q}.F|_{\ell}\big)$

{If v is existential only branch on the other truth value if the first one failed}

8: **if** v is existential and $Result == false$ **then**

9: $Result$ = **QBF-DPLL-Basic** $QProp\big(\vec{Q}.F|_{\neg\ell}\big)$

{If v is universal only branch on the other truth value if the first one succeeded}

10: **if** v is universal and $Result == true$ **then**

11: $Result$ = **QBF-DPLL-Basic** $QProp\big(\vec{Q}.F|_{\neg\ell}\big)$

12: **return** *Result*

Algorithm 1: Basic DPLL for QBF

As input the algorithm receives the problem instance $\vec{Q}.F$ and it returns the corresponding evaluated truth value. The lines 2-3 and 4-5 reflect the two base cases from the recursive definition 3.

At line 6 we see that **QBF-DPLL-Basic** must always branch on a variable from the outermost quantifier block. As mentioned before this imposes a constraint on the possible variable orderings the search can use as compared to SAT. Dynamic variable ordering is , however, allowed within a quantifier blocks. And this can have a significant impact on performance [111]. However, research on variable ordering for QBF is not as advanced as in SAT (e.g., [54], [67]).

After selecting a literal l the algorithm is called recursively on the theory reduced by ℓ

and depending on the quantification type of v the algorithm continues recursing or returns the determined result. In the existential case the algorithm only tries the opposite truth value of ℓ if the recursion failed. Otherwise it returns with success from this recursive call. This captures #3 in the Definition 3 which requires that if v is existential only one setting has to succeed. In contrast, if v is universal the algorithm only recurses on the opposite truth value if the first setting returns with success. Again, the correspondence with #4 of Definition 3 is apparent.

Theorem 1 ***QBF-DPLL-Basic*** *as shown in Algorithm 1 is sound and complete.*

Proof:

Soundness:

The proof is by induction on the number of variables in the prefix $\vec{Q}$. The first base case is when F is the empty theory, where **QBF-DPLL-Basic** soundly returns with *true*. The second base case is when F contains the empty clause where **QBF-DPLL-Basic** soundly returns with *false*.

Assume that if $\vec{Q}.F$ has n variables then **QBF-DPLL-Basic** returns from the root node with the correct answer. Now consider a formula $\exists(\forall)v\vec{Q}.F$ that has $n+1$ variables. By the induction hypothesis the value returned by **QBF-DPLL-Basic** on $\vec{Q}.F$ with n variables is sound. If the returned truth value is true and v is existential, **QBF-DPLL-Basic** returns *true* and is sound. If the returned value is true and v is universal, **QBF-DPLL-Basic** recurses on the opposite truth value of v. If this recursive call returns true, both settings of the universal are verified and **QBF-DPLL-Basic** correctly return *true*. Else, one branch of the universal failed and accordingly **QBF-DPLL-Basic** returns *false*.

If the returned value on $\vec{Q}.F$ with n variables is false and v is existential, **QBF-DPLL-Basic** recurses on the opposite truth value of v. If this recursive call returns true, one setting of the existential succeeds and **QBF-DPLL-Basic** correctly returns *true*. Else, both branches of the existential failed and accordingly **QBF-DPLL-Basic**

returns *false*. If the returned value on $\vec{Q}.F$ with n variables is false and v is universal, one universal setting failed, and **QBF-DPLL-Basic** soundly returns *false*.

Consequently, both $\exists v\vec{Q}.F$ and $\forall v\vec{Q}.F$ evaluate to the correct answer. Therefore, by induction, **QBF-DPLL-Basic** applied on a formula $\vec{Q}.F$ is sound.

Completeness:

That **QBF-DPLL-Basic** is also complete follows from the fact that no recursive call has exactly the same prefix of assignments as another call (after a failure (success) the prefix has a different value for one of the existential (universal) variables). Since there are only a finite number of sets of assignments, there can only be a finite number of recursive calls, and the root **QBF-DPLL**invocation must eventually return. By the previous argument this answer must be correct. ■

2.3.2 Learning

Modern backtracking QBF solvers employ two non-chronological backtracking schemes: conflict analysis and solution analysis [66, 47, 33, 109]. While conflict analysis is very similar to the conflict analysis applied in SAT solvers, solution analysis is a technique new to QBF. Therefore, we will put an emphasis on solution analysis here.

First we introduce an extension of Algorithm 1 that makes the algorithm close to current search-based QBF solvers. The extension adds conflict as well as solution learning. The resulting **QBF-DPLL** algorithm is displayed in Algorithm 2. In addition to the boolean truth value the algorithm also returns a backtracking level and set of literals called a cube (which will be discussed later on). This extra information is used to enable non-chronological backtracking.

In comparison to Algorithm 1, with learning the input formula $\vec{Q}.F$ must be globally accessible. To do learning, the algorithm must examine how the original formula $\vec{Q}.F$ has been altered as a result of the assumptions made during search. Furthermore, in Algorithm 2 the parameter T is the trail of literals assumed by previous instantiations

(variable assignments). T not only consists of decision literals but also stores the literals forced by unit propagation. Algorithm 2 is invoked on a given input formula $\vec{Q}.F$ with the empty set of literals T and with *Level* assigned to 1.

Conflict Analysis

Conflict analysis is a standard SAT technique that involves learning new clauses via a resolution process. Here, we will not discuss learning based on conflicts in detail since it is very similar to learning in the SAT case (except for universal reduction). We refer the reader to for instance [73] for the propositional case and to [66, 47, 33, 109] for QBFs. Instead we go through the algorithm and explain the context in which conflict-based learning is applied.

With learning the employed propagation in Algorithm 2 requires the detection of contradictions and the performance of clause learning. Consequently, *QProp* returns a newly inferred clause if a conflict is discovered (line 7). A new clause is determined via a process of resolution and universal reduction and is added to the original formula $\vec{Q}.F$ (line 10).

QBF-DPLLwill then backtrack to the **asserting** level of the conflict (line 11), which is the level where all but one of the literals in the conflict clause have been falsified. This is the level where the conflict clause is made unit.

After returning from all levels deeper than *BTLevel* (line 15-16 or 21-22), the solver arrives at line 7, where we now have that the new conflict clause is unit and forces ℓ. The repeat-until loop (line 6) will now invoke *QProp* again, which will make additional inferences due to the new conflict clause in F. Then a new branching decision is made (i.e., a new variable is chosen (line 12)).

Notice that the solver does not actually undo the original decision made at this level. Rather it simply augments the reduction of $\vec{Q}.F$ by the new unit implicant ℓ (line 7). Thus the solver might return to this level on failure a number of times: each time it

```
1: ⟨bool Result, cube c, int BTLevel⟩ QBF-DPLL(Level, T)
2: if T [satisfies all clauses of F/triggers stored cube] then
3:    c = [computed new cube by solution analysis/triggered cube]
4:    u = deepest universal in c; BTLevel = Level of u was branched on
5:    return ⟨SUCCEED, c, BTLevel⟩{Note if c is empty then BTLevel = 0}
6: repeat
7:    conflict = QProp(Q⃗.F|_T)  {Must detect contradictions+perform clause learning}
8:    if (conflict != NULL) then
9:       BTLevel = Level where conflict clause was made unit
10:      Q⃗.F = Q⃗.F ∪ conflict {Learnt clauses modify original formula F)}
11:      return ⟨FAIL, -, BTLevel⟩
12:   Pick minimal (i.e., outermost) v not yet assigned by T and let ℓ = v or ¬v
13:   T' = T ∪ ℓ {Add ℓ to trail T}
14:   ⟨Result, c, BTLevel⟩ = QBF-DPLL(Level + 1, T′)
15:   if (BTLevel < Level) then
16:      return ⟨Result, c, BTLevel⟩
17:   if (Result == SUCCEED) then
18:      cube[l] = c {Store cube}
19:      T' = T ∪ ¬ℓ {Add ¬ℓ to trail T and solve recursively}
20:      ⟨Result, c, BTLevel⟩ = QBF-DPLL(Level + 1, T′)
21:      if (BTLevel < Level) then
22:         return ⟨Result, c, BTLevel⟩
23:      if (Result == SUCCEED) then
24:         newcube = cube[l] ∪ c \ {¬l, l} {Tailing existentials are removed}
25:         u = deepest universal in newcube; BTLevel = Level of u was branched on
26:         return ⟨SUCCEED, c, BTLevel⟩ {Note if c is empty then BTLevel = 0}
27: until FALSE {only exit from this loop via a return to a higher level}
```

Algorithm 2: DPLL for QBF with Learning

discovers that another literal is implied at this level. Eventually, the recursive call at line 20 returns success at this level and it will return to a higher level. (Each failure return sets another variable, so a failure return to this level at line 7 can only occur a finite number of times.)

Solution Analysis

Success returns occur as a consequence of solution analysis (line 2-5). Solution analysis (e.g., [111], [33], [66]) is a technique particular to QBF. It starts when a complete satisfying assignment π of F is found a subset of π is identified that satisfies every clause in F.

Although cubes were previously discussed in [109, 33, 66] we now give a new formalization of this concept. Our definition tries to capture both the formal properties of a cube in a way that is more closely related to their usage in a search-based solver.

First we define the relation $\leq_u$ for two consistent sets of literals c and c'. A set of literals c is consistent if $\forall l.l \in c \implies \neg l \notin c$.

We say that $c' \leq_u c$ iff $\forall l \in c' \setminus c$ we have $qb(l) \leq umax(c)$ where $umax(c)$ is the deepest quantifier block containing a universal literal of c.

Definition 4 *Cube*

A consistent set of of literals c is a cube for the QBF formula $\vec{Q}.F$ iff for all consistent sets of literals c' such that $c' \leq_u c$ we have that $\vec{Q}.F|_{c'}$ is QSAT.

In other words a cube is defined by the property that a QBF $\vec{Q}.F$ reduced by it is QSAT regardless of the setting of the literals not mentioned in the cube that are upstream of the deepest universal in the cube. According to this definition the empty cube c ,i.e., the empty set of literals is a cube iff the original formula $\vec{Q}.F$ is QSAT. Note that the quantifier level of the empty set of literals is lower than any quantifier level of any variable in $\vec{Q}.F$ (as defined at the beginning in Section 2.2).

As we will discuss in Chapter 5 this definition of cubes is not sufficient for solution

learning in a context where the theory F is dynamically partitioned. However, unlike previous definitions this definition can be more easily extended to the context of dynamic partitioning.

The following theorem justifies standard cube learning in a search-based QBF solver. Each item of this theorem is explained within a subsequent example.

Theorem 2

1. *If c is a set of literals that satisfies every clause of F, then c is a cube.*

2. *If c is a cube and e is existential and maximal in c, then $c' = c \setminus \{e\}$ is a cube*

3. *If c_1 and c_2 are cubes such that (1) there is a unique literal u such that $u \in c_1$ and $\neg u \in c_2$, (2) this clashing literal is universal, and (3) u is maximal in c_1 and $\neg u$ is maximal in c_2, then $c_1 \cup c_2 \setminus \{u, \neg u\}$ is a cube.*

Proof:

For item 1: Let c denote the set of literals that satisfies every clause in the theory. Let $c' \leq_u c$. We must show that $\vec{Q}.F|_{c'}$ is QSAT. Since $\vec{Q}.F|_c$ is the empty theory, $\vec{Q}.F|_{c'}$ is also the empty theory and by Definition 3 it is QSAT.

For item 2: Let c denote a cube, where the existential e is maximal in c. We want to show that $c \setminus \{e\}$ is a cube as well. Let $c' \leq_u c \setminus \{e\}$. We must show that $\vec{Q}.F|_{c'}$ is QSAT. Since e is maximal in c we cannot have $\neg e$ in c' (since $c' \leq_u c \setminus \{e\}$). Thus $c' \cup \{e\}$ is consistent. Furthermore, $c' \cup e \leq_u c$, and since c is a cube we have that $\vec{Q}.F|c' \cup e$ is QSAT. Then by Lemma 4 we must also have that $\vec{Q}.F|c'$ is QSAT.

For item 3: Let c_1 and c_2 denote cubes that satisfy the conditions of item 3. Then let c_3 denote the cube arising from the resolution of $c1$ and $c2$ with u_r as the corresponding clashing universal, $u_r \in c_1$, $\neg u_r \in c_2$, and $c_3 = c_1 \cup c_2 \setminus \{u_r, \neg u_r\}$). Let $c' \leq_u c_3$. We must show that $\vec{Q}|_{c'}$ is QSAT. We do this by showing that an even stronger formula than $\vec{Q}.F|_{c'}$ is QSAT, which will imply that $\vec{Q}.F|_{c'}$ must also be QSAT.

In particular, let E be the set of existentials upstream of u_r that do not appear in c'. Let $\vec{e}$ be an arbitrary consistent set of literals from E, i.e., $\vec{e}$ is an fixed but arbitrary valuation of the existentials in E. Now consider the formula $\vec{Q}.F|_{c' \cup \vec{e}}$—this is a stronger formula than $\vec{Q}.F|_{c'}$ by Lemma 4. Then let $\vec{u}$ be the set of universal literals appearing in c' that do not appear in c_3. Consider the formula $\vec{Q}.F|_{c' \cup \vec{e} - \vec{u}}$; by Lemma 5 this is a stronger formula than $\vec{Q}.F|_{c' \cup \vec{e}}$, and thus a stronger formula than $\vec{Q}.F|_{c'}$. We now proceed to show that $\vec{Q}.F|_{c' \cup \vec{e} \setminus \vec{u}}$ is QSAT.

The set $c' \cup \vec{e} \setminus \vec{u}$ instantiates every existential upstream of u_r. Thus $Q.F|_{c' \cup \vec{e} \setminus \vec{u}}$ consists of an initial universal quantifier block containing u_r. Since the variables in each quantifier block can be ordered in an arbitrary fashion without affecting the formula, $Q.F|_{c' \cup \vec{e} \setminus \vec{u}}$ can be written as the QBF $\forall u_r(\Phi)$ where Φ is itself a QBF formula. By Definition 3 $\forall u_r(\Phi)$ is QSAT iff both $\Phi|_{u_r}$ and $\Phi|_{\neg u_r}$ are QSAT. $\Phi|_{u_r}$ is equivalent to $\vec{Q}.F|_{c' \cup \vec{e} \setminus \{\vec{u} \cup u_r\}}$ while $\Phi|_{\neg u_r}$ is equivalent to $\vec{Q}.F|_{c' \cup \vec{e} \setminus \vec{u} \cup \neg u_r}$. We have that $c' \cup \vec{e} \setminus \{\vec{u} \cup u_r\} \leq_u c_1$ and $\vec{e} \setminus \{\vec{u} \cup \neg u_r\} \leq_{u_r} c_2$. Since both c_1 and c_2 are cubes both of these formulas must be QSAT, and hence $\vec{Q}.F|_{c' \cup \vec{e} \setminus \{\vec{u} \cup u_r\}}$ is QSAT and hence $\vec{Q}.F|_{c'}$ is QSAT as required. ∎

Theorem 2 captures the essential processes that occur within solution learning in a search-based QBF solver as we illustrate in the following example. As an example assume that we have the following QBF:

$$\exists e_1 \forall u_1 \exists e_2 \forall u_2 \exists e_3 (\neg e_1 \vee e_3) \wedge (\neg e_2 \vee u_2 \vee e_3) \wedge (\neg e_2 \vee u_1) \wedge (\neg e_3 \vee \neg u_1)$$

In addition, we assume that the solver assigned the variables in such a way that it achieved a complete satisfying assignment π of F. The assignment π sets the variables to the following values:

$$\pi(e_1) = false, \pi(u_1) = false, \pi(e_2) = false, \pi(u_2) = true, \pi(e_3) = false$$

$\vec{Q}.F|_\pi$ is the empty theory and consequently π is obviously a cube (by Item 1 of Theorem 2). However, given π there exist many different valid cubes that can be extracted from it. The main reason for this is the fact that most of the clauses are satisfied by

more than one literal. Solvers compute and store one particular cube from each satisfying assignment, cubes are heuristically generated (for efficiency) by greedily selecting a subset of the satisfying assignment sufficient to satisfy all of the clauses. [109].

In general, solvers try to minimize both the number of total literals in the cube and the number of universals (e.g., [109] [95]). The smaller the number of universals in a cube, the fewer universal settings that remain to be verified. Hence, the right subset of the satisfying assignment can have a critical effect on the efficiency of search.

One method that aims at maximizing both properties simultaneously is to maximize the number of *tailing* existentials. The motivation for this method emerges from the observation that tailing existentials satisfy more clauses in practice (e.g., [95], [90]) and can be disregarded within a cube (by Item 2 of Theorem 2).

In the given example, a possible set of literals that satisfy F is: $(u_1 \wedge e_2 \wedge u_2 \wedge \neg e_3)$. After the removal of tailing existentials we obtain the cube $C_1 = (u_1 \wedge e_2 \wedge u_2)$.

After determining a cube, c, and removing all tailing existentials the solver can then backtrack to the deepest universal in the cube, skipping other universals and existentials not mentioned in the cube. At the point the deepest universal $l \in c$ has been instantiated the solver is trying to show that $\vec{Q}.F|_{T \cup l}$ is QSAT (line 4 or 20). We have that $T \cup l \leq_u c$, thus we have that $\vec{Q}.F|_{T \cup l}$ is QSAT from the fact that the determined set of literals is provably a cube. Hence no further search is needed to verify the subformula $\vec{Q}.F|_{T \cup l}$, i.e, it is legitimate for the solver to backtrack to this point. Note that due to removal of tailing existentials cubes always cause a backtrack to a universal variable. Thus a success return can only occur if v is universal.

As shown in Theorem 2 Item 3 a cube containing one setting of a universal can be combined with another cube containing the other setting to obtain a new cube in a process akin to the resolution of clauses (line 24). In particular, if the deepest universal in the cube has already had its other value solved, the solver will combine these two cubes and remove the deepest universal. The apparent symmetry between clause resolution

and cube (term) resolution was discussed in [109, 33].

Following our example, assume that the solver applied solution backtracking on the cube, i.e. $C_1 = (u_1 \wedge e_2 \wedge u_2)$. The deepest universal in C_1 is u_2. Since both settings of a universal have to be verified the solver backtracks to u_2 (undoing u_2 as well) and forces the negation of u_2 ($\neg u_2$). Then it attempts to solve $F|_{\neg u_1, e_2, \neg u_2}$. Suppose that this search find the following solution π':

$$\pi'(e_1) = false, \pi'(u_1) = false, \pi'(e_2) = false, \pi'(u_2) = false, \pi'(e_3) = true$$

Furthermore assume that the cube $C_2 = \neg u_1 \wedge e_2 \wedge \neg u_2$ is extracted from π'. Now both sides of the universal u_2 are solved. The two cubes C_1 and C_2 are resolved on u_2 to produce the new cube $C_3 = \neg u_1$ where the tailing existential e_2 has been removed (using Item 2 and 3 of Theorem 2). The cube C_3 captures the fact that $F|_{\neg u_1}$ is QSAT.

Returning to Algorithm 2, we see that on success the solver always backtracks to a universal variable whose other side is not yet solved. Simply, because it backtracks to the level that assigned the deepest universal in the active cube (line 4-5). After backtracking to the corresponding level line 19 flips the truth value of the universal and the algorithm recurses with this setting (line 20).

The recursive call on line 20 returns to the current level after attempting the second universal value. This return might also be through failure which again means that there is a new conflict clause in F made unit by T (the conflict clause does not depend on the setting of the flipped universal $\neg l$). Again the repeat-until loop (line 6-27) will reinvoke *QProp* to make new inferences, followed by a a new branch decision.

One additional aspect of solution and conflict analysis is that the new clauses and cubes can be stored (learned) (see e.g., line 10 and 18), reused along other paths in the search (see e.g., line 2), and combined together to produce more powerful clauses and cubes (see e.g., line 24). Cube and clause learning is essential in achieving state of the art performance in QBF solving. Note that Theorem 2 also verifies the soundness of triggering cubes during search.

Besides storing newly inferred clauses and cubes Algorithm 2 also takes into account learnt clauses and cubes. In particular, the algorithm also succeeds if any learnt cube becomes true (line 2). Since the original theory is altered by learnt clauses (line 10) the algorithm also fails if any learned clause was falsified (line 7). Again cube and clause learning is also developed in more detail in, e.g., [44, 47, 66, 33, 110, 111].

With the enhancements of cube and clause learning **QBF-DPLL**as specified in Algorithm 2 is quite close to current QBF solvers like Quaffle [111] and QuBE [46].

Soundness and Completeness

Theorem 3 ***QBF-DPLL****as shown in Algorithm 2 is sound and complete.*

Proof:

Soundness: The algorithm exits with fail only when $BTLevel = 0$ (see line 11: **return***(FAIL,-,BTLEVEL)*. This only occurs if a learnt clause is empty—unit clauses become unit at level 1, while empty clauses become unit at level 0. Success returns to level 0 only occur if an empty cube is learnt (line 5 and 26).

In both cases the algorithm is sound as learnt clauses and cubes are correctly inferred logical consequences of the input theory. (The correctness of the learnt cubes was demonstrated in Theorem 2, the correctness of the learnt clauses is obvious from the fact that these clauses are inferred via a process of resolution).

Completeness: In the algorithm no recursive call has the same trail T, and there are only a finite number of different trails since there exists only a finite number of unique variable assignments. Consequently, the root **QBF-DPLL**invocation must eventually return with the correct answer as well. ∎

2.3.3 Space Complexity

Finally we point out that **QBF-DPLL**without learning requires only linear space (in the number of variables n), and only quadratic space (in n) when it utilizes non-chronological

conflict and solution backtracking. In the worst case with conflict and solution backtracking we must store a clause (cube) for every failed existential value (successful universal value) along the current path being explored. These clauses (cubes) have maximum size equal to the number variables n, and the current path can contain at most n literals.

However, when clause and/or cube learning is employed (i.e., the cubes and clauses are stored) the algorithm can consume as much space as can be provided. Nevertheless, learning clauses and cubes does not affect the soundness or completeness of the algorithm, it only helps to improve performance. In particular, we can adopt any strategy for deleting these learned clauses and cubes when we run out of space, without affecting the correctness of the algorithm. In this sense **QBF-DPLL**as shown in Algorithm 2, like most current SAT solvers, is an "any-space algorithm," it can utilize any space provided above and beyond its basic polynomial space requirements, but it can also work under any fixed space bound (above its basic requirements).

2.3.4 Extensions of the Basic Framework

In order to increase the efficiency of the basic backtracking framework a number of extensions have been purposed. The changes range from simple techniques that can be easily embedded in the basic framework to more fundamental changes like a different representation of clauses requiring more adaptions of the general framework that was presented here. In this section, we review some techniques that have been introduced in an attempt to improve on the basic search algorithm.

A first simple extension is known as *trivial truth* [25]. From the quantified formula $\vec{Q}.F$ all universal quantified variables are removed from both the quantifier prefix $\vec{Q}$ and the formula F. Then the remaining formula F_r is treated as a SAT instance since there are only existential quantified variables left. If F_r is satisfiable then $\vec{Q}.F$ is obviously true too: The satisfying existential assignment satisfies F under every setting of the universal variables. However, if F_r is unsatisfiable nothing about the truth value of $\vec{Q}.F$ is revealed.

Hence, empirically the employment of trivial truth turns out to be ineffective [25, 66].

Analogously to the notion of trivial truth there also exists the notion of *trivial falsity* [25]. Now no variables are removed. Instead all universal variables are treated as if they are existential again giving rise to a SAT instance. If this SAT instance turns out to be unsatisfiable then we know that also $\vec{Q}.F$ is unsatisfiable. Similarly, we know that $\vec{Q}.F$ is trivially false if it contains a clause that contains universal quantified variables only. It has been shown that trivial falsity can be beneficial on several instances [25]. However, it remains an open question as to how often trivial falsity should be employed during the search [25].

The idea of the *inversion of quantifiers* is based on the general observation that a combination of universal variables can force upstream existential variables [86]. In a QBF $qb_1^{\exists}qb_2^{\forall}...qb_n^{Q}.F$ a combination of universal quantified variables is set to to arbitrary values to detect if any of the existentials in the first quantifier block $qb_1^{\exists}$ become forced. The set of forced existentials contained in $qb_1^{\exists}$ under any combination of universal variable values can be set accordingly since they must have the determined value in any Q-Model. In this way the idea of inverting quantifiers reduces the number of possible combinations of values for the existential variables in the outermost existential quantifier block—if the outermost block is existential. Benchmark results show that this technique can be successful on instances that start with an existential quantifier block [86, 38].

Monotone literals are literals that appear in only one polarity in the formula F. For instance, in $F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$ the literal x_2 is the only monotone literal. In a SAT solver that supports the detection of monotone literals these can be set to true to reduce the size of the theory. Furthermore, this can be done at any stage of the search. For example, a literal might become monotone as a consequence of previously made assumptions. No conflicts can be caused by a truthified monotone literal. However, in most SAT solvers monotone literals are not detected since there exists no empirical evidence that it is beneficial [48].

In contrast, within QBF solvers the detection of monotone literals seems to be beneficial despite its overhead [48, 25]. This can be explained by the following reasons. In the context of QBF the search is more extensive than with SAT since all universal settings have to be verified. Hence the extra work to simplify the theory by purity can be amortized over a larger search space.

Furthermore, monotone universals are treated differently than monotone existentials. In QBF monotone existentials are handled like monotone literals in the SAT case. However, monotone universals are set to the "hard" decision. Hence, if u is monotone the algorithm would branch on $\neg u$ instead of u. In the example from before x_2 would be set to false if x_2 is a universal.

This is based on the fact that the solver has to verify both settings of a universal. From Lemma 2 we have that if the subtheory generated by the hard setting is QSAT, the subtheory under the easy setting must also be. Thus the solver needs only to solve the hard setting.

A different *representation* of clauses was presented in [43]. In this work an extension of binary decision diagrams (BDD) is used to represent the clauses of a QBF instance. This extension is called zero suppressed BDDs (ZDDs). The authors of [43] claim that ZDDs are suitable for modeling a CNF formula as they can present a set of subsets in a efficient way [71]. This representation is used with an adapted version of DPLL that has splitting and propagation rules specifically designed for ZDDs. Due to the compact representation the approach is able to store solved subproblems (similar to cubes) with a low overhead. In addition, the canonical representation of these subproblems allows efficient equality comparison. The authors of [43] utilized their approach to build a solver they called ZQSAT. The solver ZQSAT is able to detect equal subproblems in an effective fashion.

On the eccentricity computation problems used in the experimental evaluation presented in [43] the solver ZQSAT clearly outperformed all other non-specialized QBF

solvers. However, on the other benchmark families it did not perform as well. Surprisingly, the authors claim that the chosen variable ordering within a quantifier block does not improve the performance of their approach. In fact, the overhead of choosing an ordering caused a worse performance than using the initial variable ordering of quantifier prefix as specified in the input problem file.

In [41] an *incomplete* QBF solver was introduced. It utilizes an incomplete SAT solver within DPLL-QBF. At each node in the DPLL search space a SAT solver was invoked. If a SAT solution was found it could be heuristically followed in an attempt to reach a successful leaf in the QBF search. The SAT solver itself applied a state of the art dynamic variable ordering scheme so that the determination of a solution of F is not constrained by the imposed ordering of the quantifier prefix.

If the incomplete solver returned "unknown" no information for the search was gained. And even if the SAT solver could determine a solution π of F it is not the case that the assignments made in π can be followed in the QBF context due to the stronger reasoning (universal reduction). That is, not every SAT solution to F can serve as a path in a Q-Model. Given this, an incomplete SAT solver used in this way becomes nothing more than an expensive value ordering heuristic. Consequently, the empirical results of the implemented solver called WalkQSat solver reported in [41] did not display good performance.

2.4 Reduction to SAT

In general, all other approaches to solve QBFs are based on the idea of the quantified propositional formula into an unquantified propositional formula. In other words, these approaches attempt to convert QBF into SAT.

One method of reduction is based on the original resolution procedure of Davis-Putnam [31]. Another closely related approach employs Skolemization [100] to obtain a

purely propositional formula. We first describe how variable elimination is employed to remove the quantifier prefix $\vec{Q}$. Subsequently we discuss several extensions to this basic scheme. Finally, we show how QBF problems can be solved by utilizing Skolemization.

2.4.1 Variable Elimination for QBF

The variable elimination scheme based on the original resolution procedure of Davis-Putnam [31] is utilized to remove existential variables. On an innermost existential variable $v \in qb_n$ *all* possible resolutions with respect to v are carried out and all clauses that contain either v or $\neg v$ are subsequently removed. Consequently, if all existentials from a tailing quantifier block (e.g., qb_n) are removed all universals from the directly preceding quantifier block (e.g., qb_{n-1}) can be removed by universal reduction: all universals $u \in qb_{n-1}$ are now tailing universals in any clause they appear in.

In this way the quantifier prefix $\vec{Q}$ can be entirely removed by applying resolution from the inside out. If the outermost quantifier block is existential we can stop this process after removing all inner quantifier blocks. The remaining formula F' is then a SAT instance and can be solved using any SAT technique (including continuing to preform variable elimination). If F' is (UN)SAT this implies that the original QBF $\vec{Q}.F$ is Q(UN)SAT. If the outermost quantifier block is universal, then variable elimination must be applied to the end, and the process ends with a round of universal reduction. This will yield either the empty set of clauses implying that the original QBF was Q-SAT, or a set of empty clauses implying that the original QBF was Q-UNSAT. However, in general the reduction to SAT based on variable elimination can result in an exponential blow up of the original formula F.

2.4.2 Resolve and Expand

In [14] the variable elimination scheme is combined with the expansion of universally quantified variables in order to reduce the size of the resulting grounded theory. A

universal contained in the innermost universal quantifier block in $\vec{Q}$ is evaluated to its two possible truth assignments (the expansion step). After its evaluation it is removed from its quantifier block. Now we describe in more detail how to evaluate a universal to both truth values.

To expand a universal variable u from the innermost universal quantifier block qb_{n-1} (with qb_n denoting the innermost existential quantifier block) it is necessary to generate a copy of all existential variables in the subsequent quantifier block qb_n.

At the same time a copy of all the clauses that contain a variable $v \in qb_n$ is generated by duplicating each of these clauses and substituting every $v \in qb_n$ by its corresponding copy. Let S_1 denote the set of original clauses containing a variable $v \in qb_n$ and S_2 denotes the set of clauses consisting of the copy.

Then the universal $u_i \in qb_{n-1}$ selected for expansion is set to TRUE in S_1 and to FALSE in S_2. Hence, the expansion step applies the following reduction on these clauses: $S_1|_{\neg u_i}$ and $S_2|_{u_i}$. Consequently, the resulting propositional formula $F^{'}$ after the expansion of u_i is equal to $(F \setminus S_1) \cup S_1|_{u_i} \cup S_2|_{\neg u_i}$. Furthermore, the copy of the existential variables $v \in qb_n$ is added to the innermost quantifier block and the expanded universal u_i is removed from qb_{n-1}. As an example consider the following trivial QBF: $\forall u \exists e (u \wedge e)$ Expanding u results in the new non-simplified formula: $\exists e_1 e_2 (\mathbf{true} \vee e_1) \wedge (\mathbf{false} \vee e_2)$.

This method of expanding a single universal can be applied to every universal variable in the innermost universally quantified block of variables. Consequently, it is possible to remove all universal quantified variables by interleaving resolution and expansion so that in the end there remains a single existential quantifier block. Hence, the resulting problem now is a SAT instance whose SAT/UNSAT status corresponds to the QSAT/QUNSAT status of the original QBF.

Quantor [14] is a state of the art QBF solver that is based on a combination of variable elimination and universal expansion. It employs a sophisticated scheduling heuristic to alternate between resolution and expansion. The scheme tries to estimate the costs of

expansion and resolution and applies the less expensive operation. Hence, the heuristic is used to pick either a universal to expand or an existential to resolve with the goal of minimizing the increase of the resulting formula F. In addition, equivalence reasoning and subsumption are utilized to minimize the size of the grounded theory. Finally, a state of the art SAT solver is employed to solve the resulting propositional formula.

On some benchmark families Quantor outperforms all state of the art search based solvers [14, 90]. In addition, it can solve instances that can not be solved by any search based solver. However, it is also the case that search based solvers are able to solve a range of problems that Quantor is not able to solve (mainly due to memory limitations) [76, 77, 75].

More recently, [21] introduces a preprocessor based on Quantor. The main goal of the preprocessor is to simplify the theory by employing variable elimination and universal expansion without blowing up the theory by more than some fixed factor (e.g., at most double the size of the original theory). The simplified version of the theory is then outputted and a QBF solver is applied on it. The empirical results are not completely convincing due to the limited amount of benchmarking and several negative results on non-random benchmarks. Nevertheless the approach shows some promise and is worth further investigation.

2.4.3 Symbolic Quantifier Elimination

In [79] symbolic quantifier elimination for QBFs was introduced. The technique is not only based on the resolution procedure of Davis-Putnam [31], but also on a ZBDD [71] representation of the clauses in F. Again the variables are eliminated from the inside out. The elimination of existentials is based on *multi-resolution* [26].

Multi-resolution performs all possible resolutions of a variable v in a single operation through operations on the ZBDD representation of clauses. In other words, multi-resolution is the previously described variable elimination applied in a symbolic fashion.

1: ⟨bool *Result*⟩ **QBF-Multi-Resolution**$(F, S, \vec{v})$
2: *S is the ZDD representation of the clauses in F*
3: *$\vec{v}$ is the sorted list of variables in F, so that $qb(v_i) \geq qb(v_{i+1})$*
4: **for** $i = 1$ to n **do**
5: **if** v_i is existential **then**
6: $S \Leftarrow (S_{v_i^-} \times S_{v_i^+}) + S_{v_i'}$ (Multi Resolution on Existentials)
7: **else**
8: $S \Leftarrow S_{v_i^-} + S_{v_i^+} + S_{v_i'}$ (Removal of Universals)
9: $S \Leftarrow UnitProp(S)$
10: **endfor**
11: **return** $\langle\ S \neq \{\emptyset\}\ \rangle$

Figure 2.2: Multi-Resolution for QBF employing a ZBDD representation of clauses

However, due to the ZDD representation the cost of applying multi-resolution does not depend on the number of clauses, rather it depends on the size of the encoding of these clauses [26].

With this representation it is possible to perform many resolutions in one step although a huge number of clauses might be involved. Nevertheless, this requires that the underlying ZDDs have reasonable size. Hence, if the representation does not yield a significant compression of the clauses the application of multi-resolution is inefficient [26].

The implemented algorithm QMRES introduced in [79] operates on the ZBDD representation of clauses and applies resolution in a symbolic fashion. Starting from the innermost quantifier block qb_n all existential variables are eliminated by multi-resolution. Then all universals are eliminated by applying universal reduction. The algorithm is mainly based on the following two operators. The crossproduct operator $\times$ generates from two sets of clauses S_1 and S_2 the set $S = \{c | \exists c_1 \in S_1, \exists c_2 \in S_2, c = c_1 \cup c_2\}$.

And the + operator applied on two sets of clauses S_1 and S_2 results in the subsumption free set S so that for every $c \in S$ there is no $c' \subset c$ with $c' \in S$. The crossproduct operator can be used to perform multi-resolution by defining the following distinct set of clauses. Given a ZBDD f, f_l^+ and f_l^- denote the following sets $f_l^+ = \{c|c \vee l \in f\}$ and $f_l^- = \{c|c \vee \neg l \in f\}$. In addition, $f_{\setminus\{l,\neg l\}}$ denotes the set of clauses not containing l nor $\neg l$. Then multi-resolution is simply the subsumption free union of $f_l^+ \times f_l^-$ and $f_{\setminus\{l,\neg l\}}$ The complete algorithm is sketched in Figure 2.2. The experimental results gained by QMRES are competitive on a small subset of the benchmarks [79]. However, on the entire set of benchmark families its performance is rather disappointing [76].

2.4.4 Skolemization

Skizzo [9, 11] is a QBF solver based on Skolemization [100]. The QBF is lifted to a higher-order logic in order to make all the necessary functionality available. In particular the approach requires function symbols and quantification over functions. Once this expressiveness is available all existential variables are replaced by their corresponding Skolem functions. Consequently, the resulting formula consists only of universally quantified variables and Skolem functions. In the next stage the Skolem functions are in turn replaced by a conjunction of propositional formulas. These propositional formulas capture the functionality of the Skolem functions and thus they also capture the meaning of the original QBF formula. However, the resulting propositional formula is no longer in CNF. To convert it back to CNF so that a standard SAT solver can be applied requires introducing additional variables and clauses. Once this is accomplished the QBF has been successfully converted to a standard SAT problem. If the resulting propositional formula is (UN)SAT this implies that the original QBF is Q(UN)SAT. Note, however, that this process can cause exponential growth in the size of the formula [9].

To illustrate the process sketched above in more detail we go step by step through an example. Assume that we have the following input formula:

$$\forall u_1 \forall u_2 \exists e_1 \exists e_2 (u_1, u_2, e_1) \wedge (\neg u_1 \vee e_2).$$

As mentioned earlier the first step is to replace the existentially quantified variables by Skolem functions. In order to achieve this each existential must be replaced by a unique and newly introduced Skolem function. The arity of each Skolem function depends on the number of universals that the corresponding existential is scoped by.

In our example we have two existentials and consequently we have to introduce two new function symbols. We choose s_1 as Skolem function for e_1 and s_2 as the Skolem function of e_2. The arity of both Skolem functions is 2 (namely the two universals u_1 and u_2). The arity of s_2 can be further reduced since e_2 is existentially disjoint from u_2. For more details on how to reduce the dimensionality of the Skolem functions the reader is referred to [9]. In our example we obtain the two Skolem functions $s_1(u_1, u_2)$ and $s_2(u_1)$.

In the next step we can replace all existentially quantified variables by their corresponding Skolem function in the original formula and we acquire the following resulting formula:

$$(\exists s_1 \exists s_2) \forall u_1 \forall u_2 (u_1, u_2, s_1(u_1, u_2)) \wedge (\neg u_1 \vee s_2(u_1))$$

Note that we do not know what the Skolem function is, instead we only know that there exists a Skolem function (expressed by quantification over the function symbol).

The transformation of this formula to a purely propositional formula first requires for each Skolem function the addition of new existentially quantified variables and clauses. In the context of QBF each Skolem function is parameterized by a set of universal variables and the result of the function is a boolean value. First we have to introduce as many new existential variables as there are combinations of parameter settings in the corresponding Skolem function. Then for each possible input we use one unique variable to represent each corresponding value of the Skolem function.

Returning to our example we have to introduce 4 new existentially quantified variables for s_1 and 2 existentials for s_2 that represent each possible value of the Skolem functions under all possible settings of the universal variables:

s_1 *Function value*	s_2 *Function Value*
$e^1_{s_1} \triangleq s_1(u_1, u_2)$	$e^1_{s_2} \triangleq s_2(u_1)$
$e^2_{s_1} \triangleq s_1(\neg u_1, u_2)$	$e^2_{s_2} \triangleq s_2(\neg u_1)$
$e^3_{s_1} \triangleq s_1(u_1, \neg u_2)$	
$e^4_{s_1} \triangleq s_1(\neg u_1, \neg u_2)$	

Note that each newly introduced existential is quantified in the *outermost* quantifier block.

Now we can represent each Skolem function by a propositional formula using the newly introduced variables from before. In the propositional formula we use each parameter setting as an indicator for the value of the Skolem function evaluated with the corresponding parameters. In our example we have the following correspondence:

$$s_1(u_1, u_2) \triangleq (u_1 \wedge u_2 \implies e^1_{s_1}) \wedge (\neg u_1 \wedge u_2 \implies e^2_{s_1}) \wedge (u_1 \wedge \neg u_2 \implies e^3_{s_1}) \wedge (\neg u_1 \wedge \neg u_2 \implies e^4_{s_1})$$

$$s_2(u_1) \triangleq (u_1 \implies e^1_{s_2}) \wedge (\neg u_1 \implies e^2_{s_2})$$

For instance, for $s_1(u_1, \neg u_2)$ all premises in each implication are falsified except for $(u_1 \wedge \neg u_2 \implies e^3_{s_1})$. Hence, we capture the fact that the value of $s_1(u_1, \neg u_2)$ corresponds to the one of $e^3_{s_1}$.

Obviously, this propositional encoding is not in clausal form (due to the implications) and consequently it has to be further transformed to achieve a proper clausal representation:

$$s_1(u_1, u_2) \triangleq (\neg u_1 \vee \neg u_2 \vee e^1_{s_1}) \wedge (u_1 \vee \neg u_2 \vee e^2_{s_1}) \wedge (\neg u_1 \vee u_2 \vee e^3_{s_1}) \wedge (u_1 \vee u_2 \vee e^4_{s_1})$$

$$s_2(u_1) \triangleq (\neg u_1 \vee e^1_{s_2}) \wedge (u_1 \vee e^2_{s_2})$$

Now we can substitute each Skolem function by its corresponding propositional representation. Here we show the substitution of s_2 in the clause $(\neg u_1 \vee s_2(u_1))$:

$$\neg u_1 \vee ((\neg u_1 \vee e^1_{s_2}) \wedge (u_1 \vee e^2_{s_2}))$$

This substitution has to be converted to CNF as shown in the following formula:

$$(\neg u_1 \vee \neg u_1 \vee e^1_{s_2}) \wedge (\neg u_1 \vee u_1 \vee e^2_{s_2}))$$

It is important to note that in this conversion to CNF no new variables can be introduced in order to obtain a more compact formula. If one would add new existential variables to the formula they had to be quantified in the innermost quantifier block. Consequently, the universally quantified variables would not be tailing anymore and the resulting formula would be a QBF and not a SAT instance as required. The potential exponential growth in the size of the formula takes place during this conversion to CNF.

However, the resulting clauses can be further simplified (e.g., perform universal reduction, removal of tautologies) and in our example only $e^1_{s_2}$ remains. The resulting formula does not contain any universals since all of them can be removed by universal reduction (all existentials are in the outermost quantifier block as mentioned earlier). Consequently, the simplified formula is a propositional formula in CNF which does only consist of existential variables as required. Skizzo then employs a current SAT solver (e.g., [73]) on this formula and the result corresponds to the truth value of the original QBF.

Skizzo is based on the presented approach, but it also employs other methods for simplifying the resulting propositional formula some of which are based on symbolic reasoning. Despite these simplifications the conversion of the skolemized formula F to CNF can cause exponential growth in the size of the formula as mentioned above.

Benchmark results indicate that this approach is very competitive on a wide range of benchmarks ([74, 75]). However, Skizzo is a mixture of many different techniques (e.g., search, symbolic reasoning, and SAT solving) so the data on its empirical performance does not allow one to quantify the specific impact of Skolemization on its overall performance.

2.4.5 SAT Sampling

An order unconstrained approach based on a BDD representation of a Q-model is presented in[4]. The idea here is to generate arbitrary SAT solutions with a SAT solver, adding those solutions to the BDD. The BDD will eventually collapse to TRUE if the set of added SAT solutions suffice to form all paths in a Q-model.

However, the BDD can grow to an exponential size prior to collapsing. Furthermore, the SAT solver can generate SAT solutions that form paths in disjoint Q-models—thus the BDD might be even larger as it has to represent multiple distinct Q-models before one collapses to a solution. The empirical results reported in [4] do not improve on the state of the art.

2.5 Experimental Evaluation

As shown in the previous sections there already exists a variety of algorithms that solve QBF. The problem is to classify those algorithms by there efficiency in practice. Some algorithms might perform very well on some problem instances of QBF but exhibit an exponential blow up, either with respect to time or space, on other instances.

In order to proceed towards a competitive and comprehensive comparison of different solvers the quantified boolean formulas satisfiability library (QBFLIB [45]) was created in 2001. Its main goal is to provide an uniform test-bed for the empirical characterization of QBF solvers [45]. Since 2006 the QBFLIB also organizes QBF competitions that evaluate the performance of participating QBF solvers.

To date the problems suite is currently comprised of more than 13,000 instances. Among this total number of instances approximately 1,000 structured problem instances exist (the remaining instances are randomly generated). The structured instances are distributed among 25 benchmark families. They have been contributed by members of the community from different problem domains like planning, formal verification and

model checking. The size of these problem range between hundreds of variables and clauses to about $10,000$'s of variables and $100,000$'s of clauses. Also the number of quantifier alternation varies between 1 and about 40. However, in the vast majority of problem instances the number of quantifier alternations is low (between $1-3$).

While this collection is not yet as extensive as the one available for SAT (SATLIB [53]) it already enabled QBF researchers to compare their approaches in a comprehensive fashion. However, in contrast to SAT, there exists one additional major difficulty with the experimental evaluation of QBF. In comparison to SAT the verification of a QBF solution is as hard as solving the QBF itself. Consequently, the soundness of a result returned by a QBF solver is not easily verifiable. The research on the verification of QBF solutions is quite recent and still in progress [59, 107]. Achieved results in this area will further improve the experimental evaluation of QBF solvers.

2.6 Conclusions

In this chapter we have set the context for this thesis. Since all presented approaches in this thesis are improvements of the search-based DPLL algorithm for QBF we focused our attention on this technique. We explained in detail how DPLL for QBF evaluates the truth value of a given input formula.

In addition, we have reviewed the main techniques employed in practice for solving quantified boolean formulas. Two techniques are mainly based on well established algorithms, namely DPLL [30] and DP (variable elimination) [31]. They rely heavily on techniques that were developed by the SAT community over the last decade. Skolemization is a novel approach and benchmark results indicate that it is also worthwhile. While the class of algorithms that try to achieve a reduction to SAT either by DP or Skolemization require exponential space to assure correctness, the search based algorithms are exponential in time, but are any-space algorithms as described in Section 2.3.3.

The question of whether space intensive algorithms like Quantor [14], Skizzo [9], or QMRES [79] will eventually be the best way to solve QBF is still open. So far space exponential algorithms have performed very well on several benchmark families and do clearly outperform search-based solvers on these benchmarks. However, there exist several benchmarks that can be solved in only a few seconds by search-based solvers but cause for example Quantor to exhaust available memory. In addition, the wide variance in the run times achieved by search based solvers (see comparisons of search based solvers in e.g., [90]) indicate that there is a lot of room for performance improvement.

The results of the QBF competition 2006 [75] indicate that the "best" QBF solver would probably use a portfolio approach rather than any single solver. For example, our 2clsQ [96] (Chapter 3) entry which won the 2006 competition first applied a hyper binary preprocessor (PreQuel [92, 93], Chapter 3), then it ran the QBF solver Quantor [14] for a fixed period of time. Finally if the problem was still not solved 2clsQ was invoked on output of PreQuel. Due to these results and our intuition that time is a more flexible resource than space we have mainly concentrated our research efforts on search based methods.

Chapter 3

Extended Binary Resolution

3.1 Introduction

In this chapter we present a novel technique that applies a stronger rule of inference during search to improve QBF solvers. Like many techniques used for QBF, ours is a modification of techniques already used in SAT. We employ extended binary clause reasoning in order to process QBF instances in a static as well as a dynamic fashion.

In the static context we preprocess the input formula, without changing its meaning, so that it becomes easier to solve for a QBF solver. As we demonstrate at the end of this chapter our technique can be extremely effective, sometimes reducing the time it takes to solve a QBF instance by orders of magnitude. We also investigate applying extended binary resolution at each search node during the DPLL algorithm and discuss its impact on QBF solving.

The outline of this chapter is as follows. We start by presenting and discussing the underlying technique. In particular we show how to lift extended binary resolution as it has been applied within SAT to the QBF context. We not only uncover several differences with the SAT case, but also demonstrate how to resolve the issues that arise from those differences. We then present further details of our approach and state some

results about its correctness. Finally, we provide empirical evidence of the effectiveness of our approach, and close with a discussion of future work and some conclusions.

3.2 Hyper Binary resolution

The foundation of our polynomial time technique for processing QBFs is the SAT method of reasoning with binary clauses using hyper-resolution developed in [5, 7]. This method reasons with CNF SAT theories using the following "*HypBinRes*" rule of inference:

Definition 5 *Given a single n-ary clause $c = (l_1, l_2, ..., l_n)$, D a subset of c, and the set of binary clauses $\{(\ell, \neg l_i) | l_i \in D\}$, infer the new clause $b = (c - D) \cup \{\ell\}$ if b is either binary or unary.*

For example, from (a, b, c, d), $(h, \neg a)$, $(h, \neg c)$ and $(h, \neg d)$, we infer the new binary clause (h, b), similarly from (a, b, c) and $(b, \neg a)$ the rule generates (b, c). The *HypBinRes* rule covers the standard case of resolving two binary clauses (from (l_1, l_2) and $(\neg l_1, \ell)$ infer (ℓ, l_2)) and it can generate unit clauses (e.g., from (l_1, ℓ) and $(\neg l_1, \ell)$ we infer $(\ell, \ell) \equiv (\ell)$).

The advantage of *HypBinRes* inference is that it does not blow up the theory (it can only add binary or unary clauses to the theory) and that it can discover a lot of new unit clauses. These unit clauses can then be used to simplify the formula by doing unit propagation which in turn might allow more applications of *HypBinRes*. Applying *HypBinRes* and unit propagation until closure (i.e., until nothing new can be inferred) uncovers *all* failed literals. That is, in the resulting reduced theory there will be no literal ℓ such that forcing ℓ to be true followed by unit propagation results in a contradiction. This and other results about *HypBinRes* are proved in the above references.

In addition to uncovering unit clauses we can use the binary clauses to perform equality reductions. In particular, if we have two clauses $(\neg x, y)$ and $(x, \neg y)$ we can replace all instances of y in the formula by x (and $\neg y$ by $\neg x$). This might result in some tautological clauses which can be removed, and some clauses which are reduced in length

because of duplicate literals. The reduction might yield new binary or unary clauses which can then enable further *HypBinRes* inferences. Taken together *HypBinRes* and equality reduction can significantly reduce a SAT formula removing many of its variables and clauses [7].

3.3 Hyper Binary Resolution in QBF

Given a QBF $\vec{Q}.F$ we could apply *HypBinRes*, unit propagation, and equality reduction to F until closure. This would yield a new formula F', and the QBF $\vec{Q}'.F'$ where $\vec{Q}'$ is $\vec{Q}$ with all variables not in F' removed. Unfortunately, there are two problems with this approach. One is that the new QBF $\vec{Q}'.F'$ might not be Q-equivalent to $\vec{Q}.F$, so that this method is not sound. The other problem is that we miss out on some important additional inferences that can be achieved through universal reduction. We elaborate on these two issues and show how they can be overcome.

The reason why the straightforward application of *HypBinRes*, unit propagation and equality reduction to the body of a QBF is unsound, is that the resulting formula F' does not have exactly the same SAT models as F, as it is required by Lemma 1 from Chapter 2.

In particular, the models of F' do not make assignments to variables that have been removed by unit propagation and equality reduction. Hence, a Q-model of $\vec{Q}'.F'$ might not be extendable to a Q-model of $\vec{Q}.F$. For example, if unit propagation forced a universal variable in F, then $\vec{Q}'.F'$ might be QSAT, but $\vec{Q}.F$ is not (no Q-model of $\vec{Q}.F$ can exist since the paths that set the forced universal to its opposite value will not be SAT models of F).

This situation occurs in the following example. Consider the following QBF:

$$\vec{Q}.F = \exists abc \forall x \exists yz (x, \neg y)(x, z)(\neg z, y)(a, b, c)$$

We can see that $\vec{Q}.F$ is not QSAT since when x is false, $\neg y$ and z must be true, falsifying

the clause $(\neg z, y)$. If we apply *HypBinRes* and unit propagation to F, we obtain $F' = (a, b, c)$. Note that the universal variable x has been unit propagated. As anticipated, $\vec{Q}'.F' = \exists abc(a, b, c)$ is QSAT, so this reduction of F has not preserved the QSAT status of the original formula. However, it is easy to fix this problem. Making unit propagation sound for QBF simply requires that we regard the unit propagation of a universal variable as equivalent to the derivation of the empty clause. This follows directly from applying universal reduction to the unit universal clase. Universal reduction is applied in all search based QBF solvers, hence all of them treat unit propagation of a universal variable as a contridiction.

Ensuring that equality reduction is sound for QBF is a bit more subtle. Consider a formula F in which we have the two clauses $(x, \neg y)$ and $(\neg x, y)$. Since every path in any Q-model satisfies F, this means that along any path x and y must have the same truth value. However, in order to soundly replace all instances of one of these variables by the other in F, we must respect the quantifier ordering. In particular, if $x <_q y$ then we must replace y by x. It would be unsound to do the replacement in the other direction.

For example, say that x appears in quantifier block 3 while y appears in quantifier block 5 with both x and y being existentially quantified. The above binary clauses will enforce the constraint that along any path of any Q-model once x is assigned y must get the same value. In particular, y will be invariant as we change the assignments to the universal variables in quantifier block 4. This constraint will continue to hold if we replace y by x in all of the clauses of F. However, if we perform the opposite replacement, we would be able to make y vary as we vary the assignments to the universal variables of quantifier block 4: i.e., the opposite replacement would weaken the theory perhaps changing its QSAT status. The same reasoning holds if x is universal and y is existential. However, if y is universal, the two binary clauses imply that we will never have the freedom to assign y its two different truth values. That is, in this case the QBF is UNQSAT, and we can again treat this case as if the empty clause has been derived.

Again this conclusion can be justified by universal reduction. If $x <_q y$ and y is universal the two clauses $(x, -y)$ and $(-x, y)$ can be universally reduced to (x) and $(-x)$ which immediately resolve together to produce the empty clause.

Therefore a sound version of equality reduction must respect the variable ordering. We call this ($<_q$ preferred) equality reduction. That is, if we detect that x and y are equivalent and $x <_q y$ then we always remove y from the theory replacing it by x. With this restriction on equality reduction we have the following result:

Proposition 1 *Let F' be the result of applying HypBinRes, unit propagation, and ($<_q$ preferred) equality reduction to F until closure. If F' has the same set of universal variables as F (i.e., no universal variable was removed by unit propagation or equality reduction), then the Q-models of $\vec{Q}'.F'$ are in 1-1 correspondence with the Q-models of $\vec{Q}.F$. In particular, $\vec{Q}.F$ is QSAT iff $\vec{Q}'.F'$ is QSAT. On the other hand, if F' has fewer universal variables than F then $\vec{Q}.F$ is UNQSAT.*

Proof: In order to prove that the Q-models of Q.F and Q'.F' are 1-1, we first show correspondence between the SAT models of F and F'. Clearly, any SAT model of F can be mapped to a SAT model of F' by simply omitting the assignments of variables not in F'. This holds since the applied operations are sound transformations of the propositional formula. In the other direction we can map any SAT model of F' to a SAT model of F by assigning all forced variables their forced value, and assigning all equality reduced variables a value derived from the variable they were made equivalent to. That is, if x was removed because it was equivalent to $\neg y/y$, we assign x the opposite/same value assigned to y in F''s SAT model.

Given this relationship we now show that any Q-model $\mathcal{M}_q$ of $\vec{Q}.F$ can be mapped to a unique Q-model $\mathcal{M}'_q$ of $\vec{Q}.F'$ and vice versa. Here we assume that F' has the same universals as F. First we show that any Q-model $\mathcal{M}_q$ of $\vec{Q}.F$ can be mapped to a Q-model $\mathcal{M}'_q$ of $\vec{Q}.F'$. By Definition 2 each complete path in a Q-model corresponds to a SAT model. Consequently, $\mathcal{M}_q$ can be transformed to $\mathcal{M}'_q$ by removing all nodes in the Q-tree

that correspond to variables which do not appear in F'. Due to the correspondence between the SAT models each complete path is a SAT model for F' after eliminating these nodes and the tree still contains a path for every setting of the universal variables of F'.

Now we show that any Q-model $\mathcal{M}'_q$ of $\vec{Q}.F'$ can be mapped to a Q-model of $\vec{Q}.F$. We can accomplish this by adding nodes to the Q-tree of $\mathcal{M}'_q$ while obeying $\vec{Q}$. These added nodes are will be labeled with values for the variables that appear in F but not in F'. We use the described mapping from F' to F to determine the setting of the variables in these nodes. Thus each complete path of $\mathcal{M}'_q$, whiich is a SAT model of F', is converted to a complete in Mq that is a SAT model of F. Since we did not alter any universal nodes in any Q-tree (any removed or added nodes must be existentials since F' has the same number of universals as F) and each complete path in the constructed Q-trees is a SAT model of the appropriate formula, both transformations achieve the required Q-models.

Finally, we show that the described mapping is unique. Let $\mathcal{M}_q$ and $\mathcal{M}^*_q$ be different Q-models of $\vec{Q}.F$. Let $\mathcal{M}'_q$ denote the mapping of $\mathcal{M}_q$ to a Q-model of $\vec{Q}.F'$. Assume that $\mathcal{M}^*_q$ also maps to $\mathcal{M}'_q$. Since $\mathcal{M}_q$ and $\mathcal{M}^*_q$ are different they must disagree in at least one variable assignment and this variable must be existentially quantified. Then all of the employed operations are affected by this altered assignment. Consequently, the corresponding mapping of $\mathcal{M}^*_q$ to a Q-model of $\vec{Q}.F'$ must be different from $\mathcal{M}'_q$. The other direction is similar.

This gives us that the number of Q-models of each formula are equal and that the transformation must be a 1-1 mapping. Consequently, $\vec{Q}.F$ and $\vec{Q}'.F'$ are equivalent formulas and in particular $\vec{Q}.F$ is QSAT iff $\vec{Q}'.F'$ is QSAT.

If F' has fewer universals than F (it cannot have more since it can only have fewer variables than F) it follows directly that $\vec{Q}.F$ is UNQSAT because one of the sound operations applied in the transformation to F' has inferred that a universal has a forced value. ∎

This proposition tells us that we can use a SAT based reduction of F with QBF, as long as we ensure that equality reduction respects the quantifier ordering and check for the removal of universals. This approach, however, does not fully utilize the power of universal reduction as discussed in Chapter 2. So instead we use a more powerful approach that is based on the following modification of *HypBinRes* which "folds" universal reduction into the inference rule. We call this rule "*HypBinRes*+UR":

Definition 6 *Given a single n-ary clause $c = (l_1, l_2, ..., l_n)$, D a subset of c, and the set of binary clauses $\{(\ell, \neg l_i) | l_i \in D\}$, infer the universal reduction of the clause $(c \setminus D) \cup \{\ell\}$ if this reduction is either binary or unary.*

For example, from $c = (u_1, e_3, u_4, e_5, u_6, e_7)$, $(e_2, \neg e_7)$, $(e_2, \neg e_5)$ and $(e_2, \neg e_3)$ we infer the new binary clause (u_1, e_2) when $u_1 \leq_q e_2 \leq_q e_3 \leq_q u_4 \leq_q e_5 \leq_q u_6 \leq_q e_7$. Note that without universal reduction, *HypBinRes* would need 5 binary clauses in order to reduce c, while with universal reduction, 2 fewer binary clauses are required. This example also shows that *HypBinRes*+UR is able to derive clauses that *HypBinRes* cannot. Since clearly *HypBinRes*+UR can derive anything *HypBinRes* can, *HypBinRes*+UR is a more powerful rule of inference.

In addition to using universal reduction inside of *HypBinRes* we must also use it when unit propagation is used. For example, from the two clauses $(e_1, u_2, u_3, u_4, \neg e_5)$ and (e_5) (with $e_1 <_q u_i$) unit propagation by itself can only derive (e_1, u_2, u_3, u_4), but unit propagation with universal reduction can derive (e_1).

It turns out that in addition to gaining more inferential power, universal reduction also allows us to obtain the unconditionally sound processing we would like to have.

Proposition 2 *Let F' be the result of applying HypBinRes+UR, unit propagation, universal reduction and ($<_q$ preferred) equality reduction to F until closure, where we always apply universal reduction before unit propagation. Then the Q-models of $\vec{Q}'.F'$ are in 1-1 correspondence with the Q-models of $\vec{Q}.F$.*

Proof: This result can be proved given Proposition 1 by showing that universal reduc-

tion generates the empty clause whenever a universal variable is to be unit propagated or removed via equality reduction. For example, for a universal u to be forced it must first appear in a unit clause (u), but then universal reduction would generate the empty clause (given that we apply universal reduction before unit propagation). Similarly, to make a universal variable u equivalent to an existential variable e with $e \leq_q u$ we would first have to generate the two binary clauses $(e, \neg u)$ and $(\neg e, u)$ which after universal reduction would yield (e) and $(\neg e)$ which after unit propagation would yield the empty clause. Thus the cases where Proposition 1 fails to preserve Q-models are directly detected through the generation of an UNQSAT $\vec{Q}'.F'$. In this case we still preserve the Q-models—neither formula has any. ■

We close this section on the properties of hyper binary resolution in the context of QBF with the following complexity result.

Proposition 3 *Applying HypBinRes+UR, unit propagation, universal reduction and ($<_q$ preferred) equality reduction to $\vec{Q}.F$ until we reach closure can be done in time polynomial in the size of F.*

Proof: This result is proved by the following three observations: (1) F can never become larger than $|F|^2$ since we are only adding binary clauses, (2) there are at most a polynomial number of rule applications possible before closure since each rule either reduces a clause, removes a variable or adds a binary clause, and (3) at each stage detecting if another rule can be applied requires only time polynomial in the current size of the theory. ■

3.4 Preprocessor

In our first investigation of applying extended binary clause reasoning to QBF we constructed a QBF preprocessor PreQuel [93]. That is PreQuel is run on the input formula $\vec{Q}.F$ to generate an equivalent formula $\vec{Q}'.F'$ that should be easier to solve. Prequel

modifies $\vec{Q}.F$ exactly as described in Proposition 2. It applies *HypBinRes*+UR, unit propagation, universal reduction, and ($<_q$ preferred) equality reduction to F until it reaches closure. It then outputs the new formula $\vec{Q}'.F'$. Proposition 2 shows that this modification of the formula is sound, i.e., it does not change the QSAT status of the formula.

To implement the preprocessor we adapted the algorithm presented in [7] which exploits a close connection between *HypBinRes* and unit propagation. In particular, this algorithm uses trial unit propagations to detect new *HypBinRes* inferences. The main changes required to make this algorithm work for QBF were adding universal reduction, modifying the unit propagator so that it performs universal reduction prior to any unit propagation step, and modifying equality reduction to ensure it respects the quantifier ordering.

To understand how trial unit propagation is used to detect *HypBinRes*+UR inferences, consider the example above of inferring (u_1, e_2) from $(u_1, e_3, u_4, e_5, u_6, e_7)$, $(e_2, \neg e_7)$, $(e_2, \neg e_5)$ and $(e_2, \neg e_3)$. If we perform a trial unit propagation of $\neg e_2$, dynamically performing universal reduction we obtain the unit clause (u_1). Because the trial propagation started with $\neg e_2$ this unit clause actually corresponds to the binary clause (u_1, e_2) (i.e., $\neg e_2 \rightarrow u_1$). The trial unit propagation has to keep track of the "root" of the propagation so that it does not erroneously apply universal reduction (every clause reduced during this process implicitly contains e_2).

3.5 Dynamic Employment

We have also implemented *HypBinRes*+UR in a DPLL based QBF solver by modifying the 2clsEq SAT solver [5]. The resulting QBF solver, 2clsQ [96], performs *HypBinRes*+UR reasoning at every node of the search tree. An abstract outline of its algorithm is shown in Algorithm 3 which is an extension of Algorithm 2 displayed in Chapter 2.

The following changes were made to the 2clsEq SAT solver to make it into a QBF solver. First, branching on a variable had to be constrained so that the quantifier ordering is respected.

Second, equality reduction had to be modified so that it respects the quantifier ordering. In the 2clsEq implementation an entire set of variables could be detected to be equivalent at once, so we must pick a variable v from the outermost quantifier block among that set and then replace all of the other variables with v.

1: $\langle$bool *Result*, int *BTLevel*$\rangle$ **2clsQ**($\vec{Q}.F$, *Level*)
2: **if** *F contains an [empty clause/is empty]* **then**
3: *Compute a new [clause/cube] and backtrack level BTLevel by [conflict/solution] analysis*
4: **return** $\langle$*FAIL/SUCCEED*,*BTLevel*$\rangle$
5: Pick v from the first quantifier block and let $\ell = v$ or $\neg v$
6: $\vec{Q}.F = HypBinRes + UR(\vec{Q}.F|_\ell)$ *i.e., reduce by HypBinRes+UR, equality reduction, universal reduction, and unit propagation*
7: $\langle$*Result*, *BTLevel*$\rangle$ = **2clsQ**$\big(\vec{Q}.F, Level + 1\big)$
8: **if** *BTLevel* $<$ *Level* **then**
9: **return** $\langle$*Result*, *BTLevel*$\rangle$
10: **if** *v is [universal/existential]* **then**
11: Compute new [cube/clause] from the [cubes/clauses] learned from v and $\bar{v}$ by resolution
12: **return** ([*SUCCEED/FAIL*], *BTLevel*)

Algorithm 3: 2clsQ Algorithm. Invoked with the original QBF and *Level=1*. Returns (*SUCCEED*, 0) indicating QSAT or (*FAIL*, 0) indicating UNQSAT.

Third, we had to modify the code that tested for possible new applications of *HypBinRes* to account for universal reduction. When a new binary clause (x, y) is generated

we can continue to test all clauses containing $\bar{x}$ as well as all clauses containing $\bar{y}$ to see if this new binary clause triggers any new applications of *HypBinRes+UR*. For example, if $\bar{x} \in c$, we determine the set S of other literals $\ell \in c$ that can be resolved away from c by binary clauses of the form $(y, \bar{\ell})$. Then we check if $c - S$ can be universally reduced to a clause of length 2 or less. The other trigger used in 2clsEq for new applications of *HypBinRes* occurs when a k-ary clause has been reduced in size, as discussed above.

Unfortunately, detecting if the reduction of a k-ary clause generates any new binary or unary clauses under *HypBinRes+UR*is relatively expensive. With just *HypBinRes* when a clause c has just been reduced in size to length i, we only need to look for a literal x such that there are $i - 1$ binary clauses $(x, \bar{\ell})$ with $\ell \in c$. From these clauses we can then infer a new binary clause (x, y), where $y \in c$ is the single literal not covered in the set of clauses $(x, \bar{\ell})$. This can be accomplished relatively efficiently by first taking any two literals of c, l_1 and l_2 and examining the set of literals $L = \{y |$ either $(y, \bar{l}_1)$ or $(y, \bar{l}_2)$ exists$\}$. We then know that any literal x satisfying the above condition must be in L—any such literal must have a binary clause with one of $\bar{l}_1$ or $\bar{l}_2$—and we can restrict our attention to the literals in L.

Unfortunately, this strategy for limiting the set of literals to examine for potential new *HypBinRes* steps against a clause breaks down when we move to *HypBinRes+UR*. For example, consider the clauses $c = (e_1, u_1, u_2, u_3, e_2, u_4, u_5, e_3)$, $(e, \bar{e}_2)$, $(e, \bar{e}_3)$ with $e <_q e_1 <_q u_1, u_2, u_3 <_q e_2 <_q u_4, u_5 <_q e_3$. We can infer the new binary clause (e_1, e) by applying *HypBinRes+UR*. In this case, the literal e has only two binary clauses that can resolve against c, and so it does not fall into the set L defined above. Hence, it is not possible to limit our attention to the literals in L. It is still possible to detect all possible *HypBinRes+UR* inferences available from c in polynomial time, but it becomes more expensive to do so. Hence, in our implementation we do only a partial, and cheaper, test for new *HypBinRes+UR* inferences on k-ary clauses that have been reduced in size. That is, we do not achieve *HypBinRes+UR* closure in 2clsQ.

Fourth, the algorithm employs both conflict and solution analysis for learning new clauses and solution cubes. Since literals can be forced from an extensive combination of binary clause reasoning and equality reduction, it was very difficult to implement 1-UIP clause learning. Instead, 2clsQ learns 'all decision clauses' [110]. The learned clauses are used to enhance unit propagation. However, we do not perform *HypBinRes+UR* or equality reduction against these new clauses as this appears to be too expensive. Solution analysis (cube learning) is done in the manner presented in Chapter 2. The learned cubes are also used to prune branches in the search. In particular, when a universal variable is set this might trigger a cube making search below that setting unnecessary as explained in Chapter 2.

Finally, we modified the original 2clsEq branching heuristics to take into account the varying nature of QBF search. In our implementation we combined two branching heuristics in the following way. Whenever 2clsQ encounters a conflict we try to generate more conflicts by branching on variables that cause the largest number of unit propagations (under *HypBinRes* this number is equal to the number of binary clauses the variable appears in). On the other hand when 2clsQ finds a solution we try to generate more solutions by branching on variables that will satisfy the most clauses. Thus the branching heuristic switches dependent on what "mode" the search is in.

We conclude this section with the following formal results on the properties of 2clsQ.

Theorem 4 *2clsQ as shown in Algorithm 3 is sound and complete.*

Proof:

Both properties follow directly from Theorem 3 and Proposition 2. To the QBF-DPLL algorithm (see Algorithm 2) we only add a sound inference procedure and consequently the soundness of QBF-DPLL remains intact. In addition, the basic systematic recursion of QBF-DPLL is not altered. Therefore, 2clsQ is sound and complete. ∎

3.6 Empirical Results

We considered all of the non-random benchmark instances from QBFLib (2005) [45] (508 instances in total). We discarded the instances from the benchmark families von Neumann and Z since these are all very quickly solved by any state of the art QBF solver (less than 10 sec. for the entire suite of instances). We also discarded the instances coming from the benchmark families Jmc, and Jmc-squaring. None of these instances (with or without preprocessing) can be solved within our time bounds by any of the QBF solvers we tested. This left us with 468 remaining instances from 19 different benchmark families. We tested our approach on all of these instances.

All tests were run on a Pentium 4 3.60GHz CPU with 6GB of memory. The time limit for each run of any of the solvers or the preprocessor was set to $5,000$ seconds.

3.6.1 Performance of the Preprocessor

We first examine the time required to preprocess the QBF formulas by looking at the runtime behaviour of the preprocessor PreQuel on the given set of benchmark families. On the vast majority of benchmarks the preprocessing time is negligible. In particular, the preprocessing time for even the largest instances in the benchmarks Adder, Chain, Connect, Counter, FlipFlop, Lut, Mutex, Qshifter, Toilet, Tree, and Uclid is less than one second. For example, the instance Adder-16-s with $\approx 22,000$ variables and $\approx 25,000$ clauses is preprocessed in 0.3 seconds.

The benchmarks that require more effort to preprocess are C, EVPursade, S, Szymanski, and Blocks and a subset of the K benchmark:[1] k-branch-n, k-branch-p, k-lin-n, k-ph-n, and k-ph-p. To examine the runtime behaviour on these benchmark families we plot the number of input variables of each instance against the time required for preprocessing (Figure 3.1), clustering all of the K-subfamilies into one group. Both axis of the

[1]This benchmark family is divided into sub-families.

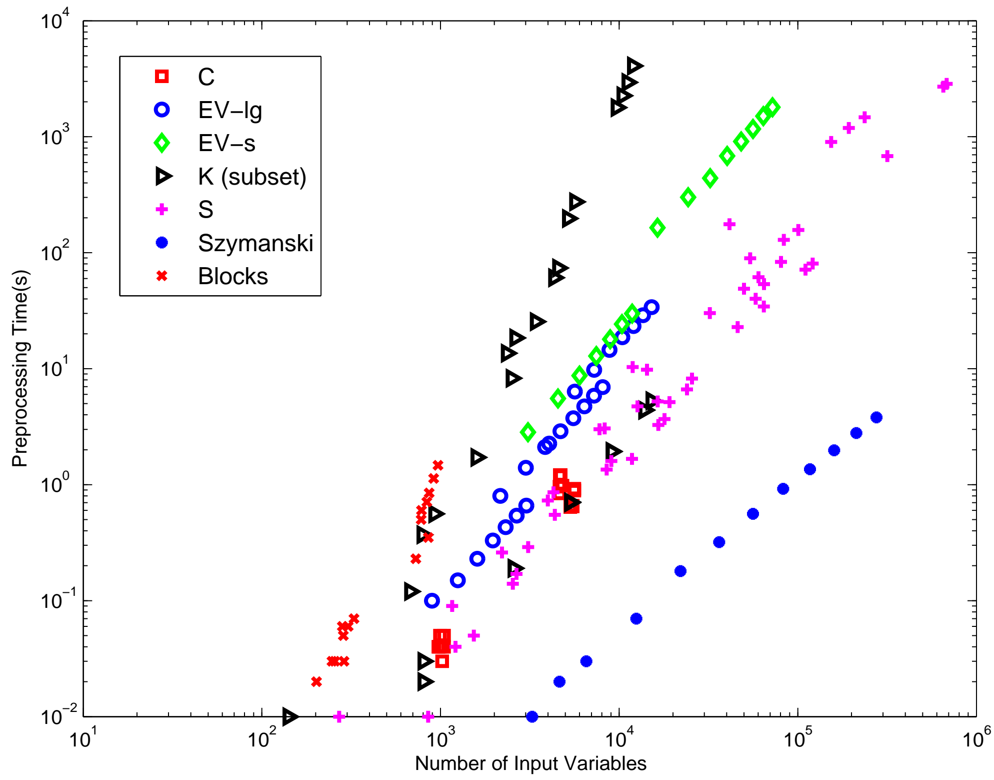


Figure 3.1: Logarithmic scale comparison between the number of input variables and the preprocessing time in seconds on a selected set of benchmark families.

plot are drawn in logarithmic scale.

Figure 3.1 shows that for all of these harder benchmarks the relationship between the number of input variables and preprocessing time is approximately linear on the loglog-plot. This is not surprising since Proposition 3 showed that the PreQuel runs in worst-case polynomial time. Any polynomial function is linear in a loglog scale with the slope increasing with the degree of the polynomial.

Fitting a linear function to each benchmark family enables a more detailed estimate of the runtime, since the slope of the fitted linear function determines the relationship between the number of input variables and preprocessing time. For instance, a slope of one indicates a linear runtime, a slope of two indicates quadratic behaviour, etc. Except for the benchmarks 'k-ph-n' and 'k-ph-p' the slope of the fitted linear function ranges between 1.3 (Szymanski) and 2.3 (Blocks) which indicates a linear to quadratic behaviour of the preprocessor. The two K-subfamilies 'k-ph-n' and 'k-ph-p' display worse behaviour, on them preprocessing time is almost cubic (slope of 2.9).

The graph also shows that on some of the larger problems the preprocessor can take thousands of seconds. However, this is not a practical limitation. In particular out of the 468 instances only 23 took more than 100 seconds to preprocess. Of these 18 could not be solved by any of our solvers, either in preprocessed form or unpreprocessed form. That is, these problems are so hard that we have no way of evaluating the effect of preprocessing them. Of the other 5 instances that were solved by some solver there exist in total 25 pairs of instances and solvers. Among these there exist only 13 pairs where some solver succeeded either on the preprocessed instance only or on both versions of the instance. On 62% of these 13 successful runs, the preprocessor yielded a net speedup. Furthermore, despite being a net slowdown on the other 38% of runs, another 38% of the runs were cases where the solver was only able to solve the preprocessed instance. So our conclusion is that except for a few instances, preprocessing is not a significant added computational burden.

Solver	*Skizzo*		*Quantor*		*Quaffle*		*Qube*		*SQBF*	
	no-pre	*pre*	*no-pre*	*pre*	*no-pre*	*pre*	*no-pre*	*pre*	*no-pre*	*pre*
# Instances	311	**351**	262	**312**	226	**238**	213	**243**	205	**239**
Time on common instances	9,748	**9,595**	10,384	**2,244**	36,382	**20,188**	41,107	**23,196**	46,147	**25,554**
Time on new instances	-	12,756	-	16,829	-	9,579	-	9,707	-	2,421

Table 3.1: Summary of results reported in Tables 3.2 and 3.3. For each solver we show its number of solved instances among all tested benchmark families with and without preprocessing, the total CPU time (in seconds) required to solve the preprocessed and un-preprocessed instances taken over the "common" instances (instances solved in both preprocessed and un-preprocessed form), and the total CPU time required by the solvers to solve the "new" instances (instances that can only be solved in preprocessed form).

3.6.2 Impact of Preprocessing

Now we examine how effective the preprocessor PreQuel is. Is it able to improve the performance of state of the art QBF solvers, even when we consider the time it takes to run? To answer this question we studied the effect preprocessing has on the performance of five state of the art QBF solvers **Quaffle** [111] (version as of Feb. 2005), **Quantor** [14] (version as of 2004), **Qube** (release 1.3) [46], **Skizzo** (v0.82, r355) [9] and **SQBF** [90] (see also Chapter 4). Quaffle, Qube and SQBF are based on search, whereas Quantor is based on variable elimination. Skizzo uses mainly a combination of skolemization, variable elimination and search, but it also applies a variety of other kinds of reasoning on the symbolic and the ground representations of the instances. Please refer to Chapter 2 for more details.

A summary of our results is presented in Table 3.1. The second row of the table shows the total time required by each solver to solve the instances that could be solved in both preprocessed and unpreprocessed form (the "common instances"). The data demonstrates that preprocessing provides a speedup for every solver. Note that the

times for the preprocessed instances *include* the time taken by the preprocessor.

On these common instances Quantor was 4.6 times faster with preprocessing, while Quaffle, Qube and SQBF were all approximately 1.8 times faster with preprocessing. Skizzo is only slightly faster on the preprocessed benchmarks (that it could already solve).

The first row of Table 3.1 shows the number of instances that can be solved within the 5000 sec. time bound. It demonstrates that in addition to speeding up the solvers on problems they can already solve, preprocessing also extends the reach of each solver, allowing it to solve problems that it could not solve before (within our time and memory bounds). In particular, the first row shows that the number of solved instances for each solver is significantly larger when preprocessing is applied. The increase in the number of solved instances is 13% for Skizzo, 19% for Quantor, 5% for Quaffle, 14% for Qube and 17% for SQBF.

The time required by the solvers on these new instances is shown in row 3. For example, we see that SQBF was able to solve 34 new instances. None of these instances could previously be solved in 5,000 sec. each. That is, 170,000 CPU seconds were expended in 34 failed attempts. With preprocessing all of these instances could be solved in 2,421 sec. Similarly, Quantor expended 250,000 sec. in 50 failed attempts, which with preprocessing could all solved in 16,829 sec. Skizzo expended 200,000 sec. in 40 failed attempts which with preprocessing could all be solved in 12,756 seconds. Quaffle expended 60,000 sec. in 12 failed attempts, which with preprocessing could all be solved in 9,579 sec. And Qube expended 150,000 sec. in 30 failed attempts, which with preprocessing could all be solved in 9,707 seconds.

These results demonstrate quite convincingly that our preprocessor technique offers robust improvements to all of these different solvers, even though some of them are utilizing completely different solving techniques.

Tables 3.2 and 3.3 provide a more detailed breakdown of the data. Table 3.2 gives a

Benchmark	*Skizzo*			*Quantor*			*Quaffle*			*Qube*			*SQBF*		
(# instances)	*Succ. %*	*time no-pre*	*time pre*	*Succ. %*	*time no-pre*	*time pre*	*Succ. %*	*time no-pre*	*time pre*	*Succ. %*	*time no-pre*	*time pre*	*Succ. %*	*time no-pre*	*time pre*
ADDER (16)	50%	954	**792**	25%	**24**	25	25%	1	1	13%	72	**27**	13%	3	**1**
adder (16)	44%	**455**	550	25%	29	**27**	42%	5	**4**	44%	**0**	1	38%	2,678	**2,229**
Blocks (16)	56%	108	**11**	100%	308	**79**	75%	1,284	**762**	69%	1774	**242**	75%	7,042	**1,486**
C (24)	25%	**1,070**	1,272	21%	140	**32**	21%	5,356	**14**	8%	3	**5**	17%	4	**0**
Chain (12)	100%	1	**0**	100%	0	0	67%	6,075	**0**	83%	4,990	**0**	58%	4,192	**0**
Connect (60)	68%	802	**5**	67%	14	**7**	70%	253	**5**	75%	7,013	**7**	67%	**0**	5
Counter (24)	54%	1,036	**731**	50%	217	**141**	38%	5	5	33%	2	**1**	38	**9**	20
EV (38)	29%	**1,450**	1,765	3%	**73**	82	26%	1,962	**1,960**	18%	4,402	**2,537**	32%	4,759	**4,508**
FlipFlop (10)	100%	6	**4**	100%	**3**	4	100%	**0**	4	100%	**1**	4	80%	5,027	**1**
K (107)	88%	**1,972**	2,228	63%	3,839	**39**	35%	21,675	**17,083**	37%	21,801	**19,203**	33%	5,563	**5,197**
Lut (5)	100%	9	9	100%	3	3	100%	1	1	100%	**3**	6	100%	1,247	**66**
Mutex (7)	100%	**0**	102	43%	**0**	1	29%	**43**	49	43%	**64**	71	43%	**1**	6
Qshift (6)	100%	**8**	9	100%	**26**	29	17%	0	0	33%	29	29	33	**1,107**	2,103
S (52)	27%	**644**	1,886	25%	**910**	1,530	2%	0	0	4%	**401**	451	2%	0	0
Szymanski (12)	42%	1,147	**179**	25%	7	0	0%	0	0	8%	**0**	200	0%	0	0
TOILET (8)	100%	**1**	25	100%	4,135	**3**	75%	**61**	84	63%	496	**325**	100%	1,307	**621**
toilet (38)	100%	84	**50**	100%	684	**243**	97%	**115**	207	100%	**58**	90	97%	**395**	3,060
Tree (14)	100%	0	0	100%	0	0	100%	37	**9**	100%	**0**	1	93%	**1,051**	1,251

Table 3.2: Benchmark family specific information about commonly solved instances. Shown are the percentage of instances that are solved in both preprocessed and unpreprocessed form and the total time in CPU seconds taken to solve these instances within each family with and without preprocessing. Best times shown in **bold**.

Benchmark	*Skizzo*		*Quantor*		*Quaffle*		*Qube*		*SQBF*	
	no-pre	*pre*	*no-pre*	*pre*	*no-pre*	*pre*	*no-pre*	*pre*	*no-pre*	*pre*
Blocks	69%	**88%**	100%	100%	75%	**88%**	69%	69%	75%	**81%**
C	25%	**29%**	21%	**30%**	21%	**25%**	8%	**21%**	17%	**25%**
Chain	100%	100%	100%	100%	67%	**100%**	83%	**100%**	58%	**100%**
Connect	68%	**100%**	67%	**100%**	70%	**100%**	75%	**100%**	58%	**100%**
FlipFlop	100%	100%	100%	100%	100%	100%	100%	100%	80%	**100%**
K	89%	**91%**	63%	**83%**	35%	**36%**	37%	**42%**	33%	**35%**
S	27%	**37%**	25%	**31%**	2%	**8%**	4%	**8%**	2%	**8%**
Szymanski	42%	**75%**	25%	**50%**	0%	0%	8%	**25%**	**8%**	0%
toilet	100%	100%	100%	100%	97%	**100%**	100%	100%	97%	97%
Uclid	0%	**67%**	0%	0%	0%	0%	0%	0%	0%	0%

Table 3.3: Benchmark families where preprocessing changes the percentage of solved instances (within our 5,000 sec. time bound). The table shows the percentage of each families' instances that can be solved with and without preprocessing.

family by family breakdown of the common instances (instances that can be solved in both preprocessed and unpreprocessed form). Specifically, the table shows for each benchmark family and solver (a) the percentage of instances that are solvable in both preprocessed and unpreprocessed form, (b) the total time required by the solvable instances when no preprocessing is used, and (c) the total time required with preprocessing (i.e., solving as well as preprocessing time). Table 3.2 shows that the benefit of preprocessing varies among the benchmark families and, to a lesser extent, among the solvers. Nevertheless, the data demonstrates that among these benchmarks, preprocessing almost never causes a significant increase in the total time required to solve a set of instances. On the other hand, each solver has at least 2 benchmark families in which preprocessing yields more than an order of magnitude improvement in solving time. There are only two cases (Skizzo on Mutex, SQBF on the toilet benchmark) where preprocessing causes a slowdown that is as much as an order of magnitude (from 0 to 102 seconds and from 395 to 3,060 seconds).

Table 3.3 provides more information about the instances that were solvable only after preprocessing. In particular, it shows the percentage of each benchmark family that can be solved by each solver before and after preprocessing (for those families where this percentage changes). From this table we can see that for each solver there exist benchmark families where preprocessing increases the number of instances that can be solved. It is interesting to note that preprocessing improves different solvers on different families. That is, the effect of preprocessing is solver-specific. Nevertheless, preprocessing allows every solver to solve more instances. It can also be noted that the different solvers have distinct coverage, with or without preprocessing. That is, even when a solver is solving a larger percentage of a benchmark it can still be the case that it is failing to solve particular instances that are solved by another solver with a much lower success percentage on that benchmark. Preprocessing does not eliminate this variability.

Some instances are actually solved by the preprocessor itself. There are two bench-

mark families that are completely solved by preprocessing: FlipFlop and Connect. While the first family is rather easy to solve the second one is considered to be hard. In fact, $\approx$ 25% of the Connect benchmarks could not be solved by any QBF solver in the 2005 QBF evaluation [76]. Our preprocessor solves the complete benchmark family in less than 10 seconds. In addition, a few benchmarks from the hard S benchmark family can be solved by the preprocessor. Again these instances could not be solved by any of the QBF solvers we tested within our time bounds.

In total, the preprocessor can completely solve 18 instances that were unsolvable by any of the solvers we tested (in our time bounds). The Chain benchmark is another interesting case (its instances have 2 quantifier alternations $\exists\forall\exists$). The instances in this family are reduced to ordinary SAT instances by preprocessing. The preprocessor was able to eliminate all existential variables from the innermost quantifier block and consequently remove all universals by universal reduction. The resulting SAT instance is trivial to solve (it is smaller than the original QBF instance). In all of these cases the extended reasoning applied in the preprocessor exploits the structure of the instances very effectively. Note that the preprocessing cannot blow up the body of the QBF since it can only add binary clauses to the body. Thus, any time the preprocessor converts a QBF instance to a SAT instance, the SAT instance cannot be much larger that the original QBF.

There were only five cases where for a particular solver preprocessing changed a solvable instance to be unsolvable (Quaffle on one instance in the K benchmarks, SQBF on one instance in the Szymanski benchmarks, Skizzo on two instances in the Blocks benchmark and on one instance in the K benchmark). This is not apparent from Table 3.3 since both Quaffle and Skizzo can still solve more instances of the K and Blocks benchmarks respectively, with preprocessing than without. However, we can see that the percentage of solved instances for SQBF on the Szymanski benchmark falls to 0% after preprocessing. This simply represents the fact that SQBF can solve one instance of

Szymanski before preprocessing and none after. That is, we have found very few cases when preprocessing is detrimental.

In total, these results indicate that preprocessing is very effective for each of the tested solvers across almost all of the benchmark families.

3.6.3 Impact of Dynamic Application

In our second investigation of applying extended binary clause reasoning to QBF we constructed a QBF solver 2clsQ [96]. As described in Section 3.5, 2clsQ applies HypBinRes+UR at every node of its DPLL search, although it does not achieve closure as explained earlier). However, at the root node closure is achieved since the processing done before search is identical to that preformed in PreQuel. We tested 2clsQ along with the five other state of the art QBF solvers mentioned in the previous section. In addition, the benchmark settings are the same as the ones described earlier.

Table 3.4 shows the performance of 2clsQ and the other five solvers on the 468 problem instances we tested. The table is broken down by benchmark family as the structural properties of the families can be quite distinct. This structural distinctions are reflected in fact that the "best" solver for each family varies widely, where we measure best by the success rate of the solver on that families' instances breaking ties by CPU time consumed. By this measurement 2clsQ is best on 3 families, which is better than any other search based solver (Quaffle, Qube, and SQBF), but not as good as Skizzo which is best on 8 families.

Another comparison is to examine the average success rate over all benchmark families, shown in the final row of the table. A high average displays fairly robust performance across structurally distinct instances. On this measure 2clsQ is again superior to the other search based solvers with an average success rate of 58%, higher than any of the other search based solvers, but again not as good as Skizzo or Quantor. In terms of CPU time, the search based solvers are roughly comparable over their solvable instances, but both

Quantor and Skizzo are notably faster.

Our first results lead to the following conclusions. Binary clause reasoning improves search based solvers, but the non-search solver Quantor and the mixture of search and variable elimination employed in Skizzo often have superior performance. The superior performance of Skizzo indicates that mixing search and variable elimination (as done by Skizzo) is very effective. We also observe that both Quantor and Skizzo are still inferior to *some* search based solver on 43% of the families.

Furthermore, if we examine those cases where a solver is able to achieve a strictly higher success rate than any other solver (indicating that it can solve some instances not solvable by any of the other solvers), we see that 2clsQ achieves this on 2 families, Quaffle on zero, Qube on zero, SQBF on one, Quantor on one, and Skizzo on 6 families. Thus we conclude that binary clause reasoning as embodied in 2clsQ has some potential in increasing our ability to solve QBF (as do the techniques embedded in SQBF, Quantor, and Skizzo).

3.6.4 Detailed Performance Analysis of 2clsQ

In SAT it was observed that binary clause reasoning could be very beneficial even when done prior to search, in a preprocessing phase [7]. Hence, a natural question was to investigate the difference between dynamic and static (i.e., before search) application of binary clause reasoning.

Here we use our preprocessor to throw light on the effect of dynamic binary clause reasoning. In particular, we are interested in the question of how much of 2clsQ's benefits accrue from the dynamic application of binary clause reasoning. Is utilizing binary clause reasoning solely in a preprocessor sufficient, or is it also useful to use such reasoning dynamically during search? To answer this question we compare the performance of 2clsQ with the other solvers on *preprocessed* instances. By using the preprocessed instances, 2clsQ's only "advantage" over the other solvers is its dynamic application of binary clause

Benchmark Families (# instances)	*2clsQ*		*Quaffle*		*Qube*		*SQBF*		*Quantor*		*Skizzo*	
	Succ. %	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*
ADDER (16)	44%	5,267	13%	1	19%	72	13%	3	25%	25	**50%**	955
adder (16)	19%	0	44%	5	**44%**	0	38%	2,677	25%	30	44%	454
Blocks (16)	50%	46	75%	1,284	69%	1,774	75%	2,043	**100%**	308	69%	2,068
C (24)	21%	16	21%	5,356	8%	4	17%	4,741	21%	140	**25%**	1,070
Chain (12)	**100%**	0	67%	6,075	83%	4,990	58%	4,192	**100%**	0	100%	1
Connect (60)	**100%**	7	70%	254	75%	7,013	67%	0	67%	14	68%	802
Counter (24)	33%	4,319	38%	5	33%	2	38%	9	50%	217	**54%**	1,035
EV-Pursuer(38)	26%	2,836	26%	1,963	18%	4,401	**32%**	4,759	3%	74	29%	1,450
FlipFlop (10)	100%	4	**100%**	0	100%	1	80%	5,027	100%	3,260	100%	6
K (107)	35%	20,575	35%	18,451	37%	25,397	33%	5,563	64%	3,855	**88%**	2,081
Lut (5)	100%	19	**100%**	1	100%	3	100%	1,246	100%	3	100%	9
Mutex (7)	43%	22	29%	43	43%	64	43%	1	43%	0	**100%**	1
Qshifter (6)	33%	59	17%	0	33%	29	33%	1,108	100%	26	**100%**	8
S (52)	8%	9	2%	0	4%	401	2%	1	25%	910	**27%**	643
Szymanski (12)	**67%**	2,741	0%	0	8%	0	8%	1,203	25%	7	41%	1,147
TOILET (8)	75%	528	75%	61	63%	496	100%	1,308	100%	4,135	**100%**	1
toilet (38)	84%	47	97%	115	**100%**	58	97%	395	100%	684	100%	84
Tree (14)	100%	296	100%	37	**100%**	0	93%	1,051	**100%**	0	**100%**	0
Summary	58%	36,793	50%	33,653	52%	44,708	51%	35,326	64%	10,432	71%	11,817

Table 3.4: Percentage of each Benchmark family solved and time taken for solved instances in CPU seconds (5,000 sec. consumed by each unsolved instances is not counted). For each family the solver with highest success rate is show in bold, where ties are broken by time required to solve these instances. The summary line shows the average success rate over all benchmark families and the total time taken (on solved instances only).

Benchmark Families	*2clsQ*		*Quaffle*		*Qube*		*SQBF*		*Quantor*		*Skizzo*	
(# instances)	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*
ADDER (16)	44%	5,267	13%	1	19%	26	13%	1	25%	26	**50%**	792
adder (16)	19%	0	44%	4	**44%**	1	38%	1,546	25%	27	44%	550
Blocks (16)	50%	46	88%	1,025	69%	242	82%	3,434	**100%**	79	88%	11
C (24)	21%	16	25%	4,947	21%	683	25%	20	29%	5,189	**29**%	1,483
Chain (12)	100%	0	100%	0	100%	0	100%	0	100%	0	100%	0
Connect (60)	100%	7	100%	7	100%	7	100%	7	100%	7	100%	7
Counter (24)	33%	4,319	38%	5	33%	1	38%	20	50%	141	**54%**	731
EV-Pursuer(38)	26%	2,836	26%	1,961	18%	2,537	32%	4,508	5%	4,809	**39%**	5,753
FlipFlop (10)	100%	4	100%	4	100%	4	100%	4	100%	4	100%	4
K (107)	35%	20,575	36%	21,446	42%	30,606	35%	12,859	83%	6,898	**91**%	5,333
Lut (5)	100%	19	**100%**	1	100%	6	100%	66	100%	3	100%	9
Mutex (7)	43%	22	29%	49	43%	71	43%	6	43%	1	**100%**	100
Qshifter (6)	33%	59	17%	0	33%	29	33%	2,103	100%	29	**100%**	8
S (52)	8%	9	8%	9	10%	452	8%	9	31%	1,538	**37%**	1,538
Szymanski (12)	67%	2,741	0%	0	25%	199	0%	0	25%	109	**75%**	4,680
TOILET (8)	75%	528	75%	84	63%	325	100%	621	**100%**	3	**100%**	3
toilet (38)	84%	47	97%	221	100%	90	97%	3,061	100%	243	**100%**	50
Tree (14)	100%	296	100%	8	100%	1	93%	1,251	**100%**	0	**100%**	0
Summary	58%	36,793	55%	29,772	56%	35,281	57%	29,518	69%	19,108	81%	23,895

Table 3.5: Experiments from Table 3.2 repeated except that the other solvers are supplied with instances *preprocessed* by binary clause reasoning. Again unsolved instances consumed 5,000 sec., and for each family the solver with highest success rate is show in bold, where ties are broken by time required to solve these instances. The summary line shows the average success rate over all benchmark families and the total time taken (on solved instances only).

reasoning. Our results are shown in Table 3.5.

These results show that a significant part of the gains achieved from binary clause reasoning occurs statically prior to search. In terms of average success rate, 2clsQ still at 58% is now closer in performance to the other search based solvers all of which have gained, and still inferior to Quantor and Skizzo which have gained significantly from binary clause preprocessing. We also see that two of the families where 2clsQ was achieving superior performance, Chain and Connect, have been so reduced by preprocessing that all solvers now achieve similar performance on them. In fact, all instances of Connect are completely solved by preprocessing, and all instances of Chain are reduced to simple SAT problems by preprocessing.

Nevertheless, the results do show that dynamic binary clause reasoning improves the efficiency of search in QBF solvers. In particular, 2clsQ remains more effective than other purely search based solvers even when the effect of inference prior to search is factored out. The question now is whether or not these improvements to search are useful, given the effectiveness of variable elimination used by Quantor and Skizzo.

3.6.5 Filtering out instances best solved by variable elimination

To address this question we look more closely at how effective dynamic binary clause reasoning is on instances that are more suitably solved by search. In particular, it does not really matter much if (dynamic) binary clause reasoning improves the efficiency of solving by search instances that are more easily solved by variable elimination.

We examined those instances that would be solved very quickly by variable elimination, and to factor out the effect of binary clause reasoning prior to search we first preprocessed these instances. In particular, we found that a large number of instances (approximately 285) could be solved by Quantor after preprocessing in 25 seconds or less. In fact Quantor and Skizzo are obtaining a significant head start in their average success rate over the search base solvers from these “easy” instances.

After filtering out these instances a number of benchmark families were completely eliminated. That is, all of their instances were best suited for variable elimination after preprocessing. This left us with the benchmark families Adder, adder, C, Connect, Counter, EV-Pursue, K, Mutex, S, Toilet and Szymanski. However, even among these families several instances were eliminated as being easy. In this analysis we also eliminated all instances that could not be solved by any of the solvers as such instances are not useful when comparing solvers. In total we ended up with 72 instances remaining in 10 different benchmark families.

Table 3.6: Solver performance on "non-easy" preprocessed instances (i.e., instances that could not be solved in 25 seconds by Quantor after preprocessing. Uniquely solved instances shown in bold.

Family	*Instance*	*2clsQ*	*Quaffle*	*Qube*	*SQBF*	*Quantor*	*Skizzo*
ADDER	*Adder2-4-c*	0	-	26	-	-	111
	Adder2-6-c	7	-	-	-	-	-
	Adder2-8-s	-	-	-	-	-	12
	Adder2-8-c	16	-	-	-	-	-
	Adder2-10-s	-	-	-	-	-	437
	Adder2-10-c	3,812	-	-	-	-	-
	Adder2-12-s	-	-	-	-	-	230
	Adder2-12-c	1,432	-	-	-	-	-
adder	***adder-8-sat***	-	-	-	-	-	12
	adder-8-unsat	-	0	0	0	-	-
	adder-10-unsat	-	0	0	935	-	-
	adder-12-sat	-	-	-	-	-	314
	adder-12-unsat	-	0	0	191	-	-

Continued on next page

Table 3.6—continued from previous page

Family	*Instance*	*2clsQ*	*Quaffle*	*Qube*	*SQBF*	*Quantor*	*Skizzo*
	adder-14-unsat	-	0	0	419	-	-
	adder-16-unsat	0	2	0	-	-	-
C	***C6288-10-1-1-out***	-	-	-	-	-	1,436
	C880-10-1-1-inp	1	4	3	3	905	23
Counter	***counter-16***	-	-	-	-	-	721
	counter-r-8	-	-	-	-	60	1
	counter-re-8	-	-	-	-	79	3
EV-Pursue	*ev-pr-4x4-5-3-1-lg*	1	1	0	1	82	24
	ev-pr-4x4-5-3-1-s	-	-	-	-	-	6
	ev-pr-4x4-7-3-1-lg	17	3	16	1	-	1,469
	ev-pr-4x4-7-3-1-s	-	-	-	-	-	973
	ev-pr-4x4-9-3-1-lg	180	65	2,174	2	-	-
	ev-pr-4x4-9-3-1-s	-	-	-	-	-	1,679
	ev-pr-4x4-11-3-1-lg	390	990	-	3	-	-
	ev-pr-4x4-13-3-1-lg	-	-	-	4	-	-
	ev-pr-4x4-15-3-1-lg	-	-	-	5	-	-
	ev-pr-4x4-17-3-1-lg	-	-	-	7	-	-
	ev-pr-6x6-5-5-1-2-lg	4	5	2	24	-	2
	ev-pr-6x6-5-5-1-2-s	-	-	-	-	-	258
	ev-pr-6x6-7-5-1-2-lg	60	67	44	172	-	2
	ev-pr-6x6-7-5-1-2-s	-	-	-	-	-	462
	ev-pr-6x6-9-5-1-2-lg	823	784	-	3,708	-	235
	ev-pr-6x6-11-5-1-2-s	-	-	-	-	-	606
	ev-pr-8x8-5-7-1-2-lg	3	2	3	2	4,727	3

Continued on next page

Table 3.6—continued from previous page

Family	*Instance*	*2clsQ*	*Quaffle*	*Qube*	*SQBF*	*Quantor*	*Skizzo*
	ev-pr-8x8-7-7-1-2-lg	68	9	298	578	-	8
	ev-pr-8x8-9-7-1-2-lg	1,292	34	-	-	-	12
	ev-pr-8x8-11-7-1-2-lg	-	-	-	-	-	18
K	*k-branch-n-4*	141	-	93	1,190	-	12
	k-branch-n-8	-	-	-	-	-	40
	k-branch-p-4	1,858	389	20	147	32	0
	k-branch-p-8	-	-	-	-	-	0
	k-branch-p-12	-	-	-	-	-	52
	k-d4-n-8	-	-	-	-	-	0
	k-d4-n-12	-	-	-	-	-	0
	k-d4-n-16	-	-	-	-	-	0
	k-d4-n-20	-	-	-	-	-	1
	k-d4-n-21	-	-	-	-	-	1
	k-lin-n-20	1,493	-	1,370	-	66	74
	k-lin-n-21	1,511	-	1,593	-	82	87
	k-ph-n-16	287	261	4,729	4,334	198	198
	k-ph-n-20	2,636	2,204	-	-	1,790	1,806
	k-ph-n-21	4,254	3,668	-	-	2,950	2,977
	k-ph-p-12	-	-	-	-	1,689	-
Mutex	***mutex-16s***	-	-	-	-	-	1
	mutex-32s	-	-	-	-	-	9
	mutex-64s	-	-	-	-	-	22
	mutex-128s	-	-	-	-	-	70
S	*s499-d4-s*	-	-	-	-	228	107

Continued on next page

Table 3.6—continued from previous page							
Family	*Instance*	*2clsQ*	*Quaffle*	*Qube*	*SQBF*	*Quantor*	*Skizzo*
	s499-d8-s	-	-	-	-	-	1,878
	s641-d2-s	-	-	-	-	294	18
	s713-d2-s	-	-	-	-	448	29
	s820-d2-s	-	-	-	-	429	33
	s3330-d2-s	-	-	-	-	107	11
Szymanski	*szymanski-12-s*	221	-	-	-	105	1,183
	szymanski-14-s	677	-	-	-	-	954
	szymanski-16-s	1,780	-	-	-	-	1,992
	szymanski-18-s	-	-	-	-	-	373
Toilet	*toilet-a-10-01.16*	-	59	37	-	103	32
	toilet-c-10-01.16	-	72	46	655	110	9
Solved Instances		27	21	22	20	20	57
Total time on solved instances		23,238	11,884	10,628	13,019	14,534	21,252
Number of Uniquely solved Instances		3	0	0	3	1	22

In Table 3.6 we show the results of the solvers on these remaining *preprocessed* instances. In the table a '-' is used to indicate that the particular solver could not solve the instances within a 5,000 CPU second time bound. These results show that dynamic binary clause reasoning as performed in 2clsQ is effective on these harder instances. 2clsQ solves more of these instances than any other solver (27) except Skizzo. We also see that Quantor with 20 solved instances is less effective on these remaining instances than the improved search achieved by dynamic binary clause reasoning in 2clsQ. We also see that Skizzo, with its combination of search and variable elimination remains by far the most effective approach on these remaining instances with 57 instances solved.

Finally, if we look at the number of uniquely solved instances we see that both 2clsQ and SQBF can solve 3 instances not solvable by any other solver. These include instances that to the best our knowledge have never been solved before, e.g.,'Adder2-10-c' and 'Adder2-12-c'. These two solver embed techniques for improving search, and we see that these techniques can be useful for improving our ability to solve QBF. Skizzo can solve 20 instances not solvable by any other solver, so we see that combining variable elimination and search appears to be the most powerful current technique for solving QBF. However, the search employed by Skizzo does not include the innovations of SQBF or 2clsQ. Hence, our results point to at least one direction for building a QBF solver superior to any that currently exists.

It is also worth noting that 2clsQ and SQBF implement many of the techniques of Quaffle and Qube, so it is hardly surprising that all of the instances solved are also solved by some other search based solver. This does not detract from the techniques pioneered in these solvers, like clause and cube learning, which are essential for search based solvers. The uniquely solved instances speaks instead to the value of the new techniques utilized in other solvers: variable elimination in Quantor, binary clause reasoning in 2clsQ, SAT solving lookahead in SQBF (Chapter 4), and the mixture of variable elimination and search in Skizzo. The data indicates that these new techniques all have some value in improving our ability to solve QBF.

3.7 Related Work

In this section we review the similarities between our approach and the methods applied in existing QBF solvers. We conclude that *HypBinRes* has not been previously lifted to the QBF setting, although equality reduction and binary clause reasoning have been used in some state-of-the-art QBF solvers. Our experimental results support this, since our preprocessor aids the performance and reach of even the solvers that employ binary

clause reasoning and equality reduction.

Skizzo applies equality reduction as part of its symbolic reasoning phase [9]. [9] makes the claim that Skizzo's SHBR rule performs a symbolic version of hyper binary resolution. However, a close reading of the papers [9, 11, 10, 12] suggests that in fact the SHBR rule is a strictly weaker form of inference than *HypBinRes*. SHBR traverses the binary implication graph of the theory, where each binary clause (x, y) corresponds to an edge $\neg x \rightarrow y$ in the graph. It detects when there is a path from a literal l to its negation $\neg l$, and in this case, unit propagates $\neg l$. This process will not achieve *HypBinRes*. Consider the following example where *HypBinRes* is applied to the theory $\{(a, b, c, d), (x, \neg a), (x, \neg b), (x, \neg c)\}$. *HypBinRes* is able to infer the binary clause (x, d). Yet the binary implication graph does not contain any path from a literal to its negation, so Skizzo's method will not infer any new clauses. In fact, the process of searching the implication graph is well known to be equivalent to ordinary resolution over binary clauses [3]. On the other hand, *HypBinRes* can infer anything that SHBR is able to since it captures binary clause resolution as a special case. Therefore SHBR is strictly weaker than *HypBinRes*. This conclusion is also supported by our experimental results, which show, e.g., that our preprocessor is able to completely solve the Connect Benchmark where as Skizzo is only able to solve 68% of these instances.

The variable elimination algorithm of Quantor also bears some resemblance to hyper binary resolution, in that variables are eliminated by performing all resolutions involving that variable in order to remove it from the theory. General resolution among n-ary clauses is a stronger rule of inference than *HypBinRes*, but it is difficult to use as a preprocessing technique due to its time and space complexity (however see [34]).

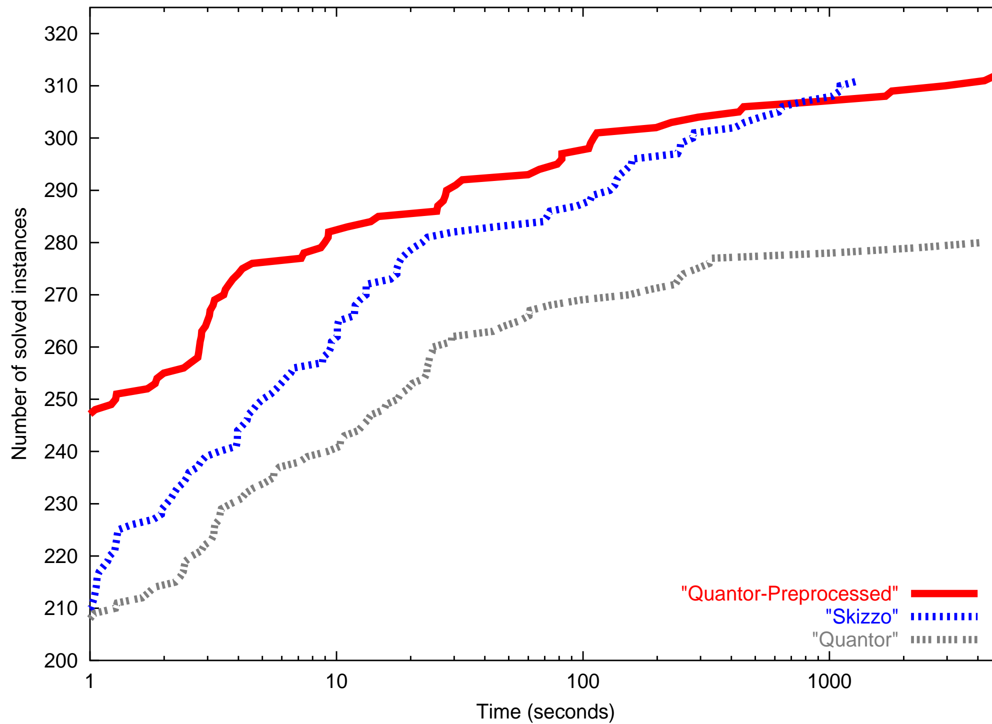


Figure 3.2: Logarithmic time scale comparison of Quantor and Skizzo on the original and Quantor on the preprocessed benchmarks. Shown is the time in seconds versus the number of instances solved.

3.8 Extensions of Preprocessing

Additional techniques for preprocessing remain to be investigated. Based on the data we have gathered with our preprocessor PreQuel, we can conclude that a very effective technique would be to run PreQuel followed by running Quantor for a short period of time (10-20 seconds). This technique is capable of solving a surprising number of instances. As shown in Figure 3.2 the combination of the preprocessor and Quantor is in fact able to solve more instances than Skizzo [9]. Hence, by simply employing hyper resolution and variable elimination it is possible to gain an advantage over such

sophisticated QBF solvers as Skizzo. Furthermore, this technique solves a number of instances that are particularly problematic for search based solvers. Figure 3.2 shows that this technique (the "Quantor-Preprocessed" line) can solve approximately 285 instances within 10 seconds. Yet if we continue to run Quantor for another 5000 seconds very few additional problems are solved (about 25 more instances). We have also found that search based solvers can solve a larger number of these "left-over" instances than Quantor.

This suggests the strategy of first running the preprocessor, then running Quantor, and then a search based solver if Quantor is unable to solve the instance quickly. As a matter of a fact this approach was the basis of the solver systems we entered in the the QBF competition 2006 [49]. This combined approach was able to outperform all other state-of-the-art QBF solvers, and the three systems we entered in the competition place first, second and third in the competition. The first place was achieved by the described preprocessing combination of first running PreQuel then Quantor and 2clsQ as a search based solver. Second place was achieved by first running Prequel followed by running Quantor without a time limit. And third place was achieved by the PreQuel-Quantor preprocessing combination followed by our SQBF search based solver (see Chapter 4). In addition to the benchmark results presented here these competition results verify the impact of our techniques in an independent fashion.

Another interesting approach would be to investigate obtaining the partially eliminated theory from Quantor after it has run for a few seconds, and then seeing if it could be further preprocessed or fed directly into a search based solver. The Skizzo solver [9] attempts to mix variable elimination with search in a related way, but it does not employ the extended preprocessing reasoning we have suggested here.

3.9 Conclusion

Our main conclusion is that extended binary clause reasoning is effective for QBF. If used prior to search in a preprocessor it is able to speed up both search based and variable elimination based solvers. Our empirical results as well as the QBF competition results [49] also show that such reasoning can also be useful in a dynamic context, and that certain problem instances can be solved with such reasoning that do not seem to be otherwise solvable.

However, although our empirical results identify binary clause reasoning as being useful techniques for solving QBF, understanding more clearly how to best to combine this reasoning with other kinds of inference, especially variable elimination, remains an open question. We investigate this question more fully in Chapter 6. There we will see that it is possible to find ways of applying binary clause reasoning in a more focused manner that can cooperate with other kinds of inference.

Chapter 4

Using SAT in QBF

4.1 Introduction

In this chapter we develop an algorithm that makes extensive use of order-unconstraint SAT solving in an attempt to alleviate (but not completely remove) the constraint variable ordering required by QBF (see Chapter 2). Our exploits learning in a fundamental way but still requires only polynomial space for its correctness and retains the any-space character common to search algorithms with learning (Chapter 2).

The idea is to utilize a backtracking SAT solver in a backtracking QBF solver. Because both solvers are doing backtracking search we are able to develop techniques to integrate them very tightly. For example, both solvers search the same tree and share all of their datastructures, including using the same stack to store the current path.

The key innovation of our method lies in techniques for sharing information between the two solvers so that information computed during SAT solving can be used to improve QBF solving and vice versa. As we will explain the SAT solver is able to uncover information that would be very difficult or costly for a standard QBF solver to uncover. This information can then be used to speed up QBF solving. The result is a QBF solver

that is able to improve on current state of the art on a number of benchmark suites[1].

4.2 SQBF

As explained in Chapter 2 there is no escaping the fact that in QBF we have to ensure that both settings of each universal variable are solvable. The constraint on variable ordering imposed by the quantifier sequencing can also be a significant impediment to performance. In SAT, e.g., it is provable that an inflexible variable ordering can cause an exponential explosion in the size of the backtracking search tree. That is, there exist families of UNSAT problems for which **any** DPLL search tree in which each branch follows a fixed variable ordering is exponential in size, whereas a quasi-polynomially ($O(n^{\log n})$) sized DPLL search tree exists when a dynamic ordering is used [23, 8].

This observation (also bolstered by empirical observations of the tremendous impact variable ordering has on DPLL SAT search [73]), is the underlying motivation for our approach. In particular, consider a QBF formula $\vec{Q}.F$ in which the body F is UNSAT. If all of quantifier blocks have size 1, QBF-DPLL will be forced to follow a fixed static variable ordering in proving $\vec{Q}.F$ to be UNQSAT. On the other hand an order unrestricted SAT solver might be able to determine that F is UNSAT very quickly, which will immediately tell us that $\vec{Q}.F$ is UNQSAT.

The idea of testing the body of the formula, F, can be used recursively at every invocation of QBF-DPLL (see Algorithm 2 in Chapter 2) on the body of the QBF subformula that is to be solved in that recursion prior to a full QBF search. If the body $F|_T$ is UNSAT, we can backtrack immediately. If $F|_T$ is SAT, then we still do not know whether or not $\vec{Q}.F|_T$ is QSAT, so we have to continue recursively solving $\vec{Q}.F_T$ with our QBF solver. Note that this process is related to testing trival truth (see Chapter 2) at every node. However, while with trivial truth only the binary result (SAT or UNSAT)

[1] This statement was made in 2005—state of the art evolved since then, but the approach presented here remains to be competitive on several benchmark families (see e.g., [49]).

is utilized during QBF solving we exploit the information gathered from the SAT solver more extensively via (1) following the SAT solution in the QBF solver and (2) clause learning during SAT solving as we will discuss in the subsequent paragraphs.

Furthermore, if F_T is SAT the SAT solver will find a satisfying truth assignment for F. This truth assignment is a sensible candidate for the left-most path in a Q-model of F_T. So after we obtain the SAT solution we can follow this solution in the QBF solver during its first (left-most) descent. It can, however, be the case that the SAT solution is not in fact a feasible left-most path for the QBF solver. In particular, this truth assignment might not survive the stronger Q-propagation (unit-propagation plus universal reduction, see Chapter 2) performed by the QBF solver. For example, if $\vec{Q}.F = \forall a, b.\exists c.(a, c) \wedge (b, \neg c)$, then the SAT solver could return $\pi = \langle \neg a, b, c \rangle$ as SAT truth assignment for F. However, the QBF solver following this solution would first instantiate $\neg a$ which by Q-Propagation would reduce $\vec{Q}.F$ to $\forall b.()$, i.e., F would contain an empty clause.

Putting these pieces together we obtain the S-QBF algorithm given in Algorithm 4. The algorithm is a modification of QBF-DPLL presented in Chapter 2. S-QBF is first invoked with *Level* equal to 1, the empty trail T and the empty variable assignment $\pi = \{\}$. Its first task is to find a SAT solution (line 3-7). The SAT solver might discover a number of literals implied at higher levels. Literals implied at higher levels cause S-QBF to backtrack, assert those literals, and then proceed downwards again. The SAT solver might also discover literals implied at the current level. These literals are used to reduce the input formula $\vec{Q}.F$ (line 7) via Q-propagation: these literals are independent of any choices made by the SAT solver so their consequences need to be accounted for by the QBF solver. After Q-propagating these implied literals the SAT solver is called again to see if it can find a SAT solution in light of these added constraints on F.

Eventually, the SAT solver finds a SAT solution (π is returned containing this solution), or causes a backtrack to a higher level in the QBF solver. If a solution is found,

1: $\langle$bool *Result*, cube c, int *BTLevel*$\rangle$ **S-QBF**(*Level*, T, π)

2: Perform non-chronological solution backtracking as in QBF-DPLL lines 2-5.

3: **while** $\pi == \{\}$ **do** { No current SAT solution }

4: $\quad\langle\pi$, -, *BTLevel*$\rangle$ = **SAT**(*Level*, T)

5: $\quad$**if** *BTLevel* < *Level* **then** { **SAT** can cause **S-QBF** to backtrack }

6: $\quad\quad$**return** $\langle$*FAIL*, -, *BTLevel*$\rangle$

7: $\quad QProp\big(\vec{Q}.F|_T\big)$

8: **repeat** { First and subsequent invocations of **S-QBF** need to find new SAT solution }

9: $\quad conflict = QProp\big(\vec{Q}.F|_T\big)$

10: $\quad$Perform non-chronological conflict backtracking as in QBF-DPLL lines 8-11.

11: $\quad$Pick v from the first quantifier block and let $\ell = \pi(v)$

12: $\quad T' = T \cup \ell$ {Add ℓ to trail T}

13: $\quad\langle$*Result*, c, *BTLevel*$\rangle$ = **S-QBF**$\big(Level + 1, T', \pi\big)$

14: $\quad$**if** (*BTLevel* < *Level*) **then**

15: $\quad\quad$**return** $\langle$*Result*, c, *BTLevel*$\rangle$

16: $\quad\pi = \{\}$ {Subsequent invocations need to find new SAT Solution}

17: $\quad$**if** (*Result* == *SUCCEED*) **then**

18: $\quad\quad cube[l] = c$ {Store cube}

19: $\quad\quad T' = T \cup \neg\ell$ {Add $\neg\ell$ to trail T and solve recursively}

20: $\quad\quad\langle$*Result*, c, *BTLevel*$\rangle$ = **S-QBF**$\big(Level + 1, T', \pi\big)$

21: $\quad\quad$**if** (*BTLevel* < *Level*) **then**

22: $\quad\quad\quad$**return** $\langle$*Result*, c, *BTLevel*$\rangle$

23: $\quad\quad$**if** (*Result* == *SUCCEED*) **then**

24: $\quad\quad\quad newcube = cube[l] \cup c \setminus \{\neg l, l\}$ {Tailing existentials are removed}

25: $\quad\quad\quad u$ = deepest universal in newcube; *BTLevel* = Level of u was branched on

26: $\quad\quad\quad$**return** $\langle$*SUCCEED*, c, *BTLevel*$\rangle$ {Note if c is empty then $BTLevel = 0$}

27: **until** FALSE {only exit from this loop via a return to a higher level}

Algorithm 4: S-QBF

the QBF solver *heuristically* tries to follow this solution (in quantifier order) by choosing a value for v that agrees with π (line 11). The SAT solution π is passed down to the next recursion where it is followed as far as possible, either to a failure or a valid solution in the QBF context at line 2. Q-propagation might cause S-QBF to fail while following π even though π is a SAT solution. Note that Q-Propagation cannot be applied in the SAT solver since Q-Propagation is only valid when the variables are instantiated in quantifier order whereas the SAT solver is order unconstrained.

Any conflicts encountered will cause a backtrack which will return to line 20 or 13 of some invocation after which the next invocation will call the SAT solver again. Thus the SAT solver is being used to refute UNSAT subtrees, and more importantly to compute new conflict clauses that can (a) cause the QBF solver to backtrack and (b) discover that various literals are implied at previous levels of the search. All of this information, computed by the SAT solver, is sound for the QBF solver: UNSAT subtrees are UNQSAT, any new clause learned by the SAT solver is a valid new clause for the QBF solver, and if a literal ℓ is SAT implied at a previous level of the tree then ℓ is Q-SAT implied at that level as well.

It should be noted that the SAT solver can also make an S-QBF invocation backtrack from line 20, even though we know that the other side of the universal branched on in that invocation has already been successfully solved. This might seem strange, since at this point we already know that the current prefix (above the *Level* of this invocation) contains at least one satisfying assignment below it. Thus one might think that the SAT solver could never then conclude that the prefix is contradictory. However, although the prefix is not SAT contradictory, it could still be QBF contradictory.

For example, say that the prefix contains the literal a, the body F contains the clauses $(\neg a, \neg b, c, d)$, $(\neg a, \neg b, c, \neg d)$, $(\neg a, \neg b, \neg c, d)$, $(\neg a, \neg b, \neg c, \neg d)$, b is universal, $b <_q c$, and $b <_q d$. The QBF solver will be able to solve the setting $\neg b$ without difficulty, as this setting satisfies all of these clauses. However, when at line 20 the setting b is made these

four clauses become contradictory. Q-propagation cannot detect the contradiction so the SAT solver will be invoked in the next recursive S-QBF call. SAT will be able to learn the new clause $(\neg a, \neg b)$, which after universal reduction becomes $(\neg a)$. This will cause the QBF solver to backtrack all the way to the point where a was added to the prefix.

4.3 Integration of SAT and QBF

In our implementation of S-QBF we built our own SAT solver (utilizing all of the modern techniques like 1-UIP clause learning, watched literals, etc. [73]). In this way we were able to obtain a much tighter integration between the SAT solver and the QBF solver, e.g., sharing of datastructures.

Clause learning is the basic unit of communication between the two solvers. As pointed out above, learned clauses are not necessary for correctness, but they are very helpful for efficiency. In particular, both the QBF solver, via contradictions generated via Q-propagation, and the SAT solver via contradictions generated via unit propagation can learn clauses. Universal reduction is applied to these learned clauses to make them more powerful. All of these learned clauses arise from sequences of Q-resolution steps, thus as shown in [22] they are all logical consequences of the input QBF. That is, they do not alter the QSAT status of the input. This means that any clause learned by either solver can be used by both solvers to prune paths from the search space they explore.

This is useful as each solver is able to learn different kinds of clauses. In particular, since the SAT solver is order unrestricted it can learn powerful clauses via its VSIDS heuristic [73] which would never be learned by the order restricted QBF solver. These clauses can significantly prune the set of paths explored by the QBF solver. On the other hand the QBF solver is able to employ stronger Q-propagation and so it also can learn clauses that the SAT solver could never learn. These clauses allow the SAT solver to prune paths that are fine from the point of view of SAT but which are contradictory with

respect to QBF.

Another way that the SAT and QBF solvers are integrated involves techniques for finding "good" SAT solutions (if any exist) [41]. In particular, a good SAT solution is a solution that will allow the QBF solver to generate a good cube (at line 3) if the QBF solver is able to follow the SAT solution down to a leaf. Our technique here is to alter the SAT heuristic for choosing the next decision literal so as to minimize the number of clauses satisfied only by universal variables in the solution. In our implementation we try to branch on existentials that appear in clauses currently only satisfied by universals. Thus, this heuristic tries to ensure that as many clauses as possible are satisfied by existentials. This will result in a smaller cube being generated during solution analysis (also see Chapter 2).

Finally, unlike the rigid prescription of Algorithm 4, our implementation employs some additional heuristic flexibility in deciding when to invoke the SAT solver.

The most important difference is that on many problems the SAT solver will return a SAT solution that fails when we try to follow it using the stronger Q-propagation. This failure then invokes another call to SAT which returns another SAT solution which again fails as we follow it. This sequence of "SAT-ok", "QBF-bad" solutions returned by SAT can be quite long and time consuming. Hence, if this happens more than a certain number of times (5 in our implementation) we give up on SAT solving for this descent and instead try to find a solution using the QBF solver and Q-propagation. In most such cases Q-propagation is able to quickly descend to a leaf from which point we continue with S-QBF. Otherwise the Q-propagation descent learns a conflict, we backtrack, and again continue with S-QBF.

4.4 Formal Results

In this section we present a few formal results on the presented approach. The results shown here are mainly based on the formal properties of Algorithm 2 discussed in Chapter 2.

Theorem 5 *S-QBF is sound and complete.*

Proof: By Theorem 3 the underlying QBF-DPLL algorithm is sound and complete.

SAT in S-QBF only allows S-QBF to backtrack on failure, it does not affect success backtracking. Thus, *SUCCESS* returns continue to correctly prove QSAT. Furthermore, all operations performed by SAT during failure backtracking are sound Q-resolution steps ((e.g., [22], [33]), so S-QBF also preserves the property that it backtracks from the root with *false* only if its input is Q-UNSAT. That is, S-QBF retains QBF-DPLL's soundness property.

For the same reason S-QBF retains the systematic property of QBF-DPLL from Algorithm 2 and consequently S-QBF is also complete. ∎

In additon, we make the following observation:

Observation 1 *The SAT solver in S-QBF is semi-systematic. That is, it never revisits the same SAT solution, but it can revisit deadend nodes not leading to a SAT solution.*

Notice that we view the set of assignments visited by the SAT solver to be the prefix of the subtree it is exploring along with the assignments it makes in that subtree. Under this interpretation it is immediate that the same SAT solution is never revisited: the prefix of that SAT solution must be different since S-QBF never reinvokes the SAT solver on the same prefix (by the previous theorem).

It can be, however, that the SAT solver will revisit the same deadend nodes (from which it will eventually backtrack). For example, in the initial call S-QBF invokes the SAT solver with an empty prefix. This invocation can visit many deadend nodes before finally descending to a solution. Once S-QBF follows that solution down to a leaf, it

will start to backtrack from that solution trying to validate the alternate settings of the universals along that path. Each alternate setting of a universal will cause the SAT solver to be invoked again.

In its initial invocation the SAT solver might have explored this alternate setting of the universal and might have visited many deadend nodes with the alternate setting. Some of these nodes might be revisited when searching the new subtree below the flipped universal. Note that clause learning can remove some of this redundancy, but not all. The original SAT solver invocation explored with a different variable ordering, and so the clauses it learned there might not all be applicable when searching the new subtree (which has a prefix of assignments in a different order).

The first theorem is a basic correctness and completeness result, where as the observation shows that our method of invoking a SAT solver adds only a limited amount of redundancy to the search.

4.5 Empirical Results

4.5.1 Benchmark Settings

We compared an implementation of our approach with two state of the art search based QBF solvers—Quaffle [111] (version as of Feb. 2005) and Qube (release 1.3) [46]. We also ran experiments with the non search based solver Quantor [14] (version as of Jan 2004)[2] Like these solvers our implementation also utilizes techniques for detecting monotone literals, heuristics for guiding cube resolution, and some other standard improvements over the basic algorithm given in Algorithm 4.

We used the following benchmark families from QBFLib: Adder, FlipFlop, VonNeumann, Counter, Toilet c/g, Robots_D2, Term, Comp, Z4ml, S1169, S1196, S298 and

[2] Skizzo [9] was not yet available in 2005.

all instances provided by Pan and Rintanen ($\approx$ 350 instances). In addition, we used a benchmark family introduced in [83] called Game (120 instances).

We excluded the families Mutex, Szymanski and Tree since all of them can be trivially solved by simple preprocessing. Further details on these benchmark families are discussed in Chapter 3.

We also excluded all of the other families from QBFLib (2004), e.g., Jmc and Uclid, because only one or two of their instances could be solved by any of the search based solvers.

We exclude results on any instance that had one of the following properties: (1) the difference in solving time between all search based solvers is small (less than either 200 seconds or within 10% of the fastest time); or (2) no search based solver can solve it in under 5,000 seconds. The remaining results are shown in Table 4.2. All experiments were performed on a 2.4 GHz Pentium IV with 3GB of RAM.

A summary of these results is presented in Table 4.1. In this table we show the total time used by each solver for all instances in each benchmark family (among those instances shown in Table 4.2. The "Total" column show the sum of the time over all benchmarks. To obtain a time in the presence of failures we added a penalty of $5,000$ seconds per failure. (Thus the times should be used only for qualitative comparisons). In addition, the table shows the percentage of failed instances for each benchmark family and in total.

4.5.2 Discussion

Table 4.1 shows that our new approach improves the current state of the art in search based solvers at the time it was developed, in aggregate solving the most problems and taking the least time of any of the solvers. In these tests S-QBF is not always the fastest solver, but it does improve on Quaffle and Qube on 21 out of the 68 problems reported on in Table 4.2. In many of the other cases it is very competitive, being the worst

Solver	*Blocks*	*Chain*	*Comp*	*Game*	*K*	*Robots*	*Term*	*Toilet*	*Total*
S-QBF	0%	66%	25%	0%	37%	0%	0%	0%	22%
	$2,991s$	$10,493s$	$5,000s$	$1,345s$	$70,848s$	$959s$	$2,577s$	$672s$	$26h$
Qube	20%	0%	75%	57%	25%	0%	66%	50%	31%
	$10,305s$	$3,499s$	$16,030$	$39,723s$	$59,594s$	$2,373s$	$12,566s$	$11,057s$	$43h$
Quaffle	20%	33%	0%	71%	50%	0%	0%	25%	43%
	$5,709s$	$9,978s$	$69s$	$50,217s$	$96,251s$	$410s$	$299s$	$6,057s$	$47h$
S^{-}	0%	66%	50%	57%	43%	0%	0%	25%	40%
	$4,932s$	$10,439s$	$10,000s$	$42,548s$	$84,279s$	$2,400s$	$3,246s$	$9,486s$	$45h$

Table 4.1: Summary of results reported in Table 4.2. Shown are the percentage of failed runs and the CPU time used (for each benchmark family and in total).

solver of the three search based solvers on only 9 of the 68 problems. As noted above we experimented with many other benchmarks, but on these the solvers could not be effectively discriminated.

To obtain a more accurate assessment of the benefit provided specifically by our new techniques for using SAT (vs. differences in implementation and heuristics), we built a derivative of S-QBF. This derivative (denoted S^{-}) used the same code base, the same variable ordering heuristic, the same cube learning and clause learning techniques, etc. S^{-} is simply S-QBF without the SAT solver. This provided us with a much more accurate control against which to assess our new techniques.

The summary performance of S^{-}, shown in Table 4.1, demonstrates that although our base QBF solver is quite effective, our new techniques for using SAT yield clear performance advantages. Table 4.2 shows in more detail the time taken by the different solvers on individual problems (columns S^{-}, *S-QBF*, *Quaffle*, and *QuBE*).

It is also useful to examine the effect SAT has on the size of the QBF search tree. Columns *SAT-dec*, *Q-dec*, S^{-} *Q-dec* of Table 4.2 show the number of decisions made by

Problem Instance	*QSAT?*	*SAT-dec*	*Q-dec*	*S^- Q-dec*	*S^-*	*S-QBF*	*Quaffle*	*QuBE*	*Quantor*
blocks3i.5.3	0	37779	50482	439625	32.05	4.53	158.25	453.98	0.36
blocks3i.5.4	1	47300	62403	298121	11.85	3.12	11.08	4626.19	0.38
blocks4i.6.4	0	7367	6438	19931487	3116.49	0.95	**fail**	203.99	0.31
blocks4ii.6.3	0	6087	5685	6409879	1042.46	1.1	208.19	21.02	22.63
blocks4ii.7.2	0	1804960	1444039	2860315	729.34	2981.66	312.28	**fail**	43.23
chain16v.17	1	65519	131582	131582	439.97	493.32	129.3	71.14	0.04
chain19v.20	1	–	–	–	**fail**	**fail**	4849.32	1123.53	0.07
chain20v.21	1	–	–	–	**fail**	**fail**	**fail**	2304.390	0.08
comp_1_1.0_0_o	0	3401	755	–	**fail**	0.12	1.92	**fail**	0.02
comp_1_1.0_1_o	1	0	34	34	0	0	0	1030.88	0.04
comp_1_0.2_1_o	1	0	58	58	0.01	0.01	0	**fail**	0.03
comp_1_0.2_0_o	0	–	–	–	**fail**	**fail**	67.63	**fail**	0.05
game20_20_40_2	1	3855587	4425993	2754583	260.23	440.94	**fail**	98.26	0.08
game20_25_25_1	1	4517800	2213579	–	**fail**	309.46	**fail**	369.5	**fail**
game20_25_25_2	1	2109107	1168113	–	**fail**	125.29	**fail**	2874.96	**fail**
game20_25_25_3	1	920314	413170	2027831	326.64	40.06	**fail**	1150.51	**fail**
game20_25_25_4	1	3298510	1680483	–	**fail**	222.13	**fail**	1651.43	**fail**
game20_25_50_1	1	3298510	1680483	–	**fail**	221.74	**fail**	1657.63	**fail**
game50_25_25_1	1	2452664	954186	12368548	477.79	64.22	**fail**	1869.7	**fail**
game50_25_25_3	1	188743	66888	6182150	220.99	4.13	**fail**	**fail**	**fail**
game50_25_25_4	1	72203	34183	–	**fail**	1.63	**fail**	51.48	**fail**
game100_25_25_2	1	36165	24291	–	**fail**	0.73	**fail**	**fail**	9.26
game100_25_25_3	1	32923	16184	–	**fail**	0.63	4.06	**fail**	0.04
game150_25_25_1	0	0	21	21	0	0	0	**fail**	0.01
game150_25_25_2	1	208546	175239	–	**fail**	4.22	4.34	**fail**	0.01
game150_25_25_4	1	14604	13567	41798186	1262.76	0.3	208.79	**fail**	0.01
k_branch_p-5	1	–	–	–	**fail**	**fail**	**fail**	3854.78	**fail**
k_d4_p-6	0	5542611	55260801	2005	0.42	1689.13	**fail**	837.45	1.43
k_dum_n-6	1	1876929	1639193	1692680	221.21	122.79	**fail**	117.42	0.02
k_dum_n-8	1	–	–	–	**fail**	**fail**	**fail**	2916.89	0.06
k_dum_p-11	0	–	–	–	**fail**	**fail**	871.44	1014.83	5.32
k_grz_n-9	1	366963	294974	736851	117.68	22.32	3534.32	67.06	3.86
k_grz_n-12	1	1231288	1106900	2884937	3093.12	285.7	**fail**	250.53	10.3
k_grz_n-13	1	1420342	1277434	3339392	4046.65	353.39	**fail**	253.01	11.29
k_grz_n-16	1	5110635	4232820	–	**fail**	711.97	**fail**	1253.97	32.15
k_grz_n-17	1	6310863	5229135	–	**fail**	1396.91	**fail**	1321.97	20.7
k_grz_p-10	0	–	–	–	**fail**	**fail**	**fail**	164.81	6.78
k_grz_p-14	0	–	–	–	**fail**	**fail**	**fail**	1270.28	17.19
k_grz_p-16	0	–	–	–	**fail**	**fail**	2481.57	1694.67	27.73
k_grz_p-17	0	–	–	–	**fail**	**fail**	3107.51	1922.98	21.37
k_lin_n-7	1	1836874	900248	174011	404.32	194.34	169.26	49.75	454.34
k_lin_n-14	1	4503632	2422960	–	**fail**	4030.32	2525.31	1353.86	**fail**
k_lin_n-15	1	–	–		**fail**	**fail**	3008.53	2108.53	**fail**
k_path_n-5	1	3814468	3658630	3037899	473.3	493.5	**fail**	158.02	0
k_path_n-6	1	–	–	–	**fail**	**fail**	**fail**	1514.29	0.01
k_path_p-6	0	2895489	2490412	823834	101.87	406.71	270.42	30.26	0.01
k_ph_n-15	1	–	–	4072609	3731.09	**fail**	283.51	158.02	2962.78
k_poly_n-3	1	4702368	2945933	5078474	1445.27	426.24	**fail**	151.16	0
k_poly_n-4	1	–	–	–	**fail**	**fail**	**fail**	1651.2	0
k_poly_p-7	0	0	83	83	0	0	0	**fail**	0.01
k_poly_p-8	0	0	99	99	0	0	0	**fail**	0.02
k_poly_p-10	0	0	123	123	0	0	0	**fail**	0.04
k_poly_p-11	0	0	131	131	0.01	0.01	0	**fail**	0.03
k_poly_p-12	0	0	147	147	0.01	0.01	0	**fail**	0.03
k_poly_p-14	0	0	171	171	0.01	0.01	0	**fail**	0.03
k_poly_p-17	0	0	203	203	0.01	0.01	0	**fail**	0.03
k_t4p_n-2	1	2400994	2228055	1410656	645.73	709.56	**fail**	84.11	0.02
k_t4p_p-4	0	–	–	–	**fail**	**fail**	**fail**	194.57	0.1
robots1_5_2_72.7	1	21720	3002426	313292	44.14	221.7	19.64	1385.68	**fail**
robots1_5_2_42.7	0	29395	7713081	4458791	1519.08	672.14	288.06	565.01	**fail**
robots1_5_2_61.6	0	17992	4529115	4619291	836.47	268.29	99.34	424.87	**fail**
term1_1_0.2_0_i	0	2708395	2655162	2906302	3238.12	2555.78	296.52	**fail**	**fail**
term1_1_1.0_1_o	1	129	88	722	0.03	0.02	0.06	2566.76	0.07
term1_1_1.0_0_o	0	36105	6769	7276	7.86	18.65	3.11	**fail**	1.57
toilet6.1.11	0	54468	44831	108215	48.5	22.47	9.21	307.92	0.09
toilet7.1.13	0	347166	273852	1225940	3570.54	617.92	39.76	**fail**	1.14
toilet7.1.14	1	888	1097	712183	867.72	0.32	45.65	749.85	0.02
toilet10.1.20	1	57	264	–	**fail**	0.1	**fail**	**fail**	**fail**

Table 4.2: Detailed benchmark results achieved by S-QBF, S^-, and other QBF solvers

the SAT solver, the number of decisions made by the QBF solver (in S-QBF), and the number of decisions made by S$^-$ (where SAT is not used). In most cases we see that the SAT solver is able to significantly reduce the number of decisions the QBF solver needs to make (comparing columns *Q-dec* and *S$^-$ Q-dec*). In fact, in many cases the sum of the SAT and QBF decisions in S-QBF is less than the number of QBF decisions used by the pure QBF solver S$^-$.

QBF decisions are more expensive than SAT decisions as they require extra work (e.g., triggering of cubes, detecting monotone literals, detecting the empty theory). Hence reducing the number of QBF decisions has a strong impact on the run-time (e.g, in the *Blocks*, *Game*, and *Toilet* benchmarks). In our implementation SAT decisions are made 5 to 10 times faster than QBF decisions depending on the problem instance. This means that using SAT can yield improvements even when the sum of decisions in SAT and QBF is higher than the number of decision made by pure QBF (in S$^-$) (e.g., the *K* benchmarks).

The SAT solver can, however, sometimes be a waste of time. For example the *Chain* benchmarks contain Q-propagation implication chains under which a QBF solver will never encounter a failure. Thus it is pointless to use a SAT solver to detect failures, and we see that on *chain16v.17* S-QBF performs the same number of Q-decisions as S$^-$. S$^-$ fails on the two larger chain problems, even without the slow down of extraneous SAT solving. This is because the low-level efficiency of our solver is not as optimized as Qube or Quaffle. In some cases SAT solving can even be harmful, as following its solutions can be misleading. For example, on *k_d4_p-6* S-QBF makes many more QBF decisions than when SAT is not used (S$^-$). But in the vast majority of the cases SAT is more informative than misleading.

Quantor is another state of the art QBF solver, but it is not based on backtracking search. Instead Quantor utilizes a variable elimination scheme based on the original resolution procedure of Davis-Putnam [31] and an additional scheme of universal expansion

(see Chapter 2). Quantor's approach often superior on these benchmarks. However, its failure rate is 24% which is slightly higher than that achieved by S-QBF. Furthermore, while we expect a few more problems could be solved by S-QBF given more time, Quantor is exhausting addressable memory on most of its failures. Overall, space exponential algorithms have the disadvantage that space is a much less flexible resource than time.

Again, the question of whether space intensive algorithms like Quantor, Skizzo [9], or QMRES [79] will eventually be the best way to solve QBF remains open. However, we are more optimistic about search based methods as we already concluded in Chapter 2. In particular, the wide variance in the times achieved by search based solvers shows that there is a lot of room for improvements in heuristics. Several instances in the *Game* benchmark family illustrate this point. Some can be solved in only a few seconds by S-QBF but cause Quantor to exhaust available memory.

4.6 Conclusions

In this chapter we have presented an approach for integrating order unconstrained SAT solving within an order constrained QBF solver. By utilizing clause learning techniques, and the fact that a SAT learned clause is valid for QBF, we have been able to achieve a tight integration between the SAT solver and the QBF solver so that information computed in each part can be used to improve the performance of the other part.

A number of natural questions remain, most of which center around the issue of obtaining more information from the SAT solving computations. Our techniques mainly take advantage of failure information computed by the SAT solver, and we have shown that this can make a tremendous difference in performance. We have also found that the heuristic technique of guiding the SAT solver to find a "good cube" solution can have a large impact on performance. In general, however, there is considerable room for improvement in the whole area of heuristics for QBF as it is for instance shown in

Chapter 6, and an intriguing open question is whether or not useful heuristic information could be gathered during SAT solving.

Chapter 5

Dynamically Partitioning

5.1 Introduction

In this chapter we extend the idea of dynamic partitioning, originally utilized in #SAT solvers, to make it applicable in a QBF solver. #SAT is the problem of counting the number of models of a CNF formula, and the idea of dynamic partitioning for solving #SAT was first utilized in [58]. That work presented a DPLL based algorithm for #SAT which examined the remaining CNF theory at each node of the search tree. The algorithm tried to partition the remaining theory into disjoint components that shared no variables. The disjoint components could then be solved independently of each other, resulting in a significant improvement in run time. In particular, since the run time is worst case exponential in the number of variables, partitioning can move us from $O(2^n)$ to $kO(2^{n/k})$ if the problem can be broken into k equally sized partitions. Applying this recursively can potentially yield an exponential speed up. See [27, 6] for more detailed theoretical results characterizing the speedups that can be achieved from partitioning.

Here we apply dynamic partitioning to QBF. We first make the observation that a QBF theory can be partitioned into independent components as long as these components share no *existential* variables. That is, QBF components do not have to be completely

disjoint as is the case with #SAT, just so long as they are existentially disjoint.

We then show how clause learning in search based QBF solvers can be quite easily extended to deal with partitioning. Extending cube (solution) learning to the case where partitioning is used is considerable more complex, and is perhaps the key innovation of our work. We have implemented our ideas in a new QBF solver 2clsP. 2clsP is built on top of the 2clsQ [96] solver (see Chapter 3), which with the addition of some preprocessing techniques was the top scoring solver in the 2006 QBF competition [77]. We show empirically that these new ideas yield a significant improvement in 2clsQ's performance.

We also demonstrate that 2clsP offers performance that is superior to other search based solvers and in many cases superior to non-search based solvers like Quantor [14] and Skizzo [9]. These results underscore the potential that partitioning, when properly augmented with clause and cube learning, has for helping us improve current QBF solvers.

In the sequel we first present some necessary additional background not introduced in Chapter 2, setting the context for our methods. We then present the details of how clause and particularly cube (or solution) learning can be extended to partitioning. Then we provide empirical evidence of the effectiveness of our approach, and close with a discussion of future work and some conclusions.

5.1.1 Partitioning QBF

We begin this section with a discussion on the conditions under which a QBF can be partitioned into a conjunction of independent sub-formulas. First we recall the following two standard logical laws for quantifiers. Let Φ_1 and Φ_2 be propositional formulas.

1. If Φ_1 does not contain x then

$$\exists x.(\Phi_1 \wedge \Phi_2) \Leftrightarrow (\Phi_1 \wedge \exists x.\Phi_2) \text{ and } \forall x.(\Phi_1 \wedge \Phi_2) \Leftrightarrow (\Phi_1 \wedge \forall x.\Phi_2).$$

2. $\forall x.(\Phi_1 \wedge \Phi_2) \Leftrightarrow (\forall x.\Phi_1 \wedge \forall x.\Phi_2)$

Observation 2 *If F is a CNF formula such that $F = F_1 \wedge F_2$ where F_1 and F_2 are CNF and the clauses in F_1 and F_2 share no existential variables, then $\vec{Q}.F \Leftrightarrow \vec{Q}_1.F_1 \wedge \vec{Q}_2.F_2$, where $\vec{Q}_i$ is the subsequence of $\vec{Q}$ containing only the variables of F_i.*

To see that this is true we first rewrite $\vec{Q}.F$ as $\vec{Q}.(F_1 \wedge F_2)$, then we proceed to use the above logical laws to distribute the variables of $\vec{Q}$ to F_1 or F_2, starting with the innermost quantified variables of $\vec{Q}$. We can apply this observation multiple times to separate $\vec{Q}.F$ into a conjunction of k smaller QBFs.

For example,

$$\begin{aligned}
&\forall u_1 \exists e_1 \forall u_2 \exists e_2 e_3.\big((u_1, \neg e_1) \wedge (u_2, \neg e_2) \wedge (u_2, e_3)\big)\\
&\Leftrightarrow \forall u_1 \exists e_1 \forall u_2 \exists e_2.\big((u_1, \neg e_1) \wedge (u_2, \neg e_2) \wedge \exists e_3.(u_2, e_3)\big)\\
&\Leftrightarrow \forall u_1 \exists e_1 \forall u_2.\big((u_1, \neg e_1) \wedge \exists e_2.(u_2, \neg e_2) \wedge \exists e_3.(u_2, e_3)\big)\\
&\Leftrightarrow \forall u_1 \exists e_1.\big((u_1, \neg e_1) \wedge \forall u_2 \exists e_2.(u_2, \neg e_2) \wedge \forall u_2 \exists e_3.(u_2, e_3)\big)\\
&\Leftrightarrow \forall u_1.\big(\exists e_1.(u_1, \neg e_1) \wedge \forall u_2 \exists e_2.(u_2, \neg e_2) \wedge \forall u_2 \exists e_3.(u_2, e_3)\big)\\
&\Leftrightarrow \forall u_1 \exists e_1.(u_1, \neg e_1) \wedge \forall u_2 \exists e_2.(u_2, \neg e_2) \wedge \forall u_2 \exists e_3.(u_2, e_3)
\end{aligned}$$

5.1.2 Partitioning for a Search Based QBF Solver

Observation 2 immediately yields a partitioning search based QBF solver as shown in Algorithm 5.

We branch on variables respecting the order of quantification, just as in a standard search based QBF solver (see Chapter 2). However, after every variable has been instantiated (at this stage some propagation can also be preformed to further reduce the remaining theory) we check if the remaining theory can be broken up into existentially disjoint partitions (line 7). This can be accomplished in time linear in the size of the remaining theory with a simple depth-first search or a union-find algorithm. We then solve these partitions independently (line 10). Since the remaining theory is equivalent

```
1  QBF-Prt(Q⃗.F)
2  if F contains an [empty clause/is empty] then
3      return([FALSE/TRUE])
4  for ℓ ∈ {v, v̄} for some v ∈ F with outermost scope in Q⃗ do
5      success = TRUE
7      Partitions = Partition(Q⃗.F|ℓ)
8      foreach Q⃗i.Pi ∈ Partitions do
10         if QBF-Prt(Q⃗i.Pi) = FALSE then
11             success = FALSE
12     if [¬success/success] AND v is [universal/existential] then
13         return[FALSE/TRUE]
14 if v is [universal/existential] then
15     return[TRUE/FALSE]
```

Algorithm 5: DPLL for QBF with dynamic partitioning

to the conjunction of these partitions, they must all be true for the entire theory to be true. Hence, we can stop if any of these partitions is false.

Unfortunately, although partitioning is a good idea, our empirical investigations allowed us to conclude that this simple version of partitioning is completely ineffective in practice. Fundamental to the performance of search based QBF solvers are the techniques of clause and cube learning (see Chapter 2 and [111, 47, 33, 66]). Without these techniques a partitioning solver performs much worse than current search based solvers that employ learning. One of the key contributions in this chapter is to show how learning can be extended so that it can be applied in the context of partitioning.

5.1.3 Quantifier Trees

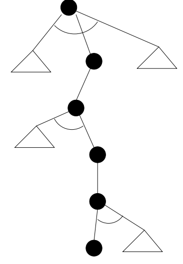
In [12] Benedetti uses the logical laws for quantifiers mentioned above to build a Quantifier Tree for a QBF. The quantifier tree specifies, among other things, a *static* decomposition of the QBF. That is, it specifies a decomposition that ignores the truth value assigned to each variable. Benedetti also points out that such trees can be used in a partitioning search based QBF solver similar to **QBF-Prt** presented above.

There are two main difference between this work and what we present here. First, as noted above the simple notion of of partitioning presented in **QBF-Prt** is ineffective without learning. As we will see adding learning to partitioning is a non-trivial new contribution of our work. Second, the partitioning algorithm presented in **QBF-Prt** employs *dynamic* partitioning. That is, the partitioning generated when we set $v =$ TRUE can be entirely different to the partitioning generated when $v =$ FALSE. In a quantifier tree the partitioning will be the same for both truth values. Since this difference compounds as we set more variables (and as we set variables via propagation) this means that the partitions generated dynamically can be considerably more refined than those specified in a static quantifier tree.[1]

5.2 Learning with Partitioning

Search based QBF solvers employ the powerful techniques of clause and cube (solution) learning as discussed in Chapter 2 and [66, 111, 47]. As pointed out in Chapter 2 these techniques are essential for obtaining good performance from a search based QBF solver. In this section we show how learning can be used with partitioning.

[1] In [6] it was shown that for #SAT on some instances dynamic partitioning can yield a super-polynomial speedup over *any* static decomposition. We suspect that a similar result holds for QBF, but this is not yet proven.



To facilitate the subsequent discussion the figure on the right shows a sample path in the **QBF-Prt** search tree. The black circles correspond to literals made true along the current path, arcs connecting branches indicate points where the theory was split into partitions, and the triangles correspond to the other partitions that were generated along this path. The partitions on the left of the current path have already been solved, while those on the right of the current path remain to be solved. We call the partitions that lie off of the current path *inactive*, and the partition currently being solved *active*.

5.2.1 Clause Learning

For the most part clause learning as described in Chapter 2 can be used without modification in a partitioning solver. For example, if the current path leads to a conflict a conflict clause can be learned and universal reduction applied—the conflict must be a subset of the literals set along the current path. This conflict can then be used to backtrack as least far enough to undo the conflict, as below this point no solution exists for the active partition.

Since the theory is the conjunction of its partitions, the status of the inactive partitions we backtrack past is irrelevant—falsifying the active partition is sufficient to falsify the entire theory. Note that backtracking further is also possible, e.g., backtracking to the 1st-UIP point [73]. The search will continue as before from that backtrack point. Similarly, the learned clauses can then be used in unit propagation as they normally would be in a non-partitioning solver.

The main subtleties with using clause learning with partitioning have already been addressed in [97] who showed how to use clause learning and partitioning in the context of solving #SAT. It is not difficult to show that their insights also hold for QBF. First, we are allowed to ignore the learned clauses when partitioning the theory since the learned clauses are entailed by the original theory. Second, it is sound to ignore existentials from

inactive partitions that might be forced by the learned clauses. Alternatively we can allow them to be forced: any conflict generated by them will still be a valid conflict.

5.2.2 Cube Learning

In order to extend the definition of cubes and learning to allow partitioning we must first develop a new formalization of cube learning. In comparison to the definition of cubes we provided in Chapter 2 the following redefinition of cubes makes more explicit some features of cubes that are needed to extend them to partitioning.

We first define the **restriction** of a clause c to a set of variables V to be the new clause c' formed by restricting c to the variables in V, i.e., removing from c all variables not mentioned in V. For example, $\textbf{\textit{restrict}}((x, \neg y, \neg z), \{x, y, w, t\}) = (x, \neg y)$, where the literal $\neg z$ has been removed since its variable z is not in the set $\{x, y, w, t\}$. We restrict a CNF formula F, $\textbf{\textit{restrict}}(F, V)$, by restricting each of its clauses. Note that if V contains all of the variables in c then $\textbf{\textit{restrict}}(c, V) = c$, and similarly $\textbf{\textit{restrict}}(F, V) = F$ if V contains all variables in F. We say that a QBF $\vec{Q}.F$ is **satisfied by the variables** V if the QBF $\vec{Q}.\textbf{\textit{restrict}}(F, V)$ is true.

We observe some facts about restriction and its relationship with reduction (Chapter 2).

Observation 3

1. *If $V \subseteq V'$, then $\textbf{\textit{restrict}}(F, V) \models \textbf{\textit{restrict}}(F, V')$.*

2. *If $\vec{Q}.F$ is satisfiable by any set of variables V, then it must also be true.*

3. *If $\ell \notin V$ then $\textbf{\textit{restrict}}(F, V \cup \{\ell\})|_\ell$ is equal to $\textbf{\textit{restrict}}(F|_\ell, V)$.*

4. *If $\ell \notin V$ then $\textbf{\textit{restrict}}(F, V) \models \textbf{\textit{restrict}}(F|_\ell, V)$.*

Proof: For item 1, every clause of $\textbf{\textit{restrict}}(F, V')$ is a superclause of a clause in $\textbf{\textit{restrict}}(F, V)$. For item 2, this follows from item 1 and Lemma 1 by taking V' to

be the superset of V that contains all variables of F. For item 3, this can be shown by considering what happens to every clause c of F under the stated sequence of reductions and restrictions (both restriction and reduction operate on a clause by clause basis). There are three cases to consider (a) $\ell \in c$, (b) $\neg\ell \in c$ and (c) all other clauses. For case (a) let $c = \{v_1, ..., v_j, m_1, ..., m_k, \ell\}$ where the v_i are in V and the m_i are not in V. Then $\boldsymbol{restrict}(c, V \cup \{\ell\}) = \{v_1, ..., v_j, \ell\}$ and reduction by ℓ causes c to vanish. Clearly c vanishes from $\boldsymbol{restrict}(F|_\ell, V)$ as well. For case (b) let $c = \{v_1, ..., v_j, m_1, ..., m_k, \neg\ell\}$ (v_i and m_k as before) then $(\boldsymbol{restrict}(c, V \cup \{\ell\}))|_\ell = \{v_1, ..., v_j\}$, similarly $\boldsymbol{restrict}(c|_l, V) = \{v_1, ..., v_j\}$ so c is the same in both sets. Finally for case (c) let $c = \{v_1, ..., v_j, m_1, ..., m_k\}$. Then $(\boldsymbol{restrict}(c, V \cup \{\ell\}))|_\ell = \{v_1, ..., v_j\} = \boldsymbol{restrict}(c|_\ell, V)$ and again c is the same in both sets. For item 4, we use the same three cases to show that $\boldsymbol{restrict}(F|_\ell, V)$ contains a subset of the clauses of $\boldsymbol{restrict}(F, V)$. For case (a) let $c = \{v_1, ..., v_j, m_1, ..., m_k, \ell\}$ ($v_i \in V$ and $m_i \notin V$ as before). Then $\boldsymbol{restrict}(c, V) = \{v_1, ..., v_j\}$ but $\boldsymbol{restrict}(c|_\ell, V)$ is the empty set of clauses. For case (b) let $c = \{v_1, ..., v_j, m_1, ..., m_k, \neg l\}$, then $\boldsymbol{restrict}(c, V) = \{v_1, ..., v_j\} = \boldsymbol{restrict}(c|_\ell, V)$. For case (c) let $c = v_1, ..., v_j, m_1, ..., m_k$, then $\boldsymbol{restrict}(c, V) = \{v_1, ..., v_j\} = \boldsymbol{restrict}(c|_\ell, V)$. Thus in cases (b) and (c) the same clause is in both sets, while in case (a) $\boldsymbol{restrict}(F, V)$ contains a clause not in $\boldsymbol{restrict}(F|_\ell, V)$. ■

Definition 7 *A **cube** for the formula $\vec{Q}.F$ is a set of literals ρ and a set of variables V such that (a) $\vec{Q}.F|_\rho$ is satisfied by the variables V, and (b) the variables of V are all downstream of the variables of ρ. We write* **cube**$[\rho, V, F]$ *to indicate that ρ and V is a cube for $\vec{Q}.F$.*[2]

In other words **cube**$[\rho, V, F]$ iff $\vec{Q}.\boldsymbol{restrict}(F|_\rho, V)$ is true, and V is downstream of ρ. This definition differs from the previous definition of a cube (see Chapter 2) mainly in its introduction of the set of downstream variables V. The previous definition required

[2]In the next section we will consider the case where F (the set of clauses) changes. However, the quantifier prefix, $\vec{Q}$, never changes so we can omit mentioning it in our notation.

that $\vec{Q}.F|_\rho$ be true irrespective of how the variables upstream of the deepest universal in ρ are set. That is, $\vec{Q}.F|_{\rho,\sigma}$ is true for any set of upstream literals σ.

Our new definition, on the other hand, requires that $\vec{Q}.F|_\rho$ can be made true by setting only variables downstream of ρ (i.e., the variables in V). That is, it requires that a Q-Model for $\vec{Q}.F|_\rho$ be constructed entirely from the variables in V.

If $\vec{Q}.F|_\rho$ can be satisfied by a set of downstream variables it is clearly true irrespective of how the variables upstream of the deepest universal in ρ are set. That is, the Q-model over the variables V for $\vec{Q}.F|_\rho$ will continue to be a Q-model for $\vec{Q}.F|_{\rho,s}$ for any additional set of upstream literals s.

Thus if $\mathbf{cube}[\rho, V, F]$ for some V by our new definition, it will also be the case that ρ is a cube for $\vec{Q}.F$ by our previous definition. Note however that it could be that ρ is a cube by our previous definition even though for no set V do we have $\mathbf{cube}[\rho, V, F]$.

Our new definition is motivated by a need to be explicit about the set of variables V that suffice to satisfy $\vec{Q}.F|_\rho$. Explicit mention of these variables is needed when we deal with cubes verifying partitions of the original formula, but not the entire original formula.

The following theorem shows that the standard technique for learning cubes in a search based solver (Chapter 2) are in fact sufficient to learn cubes that satisfy our new stronger definition.

Theorem 6

1. *If π is a set of literals that satisfies every clause of F, then* $\mathbf{cube}[\pi, \{\}, F]$.

2. *If* $\mathbf{cube}[\rho, V, F]$ *and v is downstream of ρ then* $\mathbf{cube}[\rho, V \cup \{v\}, F]$.

3. *If* $\mathbf{cube}[\rho, V, F]$ *and ℓ is existential and maximal in ρ, then* $\mathbf{cube}[\rho - \{\ell\}, V \cup \{\ell\}, F]$

4. *If* $\mathbf{cube}[\rho_1, V_1, F]$ *and* $\mathbf{cube}[\rho_2, V_2, F]$ *are cubes such that (1) there is a unique literal ℓ such that $\ell \in \rho_1$ and $\neg\ell \in \rho_2$, (2) this clashing literal is universal, and (3) ℓ is maximal in ρ_1 and $\neg\ell$ is maximal in ρ_2, then* $\mathbf{cube}[\rho_1 \cup \rho_2 - \{\ell, \neg\ell\}, V_1 \cup V_2 \cup \{\ell\}, F]$.

Proof: **For item 1**, we see that $\vec{Q}.F|_\pi$ is an empty set of clause, thus it is satisfiable by any set of variables. **For item 2**, this follows from Observation 3.1 since we are simply restricting by a larger set of variables $V \cup \{v\}$. **For item 3**, **cube**$[\rho, V, F]$ means that $\vec{Q}.\boldsymbol{restrict}(F|_\rho, V)$ is true and that V is downstream of ρ, the claim is that $\vec{Q}.\boldsymbol{restrict}(F|_{\rho\setminus\{\ell\}}, V \cup \{\ell\})$ is true and that $V \cup \{\ell\}$ is downstream of $\rho \setminus \{\ell\}$. Denote this formula by Γ. Since $\ell \in \rho$ it must be upstream of all of the variables in V. Hence, ℓ appears in the outermost quantifier block among the variables in Γ (the variables or Γ are only from the set $V \cup \{\ell\}$) and by definition Γ is true iff $\Gamma|_\ell$ or $\Gamma|_{\neg\ell}$ are true. In fact, $\Gamma|_\ell$ is true: $\Gamma|_\ell = \vec{Q}.\boldsymbol{restrict}(F|_{\rho\setminus\{\ell\}}, V \cup \{\ell\})|_\ell = \vec{Q}.\boldsymbol{restrict}(F|_\rho, V)$ by Observation 3.3. Since $\vec{Q}.\boldsymbol{restrict}(F|_\rho, V)$ is true, so is $\Gamma|_\ell$ and so is Γ as required. To see that $V \cup \{\ell\}$ is downstream of $\rho \setminus \{\ell\}$ we observe that ℓ is maximal in ρ, so it must be downstream of $\rho\setminus\{\ell\}$, while all of the V are already downstream of ρ. **For item 4**, let $\rho = \rho_1 \cup \rho_2 \setminus \{\ell, \neg\ell\}$ and let V be $V_1 \cup V_2$. We need to show that $\Gamma = \vec{Q}.\boldsymbol{restrict}(F|_\rho, V \cup \{\ell\})$ is true. Again we observe that ℓ appears in the outermost quantifier block among the variables in Γ. Thus Γ is true iff $\Gamma|_\ell$ and $\Gamma|_{\neg\ell}$ are both true. $\Gamma|_\ell = \vec{Q}.\boldsymbol{restrict}(F|_\rho, V \cup \{\ell\})|_\ell = \vec{Q}.\boldsymbol{restrict}(F|_{\rho\cup\{\ell\}}, V)$ (Observation 3.3). We have that $\vec{Q}.\boldsymbol{restrict}(F|_{\rho_1}, V_1)$ is true by assumption, that $\vec{Q}.\boldsymbol{restrict}(F|_{\rho_1}, V_1) \Rightarrow \vec{Q}.\boldsymbol{restrict}(F|_{\rho_1}, V)$ (since $V_1 \subseteq V$ and Observation 3.1), and that $\vec{Q}.\boldsymbol{restrict}(F|_{\rho_1}, V) \Rightarrow \vec{Q}.\boldsymbol{restrict}(F|_{\rho\cup\{\ell\}}, V)$ ($\rho \cup \{\ell\} = \rho_1 \cup (\rho_2 \setminus \{\neg\ell\})$, ℓ is maximal in $\rho \cup \{\ell\}$, ℓ is upstream of $V_1 \cup V_2$, hence all of the literals $\rho_2 \setminus \{\ell\}$ added to ρ_1 are also upstream of $V = V_1 \cup V_2$ and so not in V, and we can apply Observation 3.4 multiple times by adding the literals in $\rho_2 \setminus \{\neg\ell\}$ one by one to ρ_1 to obtain $F|_{\rho\cup\{\ell\}}$). The proof for $\Gamma|_{\neg\ell}$ is similar. ∎

Items 1, 3, and 4 of this theorem indicate that the standard technique of cube learning actually learns the newly defined cubes. Item 1 justifies finding a covering set for the clauses preformed at solution leaf nodes, item 3 justifies removing tailing existentials from a cube, and item 4 justifies combining the two cubes learnt when solving two sides of a universal variable. Standard QBF solvers do not however keep track of the set of

variables V. As we will see in a partitioning solver it becomes important to keep track of this set of variables.

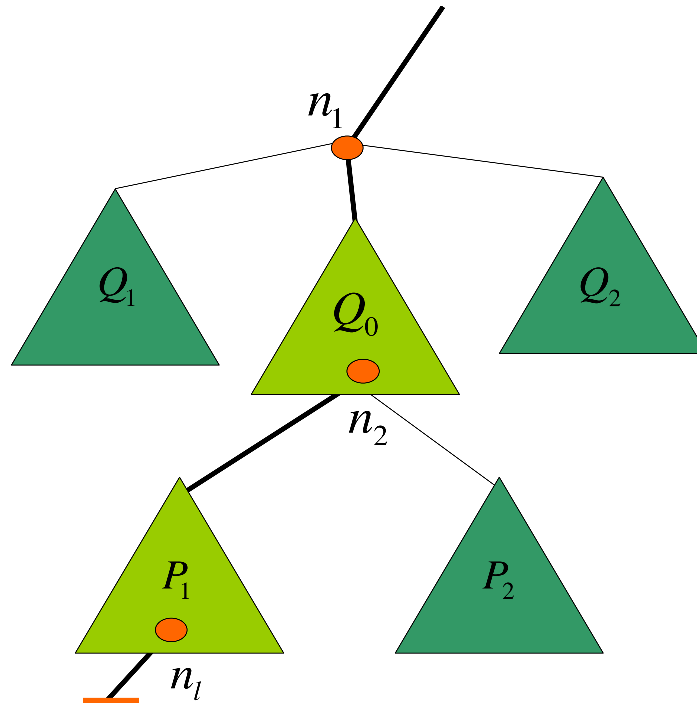


Figure 5.1: While search descends it breaks up the theory into several disjoint partitions. When the search arrives at n_1 and branches on it the theory splits up into the disjoint partitions Q_0, Q_1 and Q_2. By branching on n_2 the search continues with solving partition Q_0 and again breaks up the remaining theory into two partitions (P_1 and P_2). Branching on n_ℓ yields a solution leaf node within P_1.

Partial Cubes

With partitioning solution leaf nodes satisfy only some of the clauses of F (see the sample path diagram at the start of this section). In particular, the clauses in the inactive partitions need not be satisfied by the assignments along the current path. Consider the operation of **QBF-Prt** where each invocation is a node in its search tree. Say, as

illustrated in Figure 5.1, that the search descends along a particular path arriving at node n_1 where the remaining theory partitions into Q_0, Q_1 and Q_2. We then choose to solve Q_0 (at line 2) in the next recursive call, and continue to descend reaching a node n_2 where the theory partitions again into P_1 and P_2. Continuing with P_1 we finally reach a leaf node n_ℓ without further splitting P_1.

At n_ℓ some subset F_1 of the original clauses F have been made true by the literals set along the path to n_ℓ, and we can use item 1 of Theorem 6 to select a subset of these literals sufficient to form a cube for F_1: $\mathbf{cube}[\pi, \{\}, F_1]$. Note that in general this is not a cube for the original formula. In particular, we have not considered the clauses in the inactive partitions Q_1, Q_2 and P_2—these clauses have not necessarily been satisfied by the current path: $\mathbf{cube}[\pi, \{\}, F_1]$ is a partial cube. However, among the clauses of F_1 are included all clauses in the original formula F that contain an existential variable of the active partition P_1. In particular, let e be existential contained in P_1, and let c be a clause containing e. c must have been made true by the path to n_l (and thus must be in F_1). If not then c must lie in an inactive partition (in example one of Q_1, Q_2, or P_2). But then that partition would share an existential variable with P_1, contradicting the fact that P_1 is a partition.

Now, we continue the search using this cube to backtrack to undo the most deeply assigned literal in π. Using item 2 of Theorem 6 we can add all variables we backtrack across into the variables of the cube. Once we reach the most deeply assigned literal in π, if that literal is existential, we use item 3 of Theorem 6 to construct a new cube and backtrack further to the next deepest literal in π. If it is a universal we solve the other side, combine the two cubes using item 4, and continue to backtrack further. At each node n we obtain a $\mathbf{cube}[\rho, V_1, F_1]$ such that F_1 includes all clauses containing existentials of the active partition P_1 (this includes all clauses of P_1) along with all other clauses made true along the path to n from and V_1 contains all variables instantiated below n.

With partitioning, however, we cannot backtrack past node n_2 where the active par-

tition P_1 was created—the remaining theory under n_2 is $P_1 \wedge P_2$ and we don't know yet if P_2 is true. Rather, when our search in the subtree solving P_1 finally produces a cube $[\rho_1, V_1, F_1]$ such that all of the literals of ρ_1 are true at or above n_2, we can backtrack to n_2 and then proceed to solve P_2. At this stage, V_1 will be precisely the set of variables in P_1.

If P_2 is true, the search in P_2's subtree will yield $\mathbf{cube}[\rho_2, V_2, F_2]$ such that F_2 includes all of the clauses containing existentials of P_2 (this includes P_2) and shares with F_1 all clauses made true along the path to n_2, while V_2 contains the variables of P_2. Now we want to combine these two cubes to learn a cube which will allow us to backtrack further within the subtree solving Q_0. We claim that $[\rho_1 \cup \rho_2, V_1 \cup V_2, F_1 \cup F_2]$ is the cube we want.

Theorem 7 *Given* $\mathbf{cube}[\rho_1, V_1, F_1]$ *and* $\mathbf{cube}[\rho_2, V_2, F_1]$ *such that (1)* $\rho_1 \cup \rho_2$ *is not contradictory (i.e.,* $\forall \ell \in \rho_1 \cup \rho_2 : \neg\ell \notin (\rho_1 \cup \rho_2)$*), (2) the variables in* $V_1 \cup V_2$ *are all downstream of the variables in* $\rho_1 \cup \rho_2$ *and (3)* V_1 *and* V_2 *share no existentials, then* $\mathbf{cube}[\rho_1 \cup \rho_2, V_1 \cup V_2, F_1 \cup F_2]$.

Proof: First we observe that by assumption $V_1 \cup V_2$ are downstream of $\rho_1 \cup \rho_2$. So we only need to prove that $\vec{Q}.\boldsymbol{restrict}(F_1 \cup F_2|_{\rho_1 \cup \rho_2}, V_1 \cup V_2)$ is true. For two QBF S_1 and S_2 we write $S_1 \Leftarrow S_2$ if S_2 true implies S_1 true.

$$\begin{aligned}
& \vec{Q}.\boldsymbol{restrict}((F_1 \cup F_2)|_{\rho_1 \cup \rho_2}, V_1 \cup V_2) \\
\Leftarrow\ & \vec{Q}.\boldsymbol{restrict}((F_1 \wedge F_2)|_{\rho_1 \cup \rho_2}, V_1 \cup V_2) \\
\Leftarrow\ & \vec{Q}.\boldsymbol{restrict}(F_1|_{\rho_1 \cup \rho_2}, V_1 \cup V_2) \wedge \boldsymbol{restrict}(F_2|_{\rho_1 \cup \rho_2}, V_1 \cup V_2) \\
\Leftarrow\ & \vec{Q}.\boldsymbol{restrict}(F_1|_{\rho_1 \cup \rho_2}, V_1) \wedge \boldsymbol{restrict}(F_2|_{\rho_1 \cup \rho_2}, V_2) \\
\Leftarrow\ & \vec{Q}.\boldsymbol{restrict}(F_1|_{\rho_1 \cup \rho_2}, V_1) \wedge \vec{Q}.\boldsymbol{restrict}(F_2|_{\rho_1 \cup \rho_2}, V_2) \\
\Leftarrow\ & \vec{Q}.\boldsymbol{restrict}(F_1|_{\rho_1}, V_1) \wedge \vec{Q}.\boldsymbol{restrict}(F_2|_{\rho_2}, V_2)
\end{aligned}$$

Line 1 might involve duplicating some clauses, but yields an equivalent formula. Line 2 is justified by the fact that both restriction and reduction are applied clause by clause.

Line 3 is justified by Observation 3.1: we are restricting the clauses to a smaller set so the formula becomes stronger. Line 4 is justified because V_1 and V_2 share no existential variables (by assumption) so the formula can be partitioned. And finally line 5 is justified by Observation 3.4: none of the literals in $\rho_1 \cup \rho_i$ appear in $V_1 \cup V_2$. Finally, the conjunction on the last line is true by assumption. ∎

This theorem says that once we obtain a cube for each partition P_1 and P_2 under the node n_2 we can form a cube that satisfies all of the clauses containing existentials of P_1 or P_2 (this includes all of the clauses in P_1 and P_2) in P_1 and P_2 (each $P_i \subseteq F_i$), as well as all of the clauses satisfied along the path to n_2. In other words, the new cube $\mathbf{cube}[\rho_1 \cup \rho_2, V_1 \cup V_2, F_1 \cup F_2]$ satisfies all of the clauses containing existentials of the partition Q_0 (this includes all clauses of Q_0) since all such clauses either are in F_1 (contain an existential of P_1), in F_2 (contain an existential of P_2) or were made true along the path to n_2. We can then utilize that cube to backtrack further within the subtree solving Q_0.

Note also that (1) all of the literals of ρ_1 and ρ_2 are contained in the path to n_2 thus $\rho_1 \cup \rho_2$ is not contradictory, (2) if v is the variable branched on at node n_2, then we have that all of the variables of V_i are downstream v and the literals in ρ_i are upstream of v thus $V_1 \cup V_2$ is downstream of $\rho_1 \cup \rho_2$, and (3) since P_1 and P_2 share no existentials neither do V_1 and V_2 since V_i is the set of variables in P_i.

These observations demonstrate that the conditions of Theorem 7 are satisfied during search when partial cubes from multiple partitions are to be combined.

Although we have explained the processing of partial cubes in the context of a particular example, it is not difficult to see that the same processing works for any search tree explored by **QBF-Prt**. By adding this processing to **QBF-Prt** we obtain a partitioning QBF solver that is able to correctly learn cubes, and perform solution backtracking. This allows the search to preform non-trivial backjumping, i.e., jumping back over instantiated universals without having to check the other value, when it is searching within a

partition.

5.2.3 Triggering Learnt cubes

Besides using cubes to perform solution backtracking during **QBF-Prt**, we also want to remember previously learnt cubes and use them to avoid solving sub-formulas that can already be verfied as being QSAT by a learnt cube. In particular, a $\mathbf{cube}[\rho, V, F']$ allows us to conclude that the residual formula $\vec{Q}.\boldsymbol{restrict}(F'|_\rho, V)$ is QSAT. This also means that for any subset of the clauses in F', say E, $\vec{Q}.\boldsymbol{restrict}(E|_\rho, V)$ is QSAT. Hence, any time the current sub-formula is implied by $\vec{Q}.\boldsymbol{restrict}(E|_\rho, V)$ for some learnt $\mathbf{cube}[\rho, V, F']$ with $E \subseteq F$ it can be declared QSAT without further search.

Recognizing when a previously learnt cube verifies the sub-formula currently being solved can be accomplished by detecting the following conditions during search:

1. All of the literals in ρ are true.

2. The existential variables of the current sub-formula are precisely the existential variables of V (thus none of these variables have been assigned)

These conditions can be detected fairly easily during search. In particular, we know the current set of true literals, and thus can we determine which of the learnt $\mathbf{cube}[\rho, V, F']$ have all of their literals ρ true. Similarly since the search preforms existential partitioning and knows the partition it is currently working it, we can test if the currently uninstantiated existential variables of the active partition are the same as the existential variables in V. Note that these three conditions do not require testing whether or not the clauses of the current sub-formula (these are the remaining unsatisfied clauses of the active partition) are all contained in F'. Hence, when storing a $\mathbf{cube}[\rho, V, F']$ for latter triggering we in fact need only to store $< \rho, V >$ and can ignore F'. This also means that when processing cubes during search, as described above, we can do all of the processing without keeping track of the clauses covered by each cube (i.e., F'). In particular, we do not need

to store that information for triggering, and since the clauses covered always includes the clauses in the subtree we are backtracking out of we do not need that information for backtracking either.

Theorem 8 *Let $< \rho, V >$ be a learnt cube, and let n be a node in the* ***QBF-Prt*** *search tree such that*

- *The path to n makes every literal in ρ true*
- *The set of (unassigned) existential variables in the sub-formula being solved at n is the same as the set of existential variables in V.*
- *The sub-formula being solved at n does not contain the empty clause.*

Then the sub-formula being solved at n is QSAT.

Proof: Since $< p, V >$ is a learnt cube, their is some set of clauses F' that it covers, i.e., such that $\vec{Q}.\boldsymbol{restrict}(F'|_{\rho}, V)$ is QSAT. From the discussion above we know that when $< \rho, V >$ was first computed during search F' must include clauses of the original QBF that contain an existential of V. Now consider the sub-formula being solved at n, and denote it by $\vec{Q}.E|_{\pi}$, where E is a subset of the original clauses (those in the active partition not yet made true) and π is the set of literals assigned along the path to n. Note that $\pi = \rho \cup \sigma$, i.e., π includes ρ. All of the clauses in E contain an existential of V else they would contain no existential (the subformula contains no other existentials) and would be reduced to the empty clause by universal reduction (the subformula does not contain the empty clause). Thus the set of clauses E is a subset of F' and thus $\vec{Q}.\boldsymbol{restrict}(F'|_{\rho}, V)$ QSAT implies that $\vec{Q}.\boldsymbol{restrict}(E|_{\rho}, V)$ is also QSAT. Since none of literals in π are in V (the variables of V are unassigned), and $\pi = \rho \cup \sigma$, we can apply Observation 3 item 4 multiple times to see that $\vec{Q}.\boldsymbol{restrict}(E|_{\rho\cup\sigma}, V) = \vec{Q}.\boldsymbol{restrict}(E|_{\pi}, V)$ is QSAT, and then by Observation 3 item 2 we have that $\vec{Q}.E|_{\pi}$, i.e., the subformula below n, is QSAT.
■

The final item needed for triggering cubes within search is that every solved subtree must return a cube that can be used in the ongoing cube processing preformed by search (i.e., the processing described above). From that discussion it can be seen that the requirement for this cube be that the clauses it covers must include all clauses containing existentials of the active partition as well as all clauses made true by the current path. We know that the triggered cube $< \rho, V >$ covers all clauses containing an existential of V, so we need only to examine the set R of clauses made true along the path to n that do not contain any existential of V. We then add to ρ a set of literals from the current path, σ, sufficient to satisfy all of the clauses in R. The resulting $\mathbf{cube}[\rho \cup \sigma, V, R]$ can then be used just as if it had been computed by a search of the subtree below n rather than by a cache hit to a previously learnt cube.

In sum, we have shown in this section how cubes can be used with partitioning for non-chronological solution backtracking, and that they can also be stored and triggered to short-circuit the search of a subtree.

5.3 Soundness and Completeness

We conclude this section on the properties of our partitioning-based solver 2clsP with a soundness and completeness result. As mentioned earlier, 2clsP is an extension of 2clsQ (see Chapter 3). In addition to the techniques employed in 2clsQ it utilizes a partitioning-scheme similar to the one show in Algorithm 5. Furthermore, 2clsP incorporates the novel learning technique that was developed in the previous section.

Theorem 9 *The partitioning-based solver 2clsP with learning is sound and complete.*

Proof:

This result is based on Theorem 4 where it is shown that the underlying solver 2clsQ is sound and complete.

Soundness: We have to verify that conflict as well as solution learning as introduced in this chapter is a sound operation to determine the truth value of a node within the search tree of a partitioned-based solver.

Again the proof is by induction on the number of variables in the prefix $\vec{Q}$. The first base case is when there are no variables and F is the empty theory (no partitions), where 2clsP soundly returns with *true*. The second base case is when F contains the empty clause and consists of k partitions where 2clsP soundly returns with *false*.

Assume that if $\vec{Q}.F$ has n variables then 2clsP returns from the root node with the correct answer. Now consider a formula $\exists(\forall)v\vec{Q}.F$ that has $n+1$ variables and that splits into k existentially disjoint partitions at the root node. By the induction hypothesis the value returned by 2clsP on $\vec{Q}.F$ with n variables on each of the k partitions is sound. First assume that v is universal. If the returned value of one of the k partitions is *false* then 2clsP invoked on $\forall v\vec{Q}.F$ soundly returns *false*. If the return values of all k partitions are *true*, then 2clsP flips v's truth value and recurses with $\neg v$. If this recursive invocation returns *true* on all underlying l partitions then 2clsP soundly returns *true* on $\forall v\vec{Q}.F$. Otherwise, 2clsP soundly returns *false*.

Now assume that v is existential. If the returned value of one of the k partitions is *false* then 2clsP recurses with $\neg v$. If this recursive invocation returns *true* on all l partitions then 2clsP soundly returns *true* on $\exists v\vec{Q}.F$. Otherwise, 2clsP soundly returns *false*. If the return values of all k partitions are *true*, then 2clsP soundly returns *true* on $\exists v\vec{Q}.F$.

If 2clsP returns *false* on one of the partitions based on learned clauses we know that the result is sound (see Section 5.2.1). In addition, if 2clsP returns *true* on one of the partitions based on triggering a learned cube, we know by the results of the previous sections that we can soundly combine results from different partitions and the aggregated result is sound. Consequently, 2clsP invoked on $\exists(\forall)v\vec{Q}.F$ is sound.

Completeness: The proof of completeness is identical to the one for Theorem 4 except for the fact that now recursive calls can also result in splitting the theory into existentially disjoint sub-theories. However, by Observation 2 it is the case that a theory F can only be divided in a finite number of existentially disjoint partitions. Consequently, 2clsP invoked on a formula $\vec{Q}.F$ is complete. ∎

5.4 Implementation

We have implemented dynamic partitioning within the DPLL based QBF solver 2clsQ [95, 96]. In addition to the standard techniques employed in state of the art QBF solvers (e.g., solution analysis) 2clsQ also applies extensive binary clause reasoning at every search node (see Chapter 3, [95]). However, 2clsQ also utilizes dynamic equality reduction which we turned off due to the logical and implementational difficulties that arise when applying equality reduction and partitioning simultaneously.

Partitions are computed at each decision level by a simple depth-first search on the current theory. The complexity of this operation can be roughly stated as $O(|F| * vars_{\exists}(F))$ where $|F|$ denotes the size of the theory and $vars_{\exists}(F)$ the number of existentials in F.

We altered cube learning/solution backtracking as described in the previous section so that it could be used with dynamic partitioning. We also implemented a cube database and triggered cubes under the conditions described above.

The search requires a number of heuristic choices. Included in these choices are, deciding how to pick variables that are more likely to break the theory into partitions, deciding the order in which to solve detected partitions, and deciding when to turn off partition detection so as to minimize overhead.

A heuristic that selects a literal that satisfies the largest number of clauses can result in better partitioning since it decreases the overall connectivity. Computing articulation

points in the corresponding graphical representation of the theory and branching accordingly is an alternate strategy for increasing partitioning. However, in our experiments it seemed that the best strategy was to branch on a literal that has the highest potential to cause a conflict, irrespective of its ability to generate partitions.

Similarly, there exist many ways to sort the computed partitions to decide which partition to process next. We used the following strategy: for each partition we computed the number of binary clauses that contain an active existentially quantified variable. Then we computed the ratio of active existentials and binary clauses in each partition further weighted by the number of active universals in the partition. This weighted ratio tries to capture the degree of to which a partition is constrained. The lower the ratio the more constrainted are the existentials. The aim was to try to solve the most constrained partition first: if a partition fails we do not have to solve any more as the conjunction is immediately false.

In our experiments we observed that partitioning can slow down the search process due to its high overhead. Computing partitions at each decision level is an expensive operation. Furthermore, it is wasted work if the theory only consists of one partition.

In general, it is unlikely that a theory breaks into multiple partitions when the ratio between clauses and existentially quantified variables is rather high (e.g., 15). And in fact empirically it turned out to be the case that when partitioning was turned off on instances with a high clause/variable ratio the resulting performance was consistently improved.

Furthermore, it seems to be the case that a theory with a rather low clause/variable ratio (e.g., 3) appears to be unsuitable for dynamic partitioning as well. In this case, the theory is easily solved without partitioning, so again partitioning is not worth the overhead. Hence, when the input instance has a low or high clause/variable ratio we do not bother to try to detect partitions, and simple solve the theory as if it is a single partition.

5.5 Experimental Results

To evaluate the empirical effect of our new approach we considered all of the non-random benchmark instances from QBFLib (2005) [45] (508 instances in total). We discarded the instances from the benchmark families von Neumann and Z since these can all be solved very quickly by any state of the art QBF solver (less than 10 sec. for the entire suite of instances). We also discarded the instances in benchmark families Uclid, Jmc, and Jmc-squaring. None of these instances can be solved within a time bound of 5,000 seconds by any of the QBF solvers we tested. This left us with 465 instances from 18 different benchmark families. We tested all of these instances on a Pentium 4 3.60GHz CPU with 6GB of memory (this is a 32 bit processor so only 4GB of this memory is actually addressable by a single process). The time limit for each run of any solvers was set to 5,000 seconds.

5.5.1 2clsQ vs. 2clsP

We first compared 2clsP with 2clsQ. These two solvers are the most similar, with 2clsP only adding partitioning to the processing already performed by 2clsQ (and subtracting equality reduction). Hence this comparison gives the most information on the effectiveness of partitioning taken in isolation. Table 5.1 shows the comparison between these two solvers. The table is broken down by benchmark family as the structural properties of the families can be quite distinct.

For each solver and benchmark the success rate and the time consumed by the solver on the successfully solved instances are displayed. Bold values indicate that the particular solver achieved the highest success rate on that families' instances, where ties are broken by CPU time consumed.

On this measure 2clsP is the best solver in 9 out of the 18 benchmark families. There exists only one benchmark family (*toilet*) where 2clsQ outperforms 2clsP. On the

Benchmark Families (# instances)	*2clsQ*		*2clsP*	
	Succ. %	*time*	*Succ. %*	*time*
ADDER (16)	44%	5,267	**56%**	8,346
adder (16)	19%	0	**38%**	1,374
Blocks (16)	50%	46	50%	46
C (24)	21%	16	**25%**	14
Connect (60)	100%	66	100%	66
Counter (24)	33%	4,319	**33%**	1,220
EV-Pursuer(38)	26%	2,836	**34%**	2,282
FlipFlop (10)	100%	4	100%	4
K (107)	35%	20,575	**36%**	20,039
Lut (5)	100%	19	100%	19
Mutex (7)	43%	22	43%	22
Qshifter (6)	33%	59	**67%**	1,924
S (52)	8%	9	**15%**	3,405
Szymanski (12)	67%	2,741	67%	2,741
TOILET (8)	75%	528	75%	528
toilet (38)	**84%**	47	84%	531
Tree (14)	100%	296	**100%**	0
Summary	58%	36,791	63%	42,502

Table 5.1: Results achieved by 2clsQ and 2clsP on all tested benchmark families. Instances were timed out after 5,000 sec., and for each family the solver with highest success rate is shown in bold, where ties are broken by time required to solve these instances. In addition, the results of 2clsP are **emboldened** whenever it outperforms 2clsQ. The summary line shows the average success rate over all benchmark families and the total time taken (on solved instances only).

8 remaining benchmark families 2clsP achieves the same performance as 2clsQ. On these benchmarks the clause/variable ratio was unfavorable for partitioning, so 2clsP operated without it on these families. That is, on these families 2clsP operates exactly the same as 2clsQ does. Normally, the clause/variable ratio stays fairly constant among the problems of the same benchmark family. However, in the case of the *toilet* benchmark the ratio varies across instances, so that some of the problems in this benchmark were solved by 2clsP using partitioning and others without. This also holds for other benchmarks (e.g., *Adder, S*).

The average success rate over all benchmark families is shown in the final row of the table. A high average displays fairly robust performance across structurally distinct instances. On this measure 2clsP is superior to 2clsQ solving 63% of all instances on average compared to 58%.

When partitioning is not effective the CPU time is lower with 2clsQ than with 2clsP which was expected. Computing partitions at every decision level is an expensive operation. In summary, these results demonstrate quite convincingly that our new technique offers robust improvements to 2clsQ.

5.5.2 2clsP vs. Other solvers

We also compared our new solver 2clsP to five other state of the art QBF solvers **Quaffle** [111] (version as of Feb. 2005), **Quantor** [14] (version as of 2004), **Qube** (release 1.3) [46], **Skizzo** [9] (release 0.82), **SQBF** [90].

Quaffle, Qube, and SQBF are based on search, whereas Quantor is based on variable elimination and SAT grounding. Skizzo uses a combination of variable elimination, SAT grounding, and search, and also applies a variety of other kinds of reasoning on the symbolic and the ground representations of the instances.

Table 5.2 shows the performance of 2clsP and all other search based solvers on the 465 problem instances we tested, broken down by benchmark family.

As in the previous table we display for each solver and benchmark the success rate and the time consumed by the solver on the successfully solved instances. Again, bold values indicate that the particular solver gained the highest success rate on that families' instances breaking ties by CPU time consumed.

On this measure 2clsP is the best solver on 7 out of the 18 benchmark families. Skizzo follows with 6, Quantor with 4, Qube with 4, and Quaffle with 1. SQBF is not the best performer on any benchmark family.

The average success rate over all benchmark families is shown in the final row of the table. A high average displays fairly robust performance across structurally distinct instances. On this measure 2clsP is superior to all other search based solvers with an average success rate of 63%. It is followed by Qube (−11%), SQBF (−12%) and Quaffle (−12%). However, both Skizzo (+8%) and Quantor (+1%) achieve a better average success rate. In terms of the total CPU time, 2clsP requires the highest amount of CPU time.

In total 2clsP is a very competitive QBF solver that achieves the best performance on more benchmark families than any other solver. In addition, its average success rate is close to the best achieved by any of the tested solvers. Although the new techniques employed in 2clsP are rather complex we see that they pay off in terms of performance gains.

5.5.3 State of the art solver

The results of the QBF competition 2006 [49] indicate that the "best" QBF solver would probably use a portfolio approach rather than any single solver. For example, our 2clsQ entry which won the 2006 competition first applied a hyper-binary preprocessor (PreQuel [92, 93]), then it ran the QBF solver Quantor for a fixed period of time. Finally if the problem was still not solved 2clsQ was invoked on output of PreQuel.

Given the results displayed in [92] a very promising strategy in the competition would

Benchmark Families (# instances)	*Skizzo*		*Quantor*		*2clsP*		*Quaffle*		*Qube*		*SQBF*	
	Succ. %	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*	*Succ. %*	*time*
ADDER (16)	50%	954	25%	24	**56%**	8,346	25%	1	13%	72	13%	3
adder (16)	44%	455	25%	29	38%	1,374	42%	5	**44%**	0	38%	2,678
Blocks (16)	56%	108	**100%**	308	50%	46	75%	1,284	69%	1774	75%	7,042
C (24)	25%	1,070	21%	140	**25%**	14	21%	5,356	8%	3	17%	4
Chain (12)	100%	1	**100%**	0	**100%**	0	67%	6,075	83%	4,990	58%	4,192
Connect (60)	68%	802	67%	14	**100%**	7	70%	253	75%	7,013	67%	0
Counter (24)	**54%**	1,036	50%	217	33%	1,220	38%	5	33%	2	38%	9
EVPursade (38)	29%	1,450	3%	73	**34%**	2,282	26%	1,962	18%	4,402	32%	4,759
FlipFlop (10)	100%	6	100%	3	100%	4	**100%**	0	100%	1	80%	5,027
K (107)	**88%**	1,972	63%	3,839	36%	20,039	35%	21,675	37%	21,801	33%	5,563
Lut (5)	100%	9	**100%**	3	100%	19	100%	1	**100%**	3	100%	1,247
Mutex (7)	100%	0	43%	0	43%	22	29%	43	43%	64	43%	1
Qshifter (6)	**100%**	8	100%	26	67%	1,924	17%	0	33%	29	33%	1,107
S (52)	**27%**	644	25%	910	15%	3,405	2%	0	4%	401	2%	0
Szymanski (12)	42%	1,147	25%	7	**67%**	2,741	0%	0	8%	0	0%	0
TOILET (8)	**100%**	1	100%	4,135	75%	528	75%	61	63%	496	100%	1,307
toilet (38)	100%	84	100%	684	84%	531	97%	115	**100%**	58	97%	395
Tree (14)	**100%**	0	**100%**	0	**100%**	0	100%	37	**100%**	0	93%	1,051
Summary	71%	9,747	64%	10,412	63%	42,502	51%	36,873	52%	41,109	51%	34,385

Table 5.2: Results achieved by 2clsP and five other state-of-the-art QBF solvers on all tested benchmark families. Unsolved instances were timed out after 5,000 sec., and for each family the solver with highest success rate is shown in bold, where ties are broken by time required to solve these instances. The summary line shows the average success rate over all benchmark families and the total time taken (on solved instances only).

be to apply PreQuel and a time-limited version of Quantor as before, and Skizzo as final solver. This observation is mainly due to the good standard performance of Skizzo and the positive impact of preprocessing on Skizzo [92]. It is not clear if the employment of Quantor in the context of Skizzo would be as beneficial as it is for a search-based solver but given the performance of Quantor it should not turn out to be a drawback either.

However, depending on the benchmark families in the competition, the results shown here indicate that 2clsP together with the initial two stage processing of PreQuel and Quantor would also be able to achieve a high ranking. This is due to the fact that 2clsP remains to be a competitive solver on several benchmark families even when Skizzo is supplied with a preprocessed problem instances (e.g., the *Adder* benchmark family).

5.6 Conclusions

We have shown how dynamic partitioning can be used to obtain significant improvements to a state of the art QBF solver, 2clsQ. The key to making dynamic partitioning work is finding a way to utilize clause and cube learning in conjunction with partitioning. In this chapter we have presented an approach for accomplishing this.

There is, however, much scope for further improvements. These include better heuristics for promoting the dynamic creation of partitions, and better heuristics for deciding when to and when not to partition. One possible way to improve the approach presented here is to make use of machine learning techniques as we present in Chapter 6. For instance, it is very beneficial to know in advance if the given problem instance is likely to break into independent sub-theories otherwise the overhead of partitioning does not pay off and as pointed out earlier might be even harmful with respect to the performance. Chapter 6 introduces a general learning technique that would also allow us to automatically predict if it is useful to apply dynamic partitioning on an (sub-)instance or not.

In addition, the theory behind partial cubes can probably be elaborated further, and perhaps used to obtain further algorithmic insights.

Chapter 6

Adaptive Search

6.1 Introduction

In this chapter we develop a novel framework for solving QBF that combines a search-based QBF solver with machine learning techniques [94]. We use statistical classification to predict optimal heuristics within a portfolio-, as well as in a dynamic, online-setting. Our experimental results show that it is possible to obtain significant gains in efficiency over existing solvers in both settings.

While a few preliminary approaches exist that apply machine learning methods to solve SAT, no such approaches have been reported yet for QBF. For SAT, [78, 106] describe a methodology that starts with a fixed set of pre-chosen solvers and uses learning to determine which solvers to use for given problem instances. Similarly, [56] describe an approach to choosing optimal solvers along with optimal parameter settings for a set of problem instances, by trying to predict their run-times. Common to these and similar approaches is that they make use of a fixed-, pre-determined set of solvers. Learning methods are used to make an optimal assignment *ahead of time*, as a pre-processing step.

The main motivation behind such portfolio-based approaches is that they can make use of already developed and already highly optimized machinery, since they merely need

to solve a simple assignment problem.

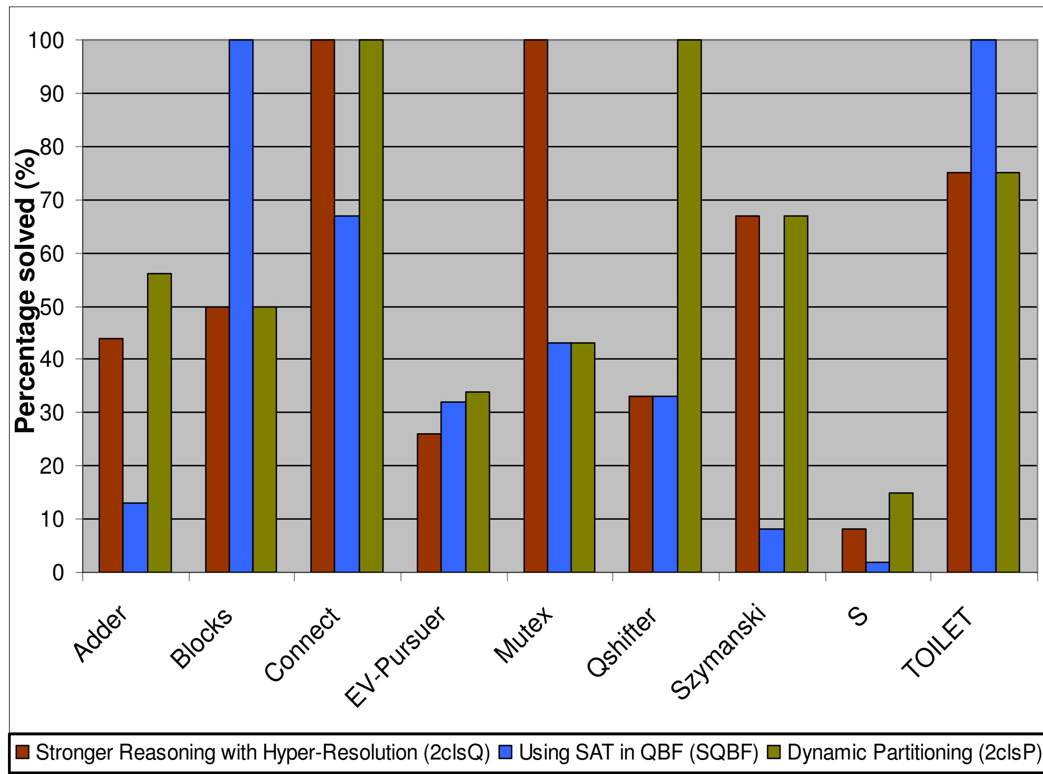


Figure 6.1: Shown is the performance of the QBF solvers 2clsQ, SQBF, and 2clsP (see previous chapters) on a range of benchmark families. For each solver we display on each benchmark family the percentage of solved instances within the corresponding benchmark family.

That such a static prediction would be also beneficial in the context of QBF is for example apparent from Figure 6.1. In this figure we show a performance comparison of the QBF solvers presented in the previous chapters. Shown is the percentage solved among several benchmark families and the performance of each solver on each benchmark family. As we can observe in Figure 6.1 the performance of the shown approaches is quite orthogonal. Clearly, the ability to automatically employ the best performing solver on each benchmark family would achieve in a better overall performance than any of those

solvers is capable of on its own.

However, a potential disadvantage of portfolio-based approaches is that they give up on the opportunity to obtain entirely novel methods that show a qualitatively different behavior than previous methods. Our dynamic approach therefore make use of a portfolio-based scheme on the level of sub-components and uses classification to choose online among a set of heuristics to solve sub-instances.

Among existing work on using learning to solve SAT the method that comes closest to our approach is probably [64]. The approach uses reinforcement learning to dynamically adjust the branching behavior of a solver, but does not make use of the properties of sub-instances that need to be solved in each step, and it failed to show an improvement over non-learning based approaches.

In this chapter, in contrast we use discriminative learning in order to dynamically *predict* optimal branching heuristics from the (sub-)instances a solver encounters at each step. Experimental results on a large corpus of example problems show the usefulness of our approach in terms of run-time as well as the ability to solve previously unsolved problem instances.

6.2 Dynamic Prediction

In this section we present our approach to integrate machine learning techniques within a search-based QBF solver. First, we briefly summarize how a search-based solver works in general. For a more detailed discussion the reader is referred back to Chapter 2. Second, we point out the main idea behind our approach and how it differs from the standard way of solving QBF.

6.2.1 An Adaptive Search-Based QBF Solver

As discussed in Chapter 2 search-based QBF solvers are based on a modification of the Davis-Putnam-Longman algorithm (DPLL, [30]). In general, DPLL works on the principle of assigning variables, simplifying the formula to account for that assignment and then recursively solving the simplified formula. The main difference to the original algorithm used to solve SAT is the fact that with QBF it is not only necessary to backtrack from a conflict but also from a solution in order to verify both settings of each universally quantified variable. A recursive version of this basic algorithm is displayed in Algorithm 2 in Chapter 2. Here we use the extended Algorithm 3 presented in Chapter 3 as the baseline algorithm.

As mentioned earlier Algorithm 2 shows how search relates to the earlier stated semantic definition of QBF. Modern backtracking QBF solvers employ conflict as well as solution learning to achieve a better performance (e.g., [111], [66], [33] and Chapter 2). Furthermore, several degrees of reasoning at each search node have been proposed. For instance, in addition to the standard closure under unit propagation, stronger inference rules like hyper-binary resolution were introduced [95] (Chapter 3). Other approaches employ a relaxation to SAT to perform a powerful look-ahead [90] (Chapter 4) or dynamic partitioning at every search node [91] (Chapter 5). Consequently, at each node the theory is not simply reduced by a single literal, but more extensive and powerful reasoning is applied.

This iterative application of reduction steps is likely to change the structural properties of the initial theory in an essential fashion. And since it is a well-known fact that for instance the performance of a heuristic varies essentially across different instances it is one of our purposes in this work to show that the performance of a heuristic can also change dynamically as we descend into the search tree.

1: ⟨bool *Result*, int *BTLevel*⟩ **QBF-2clsQ-Adaptive**($\vec{Q}.F$, *Level*)
2: **if** *F contains an [empty clause/is empty]* **then**
3: *Compute a new [clause/cube] and backtrack level BTLevel by [conflict/solution] analysis*
4: **return** ⟨*FAIL/SUCCEED*,*BTLevel*⟩
5: **Compute features of** F
6: **Predict best heuristic** h **among** $h_1, ..., h_n$
7: **Select variable** v **according to heuristic function** h
8: Let $\ell = v$ or $\neg v$
9: $\vec{Q}.F = \boldsymbol{restrict}(\vec{Q}.F, \ell)$ *reduced by HypBinRes+UR, equality reduction, universal reduction, and unit propagation*
10: ⟨*Result*, *BTLevel*⟩ = **QBF-2clsQ-Adaptive**$\big(\vec{Q}.F, Level + 1\big)$
11: **if** *BTLevel* < *Level* **then**
12: **return** ⟨*Result*, *BTLevel*⟩
13: **if** v *is [universal/existential]* **then**
14: Compute new [cube/clause] from the [cubes/clauses] learned from v and $\bar{v}$ by resolution
15: **return** ([*SUCCEED/FAIL*], *BTLevel*)

Algorithm 6: Adaptive search-based algorithm based on the 2clsQ algorithm (see Chapter 3). Lines 5 to 7 can cause the algorithm to automatically and dynamically adapt its variable ordering heuristic based on the structural properties of the underlying sub-problem.

Our changes to the recursive Algorithm 3 introduced in Chapter 3 are depicted in Algorithm 6 shown above (Lines 5 to 7). Before selecting a new variable we compute the properties of the current theory, which has been dynamically generated. Then, we use a previously trained classifier to determine which heuristic is suited best for this

theory. Note that this simple change is also applicable to other versions of search-based solvers we developed in this thesis (e.g., 2clsP in Chapter 5) but for simplicity we chose to display only the extension of 2clsQ. Since all employed heuristics are sound in the context of QBF (e.g., obey the variable ordering according to the given quantifier prefix) and the implementation presented here is based on Algorithm 3 the following result follows immediately:

Theorem 10 *The adaptive algorithm as shown in Algorithm 6 is sound and complete.*

Proof:

Both properties follow directly from Theorem 4 and the fact that all heuristics are sound.

■

In the following subsections we describe the different heuristics we designed and how we capture the structural properties of a theory.

6.2.2 Heuristics

In order to achieve a wide variety of solver characteristics we developed 10 different heuristics. All heuristics are crafted so that each of them tries to be orthogonal to the others. The next branching literal was mainly selected based on one or a combination of the VSIDS score [73] and cube score [111], the number of implied unit propagations and satisfied clauses. The VSIDS score is mainly based on recent conflicts during the search. In particular, the score of each literal participating in the conflict is increased. Analogously, the cube score [111] is computed. However, it is based on the recently encountered solutions. Stated differently, branching according to VSIDS tries to discover another conflict while the cube score guides the search towards the solution space. The other two measures behave in a similar dual fashion.

For instance, picking a literal with a high number of implied literals is likely to reduce and constrain the remaining theory maximally. In contrast, a literal that satisfies the highest number of clauses reduces the theory as well, but it does not necessarily reduce

the length of the remaining clauses in an essential way. We also use the inverse of these measures in several heuristics. As an example, Heuristic $H1$ employs a heuristic that is biased towards remaining within the conflict space when encountering a conflict and when discovering a solution it applies a heuristic that tries to stay within the solution space. In contrast, $H3$ and $H4$ are only biased towards conflicts or towards solution respectively. While the overall percentage of solved instances remains unchanged there are several benchmark families where those two heuristic choices result in a significant difference (see e.g., C, Toilet, Ev, and Texas).

We also employ other measures like the weighted sum between the number of literals forced by a literal and its corresponding VSIDS score in order to provoke a conflict even more drastically (see Heuristic $H6$).

The results achieved by Heuristic $H8$ are based on a heuristic that performs a look ahead for the existential variables and branches according the cube score for universally quantified variables. The look ahead consists of counting the number of clauses satisfied when branching on the positive/negative literal l. The count is based on the literals that are forced by the binary clauses which contain $\neg l$. The heuristic then choses the existential that satisfies the highest number of clauses.

Another example is pure literal detection which has a strong impact on the performance in QBF solving. For instance, the only difference between Heuristics $H1$ and $H2$ is that the latter does not detect purity.

There exist several other parameters that have a crucial impact on the performance of a heuristic, but we will not discuss all of them. Mainly because of the fact that it is not always straight-forward to relate a particular parameter to the performance of a heuristic.

However, it is worth mentioning that we also use a heuristic that simply employs a static variable ordering which performs surprisingly well on a subset of benchmarks (see Heuristic $H10$). In SAT it is provable that an inflexible variable ordering can cause an

exponential explosion in the size of the backtracking search tree. That is, there exist families of UNSAT problems for which any DPLL search tree where each branch follows a fixed variable ordering is exponential in size, whereas a quasi-polynomially sized DPLL search tree exists when a dynamic ordering is used [23].

However, the displayed results might indicate that the behaviour of a static variable ordering is more complex in the QBF setting. One reason for this could be that solution learning in QBF can potentially benefit from a static variable ordering. In addition, it would be interesting to explore how the use of a fixed variable ordering relates to the approach used in SAT introduced by [80]. There variable assignments are stored and after backtracking from a conflict the solver follows the stored assignments if possible. That approach achieves remarkable empirical results. Clearly, the static variable ordering we used here is related to this technique.

Finally note, that in contrast to the SAT case, the variables have to be assigned in quantifier order except for pure literals. Consequently, the branching possibilities are limited by the quantifier prefix in an essential fashion. However, it also appears to be the case that a poor decision with QBF has a much more dramatic impact than with SAT.

6.2.3 Feature Choice

In this section we point out the basic measures and give a detailed description on which features we used to characterize a QBF instance. While we summarize all extracted measures in a subsequent table we do not discuss each feature individually since this is initial work on feature selection for QBF and several of the displayed features are of question with respect to their importance to classification.

In total, we selected 62 features to capture the structure contained within an instance. All features are mainly based on the following basic properties of a QBF instance:

- # Variables (# Existentials, # Universals)

#	Feature description	#	Feature description
1	Number of active Variables	32	Ratio 10: #19 / #3
2	Number of Clauses	33	Ratio 11: #20 / #3
3	Number of Existentials	34	Ratio 12: #21 / #4
4	Number of Universals	35	Ratio 13: #22 / #4
5	Number of Quantifier Alternations	36	Ratio 14: #7 / #3
6	Number of Binary Clauses (BC)	37	Ratio 15: #7 / #4
7	Number of Ternary Clauses	38	Ratio 16: #8 / #3
8	Number of k-ary Clauses ($k > 3$)	39	Ratio 17: #8 / #4
9	Number of BC with Universals	40	Ratio 18: #6 / #1
10	Number of BC with only Existentials	41	Ratio 19: #6 / #3
11	Number of Pure Literals	42	Ratio 20: #9 / #4
12	Number of Pure Existential Literals	43	Ratio 21: #10 / #3
13	Number of Pure Universal Literals	44	Ratio 22: #4 / #6
14	Variable/Clause Ratio	45	Ratio 23: #3 / #6
15	Only Existentials/Clause Ratio	46	Ratio 24: #4 / #7
16	Only Universals/Clause Ratio	47	Ratio 25: #3 / #7
17	Positive Literals in BC: $\sum^{Vars} BC(pos(v))$	48	Ratio 26: #4 / #8
18	Negative Literals in BC: $\sum^{Vars} BC(neg(v))$	49	Ratio 27: #3 / #8
19	Positive Existentials in BC: $\sum^{\exists} BC(pos(e))$	50	Ratio 28: #1 / #8
20	Negative Existentials in BC: $\sum^{\exists} BC(neg(e))$	51	$\sum^{\exists}(BC(e) * \textit{Prefix-Level of } e)$
21	Positive Universals in BC: $\sum^{\forall} BC(pos(u))$	52	$\sum^{\forall}(BC(u) * \textit{Prefix-Level of } u)$
22	Negative Universals in BC: $\sum^{\forall} BC(neg(u))$	53	Current Prefix Level
23	Ratio 1: #3 / (#10 * #4)	54	Current Quantification
24	Ratio 2: #3 / (#7 * #4)	55	Added max. VSIDS over all Vars
25	Ratio 3: #3 / (#7 * #4 * *Max-Prefix Level*/2)	56	Added min. VSIDS over all Vars
26	Ratio 4: #3 / (#6 * #4)	57	Added max. VSIDS over all ∃
27	Ratio 5: #3 / (#6 * #4* *Max-Prefix Level*/2)	58	Added max. VSIDS over all ∀
28	Ratio 6: #3 / (#8 *#4)	59	Normalized max. VSIDS (#55/#1)
29	Ratio 7: #3 / (#8 * #4 * *Max-Prefix Level*/2)	60	Normalized min. VSIDS (#56/#1)
30	Ratio 8: #17 / #1	61	Added normalized max. Cube Score
31	Ratio 9: #18 / #1	62	Added normalized min. Cube Score

Table 6.1: Overview of the extracted features from a QBF instance. BC denotes binary clauses and $BC(v/l)$ denotes binary clauses of a variable/literal respectively. Features #59 to #62 are normalized by the number of total, active variables.

- # Clauses (# binary, # ternary, # k-ary)
- # Quantifier Alternations
- # Literal Appearances (# in binary, # in ternary, # in k-ary)
- VSIDS [73] and Cube Score [111]

Based on these fundamental attributes we additionally compute several ratios between combinations of these attributes, like the clause/variable ratio. Again, we also compute this ratio in the context of binary, ternary, and k-ary clauses. The features we computed are all summarized in Table 6.2.3. Note that we make extensive use of binary clauses in our features. This is mainly motivated by the underlying solver 2clsQ (see Chapter 3), which is based on binary clause reasoning. During its initialization process 2clsQ infers new binary clauses and consequently these clauses are also available in the feature computation of the portfolio-approach.

While we compute many features that are also applicable with SAT (see e.g., [78]) we also take into account properties that are specific to QBF. For instance, based on the the number of binary clauses that contain existentially quantified variables we compute the ratio of existentials and binary clauses further weighted by the number of universals (see for instance Feature #23). This weighted ratio tries to capture the degree of constrainedness of an instance: The lower the ratio the more constrained are the existentially quantified variables.

While the previous ratio focuses on the two different quantification types, we also take the number of quantifier alternations into account. For instance, we weighted the number of literal appearances by the number of the corresponding quantifier block. This is motivated by the fact that variables from inner quantifier blocks are often less constrained than variables from outer quantifier blocks (see for instance Features #52 and #53).

While all features are dynamically updated during search it is worthwhile to point out that Features #54 to #62 were especially chosen to capture the changes in the online

setting more carefully. For instance, the VSIDS score as well as the cube score can change quite drastically during search and different values might indicate distinct states of the search as well as different properties of the underlying sub-instances. In addition, Features #54 and #55 keep track of the actual depth of the search tree and the current type of quantification.

6.2.4 Classification

In the following we describe the approach that we use for predicting optimal heuristics for problem (sub-)instances. A key requirement for the predictor is run-time efficiency, because it can potentially be applied very often when solving a given problem instance. Furthermore, it is important to obtain well-calibrated outputs that reflect the confidences in the classification decisions over all possible heuristics. If calibrated correctly, these confidences allow us to determine at run-time, when it is worth switching the heuristic, and when not.

A simple classifier that satisfies both these requirements is multinomial logistic regression (MLR) (see e.g., [51]). While classification in general refers to the problem of learning a function that maps instances to class-labels, MLR solves this problem by building a model of the conditional probability $p(h|x)$ over (all possible) labels h given an instance x. Given such a probabilistic model, classification decisions can be made for new instances x, at 'test time', by choosing the heuristic h for which $p(h|x)$ is maximal.

In our case problem instances x are represented using the features described in the previous section. In other words, we represent a problem instance as a 62-dimensional real vector, and we define as the class for a problem instance x the heuristic that can solve x the fastest. (To obtain training data we applied each heuristic as the top-level decision on a set of benchmark datasets and recorded the run times resulting from using each heuristic. We then defined as the winning heuristic for a given problem instance simply the one whose runtime is smallest.)

We can think of the conditional probability $p(h|x)$ as a function that maps a 62-dimensional vector x to a discrete distribution over the 10 heuristics. A standard way of modeling this function is by defining

$$p(h|x) = \frac{\exp(w_h^{\mathrm{T}} x)}{\sum_{h'} \exp(w_{h'}^{\mathrm{T}} x)}, \tag{6.1}$$

where the exponential ensures positivity and the normalization that $\sum_h p(h|x) = 1$.

Given a set of training pairs $\{(x^i, h^i)\}$, that is, a set of instances x^i for which the optimal heuristic h^i is known, we can optimize this probabilistic model (or equivalently its logarithm) by maximizing:

$$\sum_i \log p(h^i|x^i) + \lambda \|w\|^2 \tag{6.2}$$

with respect to the parameters $w := (w_h)_{h=1,\ldots,n}$. The term $\lambda\|w\|^2$ penalizes large parameter values and helps avoid overfitting ([51]).

Any gradient based optimization method can be used for the maximization. Since the objective is convex, there are no local minima. (Note that, to apply the trained model to a new instance x it is sufficient to compute $w_h^{\mathrm{T}} x$ for each h and to pick the h which gives the largest value, as this h obviously also maximizes $p(h|x)$ as defined above).

The dataset we chose is certainly suboptimal because of its limited size, and to obtain even better performance additional training data could be gathered from the running system by collecting sub-instances. However, we obtained very good results already using this limited dataset, which shows that there is a sufficient degree of 'self-similarity' present in the problem-instances: The features of top-level problem instances have properties that are comparable to those of sub-instances and are good enough for generalization.

6.3 Experimental Evaluation

To evaluate the empirical effect of our new approach we considered all of the non-random benchmark instances from QBFLib 2005 and 2006 [75] (723 instances in total). To in-

crease the number of non-random instances we applied a version of static partitioning on all instances. A QBF can be divided in to disjoint sub-formulas as long as each of these sub-formulas do not share any existentially quantified variables (see, e.g. Chapter~̃efdynp).

This way we obtained a total of 1647 problem instances. However, among these instances there existed several duplicates (instances that fall into symmetric sub-theories) and instances that were solved by all approaches in 0 seconds. We discarded all of these cases and ended up with 897 instances across 27 different benchmark families. To obtain a larger dataset for training we used 800 additional random instances, that were not used for testing. On all test runs the CPU time limit was set to $5,000$ seconds. All runs were performed on 2.4GHz Pentiums with 4GB of memory which were provided by Sharcnet [98].

6.3.1 Variable Elimination vs. Search

To see whether QBF can gain from statistical approaches, we first experimented with a pure portfolio-approach, where the goal is to predict which method from a set of pre-defined methods is the best one for a given instance.

It is often easy to construct specialized methods that are *very good* at solving a *very small* set of problems, but it is much more difficult to develop a method that shows consistently good performance across a large set of problems. The advantage of using learning based approaches is that they allow us to *combine* highly specialized methods, since they can predict the right method for each (sub-)instance.

To illustrate that the features of instances can capture this orthogonality, we visualize a subset of the above mentioned data using principal components analysis (PCA) (see [51]). PCA is a standard method for reducing the dimensionality of data. It finds a lower-dimensional subspace of the original data-space, such that the projection of the data onto this subspace has maximal variance. Intuitively, this means that we find a

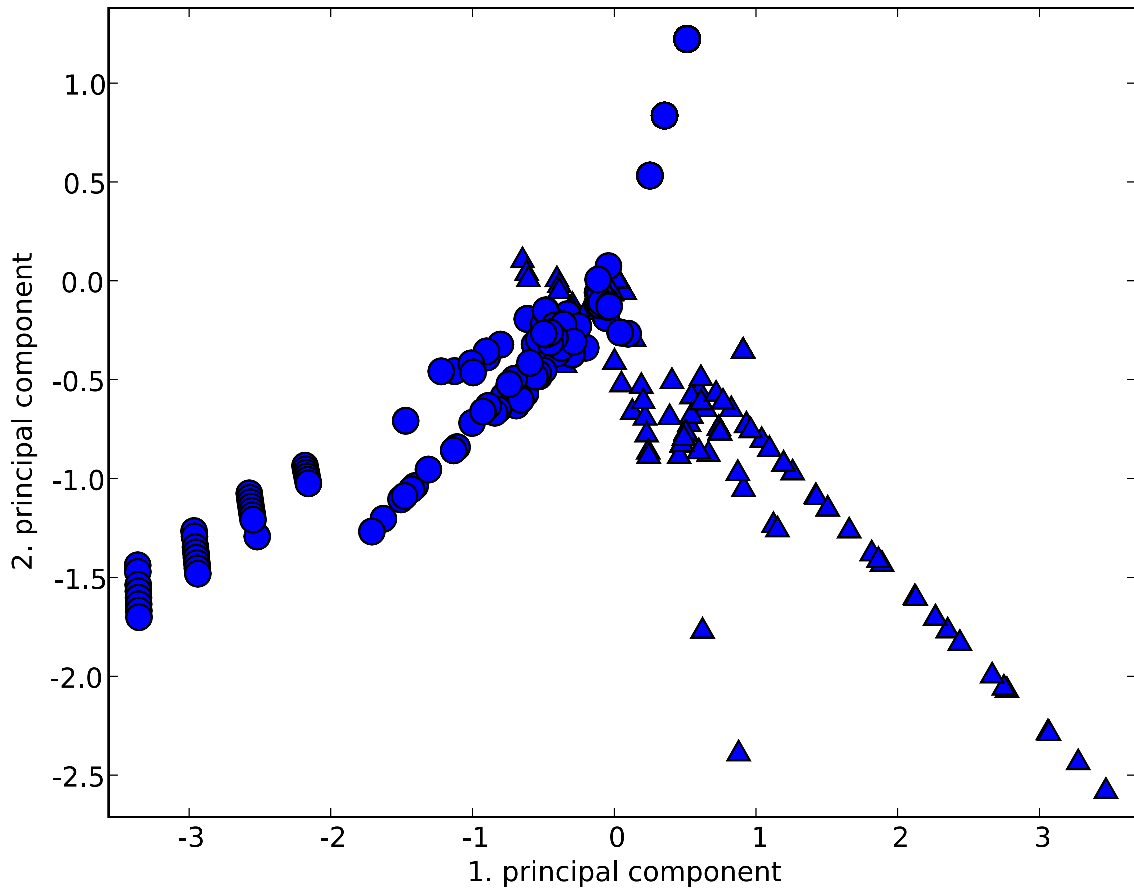


Figure 6.2: Low dimensional projection of problem instances that could be solved either by variable elimination (circles) or the search-based method (diamonds).

low-dimensional data representation that captures as much of the variability in the data as possible. For visualization we can simply choose a subspace whose dimensionality is less than or equal to 3.

Figure 6.2 shows the 2-dimensional PCA-representations of those instances that could be solved by exactly one of two solvers (but not by both simultaneously), using different symbols to indicate which of the two solvers was able to solve each instance. The two solvers are based on (i) variable elimination [14] and (ii) search as described earlier.

	PROBLEMS SOLVED	TIME SPENT (IN SEC)	INCREASE PROBLEMS SOLVED (IN %)
ONLY VARIABLE ELIMINATION	595	7918.58	**+12%**
ONLY SEARCH-BASED	423	31116.24	**+50%**
AUTOMATIC PREDICTION	666	27923.81	

Table 6.2: Performances: Variable elimination vs. search and the automatic prediction.

The plot shows that there is a quite clear separation of these instances already in two dimensions, and suggests that a linear classifier using all available features should indeed yield good performances in practice. As described above, we used logistic regression to predict which of the two methods to use in each case. We estimated λ using cross-validation on the training set. All reported final performances were computed on an independent test set. Each of the two methods has a time-out of $5,000$ seconds, after which we declare it as unable to solve the instance within a reasonable amount of time.

Table 6.2 displays the results on the test set and shows that choosing the best heuristic based on the data on an item-by-item basis yields much better performance than each of the two methods alone. In fact, the automatic prediction compared to variable elimination achieves a 12% improvement while the performance of the search based solver can be improved by more than 50%.

These results further underline the orthogonality of these two approaches. The two distinct characteristics were exploited also by the winning QBF solver of the 2006 QBF-competition [74]. The winning search-based solver used Quantor (a solver based on variable elimination [14]) as a time-limited preprocessor. Here we are able to uncover this difference and use learning to exploit it in an automated fashion.

	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10	PF	Dyn
CPU(s)	26,178	29,255	22,757	14,781	26,732	26,722	31,136	25,570	28,079	18,485	23,816	30,530
% sol	81%	74%	81%	81%	70%	77%	83%	79%	80%	79%	90%	**93%**

Table 6.3: Summary of the results achieved by the 10 different heuristics, the portfolio solver (PF), and the dynamic version (DYN) on instances across 20 benchmark families contained in the test set. For each approach the total CPU time required for the solved instances amongst all benchmark families is shown. For each approach the average percentage of solved instances amongst all solved instances per family is shown in the last row.

6.3.2 Predicting Heuristics

In this section we take the top-scoring search-based solver *2clsQ* [95] from the QBF competition [74]. We add 9 new heuristics to the original heuristic. Furthermore, we add the functionality to compute features on the fly for the current theory, and enable the solver to compute the linear classification decision in order to determine the most suitable heuristic for this theory.

We tested all heuristics on all instances and recorded their CPU times. This data was used in part to train the classifier off-line. Therefore, the data was split into a training set (628 instances, including random instances) and test set (576 instances across 20 benchmark families). All parameters were set on the training set (using cross-validation for λ).

We show a summary of the results for each heuristic ($H1$,..,$H10$) on each benchmark family in Table 6.3.2 contained in the test set. The heuristic $H1$ is the original heuristic employed in 2clsQ and consequently the performance displayed under $H1$ is a reference to the state of the art. In addition, we show the performance of the portfolio version which choses one solver for the problem in this context. Finally, we also include the results of the solver that dynamically alters its heuristic at different nodes of its search tree.

In the table we show in the first row the CPU time in seconds required on solved instances. As shown, all versions of the search-based solvers are roughly comparable in terms of CPU time over their solvable instances. We consider instances to be solvable, if they were solved by at least one approach. In the second row we display the average percentage of solvable instances among all solved instances per benchmark family and approach. On this measure approaches using a fixed heuristic for all instances vary between 70% and 83%. This variability in performance reflects the degree of variance induced by changes in the heuristic only. The table also shows that the portfolio approach – choosing a heuristic on a per-instance basis – is able to significantly outperform any approach employing a fixed heuristic. More importantly, the strategy to dynamically adjust the heuristic performs best among all approaches. In fact, it is able to outperform the best fixed heuristic by 10% on average. Furthermore, it is able to perform better than choosing the best heuristic on a per-instance basis.

In Table 3 and 4 we display more detailed results. Again we show the percentage of solved instances among all instances solved by any approach per benchmark family and approach as well as the CPU times for each approach and benchmark family. Also on these more detailed results the dynamic approach displays a robust performance (e.g., being the best method on 5 benchmark families). Table 3 also shows that our approach of dynamically adjusting the heuristic choice is able to solve instances not solvable by any other approach (see e.g., the K benchmark). Furthermore, Table 4 also shows that the overhead introduced by the dynamic feature extraction and classification is negligible.

In total, our empirical results show that the portfolio approach as well as dynamically adjusting the variable branching heuristics can be a very effective tool for solving QBF.

Bench	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10	Portfolio	Dynamic
Adder	83	33	83	78	33	83	77	56	50	**100**	94	67
BLOCKS	75	100	75	75	50	**100**	100	75	75	75	100	100
C	89	89	**100**	89	33	100	**100**	**100**	**100**	**100**	**100**	89
TOILET	67	67	83	33	67	33	66	**100**	67	67	83	83
adder	88	88	88	80	88	88	92	84	96	88	**100**	88
Counter	75	75	75	75	75	75	75	100	75	75	75	**100**
Eijk	100	100	100	100	100	100	100	100	100	100	100	100
EV	57	57	57	71	57	57	84	57	100	57	71	100
irst	100	100	100	100	100	100	100	100	**100**	100	100	100
K	95	95	95	90	95	90	95	90	90	86	95	**100**
Ken	**100**	**100**	**100**	**100**	**100**	50	**100**	**100**	**100**	50	**100**	**100**
Lut	67	67	67	67	67	67	66	67	**100**	33	67	67
Mutex	25	25	25	25	25	25	25	25	25	**100**	75	75
Nusmv	100	**100**	100	**100**	**100**	100	100	100	80	100	100	100
Qshifter	57	60	57	57	60	60	60	100	100	**100**	100	100
S	**100**	**100**	**100**	**100**	**100**	**100**	**100**	80	**100**	**100**	**100**	**100**
Sort	94	96	**100**	92	94	96	94	96	94	94	98	94
Szymanski	100	0	100	100	0	100	**100**	0	0	13	100	100
Texas	50	25	50	**100**	50	50	50	50	50	50	50	**100**
toilet	100	100	71	88	100	65	94	94	**100**	100	94	88
Summary	81	74	81	81	70	77	83	79	80	79	90	**93**

Table 6.4: Success rates achieved by the 10 different heuristics, the portfolio solver, and the dynamic version on instances across 20 benchmark families contained in the test set. For each benchmark and approach the percentage of solved instances among all solved instances per family is shown. For each family the solver with highest success rate is shown in bold, where ties are broken by time required to solve these instances. The summary line shows the average success rate over all benchmark families.

Bench	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10	PF	Dyn
Adder	2754	513	4041	387	246	5012	669	7366	6844	2412	2090	1121
BLOCKS	1	3077	3	0	1	161	2002	1	1	4	3077	4413
C	0	0	1	0	0	1	1	1	1	1	1	1
TOILET	450	424	55	116	579	206	749	832	1344	1972	1973	4159
adder	6234	8835	2790	938	9767	788	5332	2187	5652	2355	5332	4478
Counter	10	10	0	227	10	9	38	903	16	10	16	732
Eijk	255	244	1621	386	191	1782	89	20	4	5	89	255
EV	68	104	35	4423	183	66	3859	24	22	149	96	39
irst	4	4	2	8	4	2	1	1	1	1	1	1
K	7207	5794	6664	5709	5872	5409	9270	9274	10567	4306	5042	11410
Ken	0	0	0	0	0	0	0	0	0	0	0	0
Lut	0	0	207	4	0	302	3	4	89	0	0	0
Mutex	0	0	0	0	0	0	0	0	0	1	0	0
Nusmv	22	21	1149	21	21	936	1678	606	342	175	749	938
Qshifter	13	2785	13	13	2763	3430	2602	48	41	19	20	23
S	0	0	0	0	0	0	0	0	0	0	0	0
Sort	2216	2640	3252	58	26	4809	2620	3589	1225	57	3250	31
Szymanski	1805	0	1944	1122	0	1133	1103	0	0	4492	1133	1900
Texas	0	0	0	0	0	0	0	0	0	0	0	0
toilet	4090	3847	1	438	6014	1630	3	3	923	1768	15	16
Summary	26178	29255	22757	14781	26732	26722	31136	25570	28079	18485	23816	30530

Table 6.5: CPU times achieved by the 10 different heuristics, the portfolio solver (PF), and the dynamic version (DYN) on instances across 20 benchmark families contained in the test set. For each benchmark and approach the CPU times used on solved instances is shown. The summary line shows the total CPU time over all benchmark families.

6.4 Conclusions and Future Work

We believe that machine learning can be helpful to a much larger degree when solving hard combinatorial problems than it is already shown here. With QBF there exist many additional choices besides heuristics (e.g., whether to apply stronger inference/partitioning/relaxation/incomplete techniques or not) that, if selected automatically, could drastically improve the performance of a QBF solver. This applies to the portfolio-approach as well as to the online setting. Since the problem of predicting multiple labels simultaneously can entail a combinatorial explosion, recent work on structure prediction (see e.g., [70] for an overview) could be useful for this purpose.

Further directions for future work include optimal feature selection (e.g., sparse logistic regression [99]), and the use of non-linear prediction models. Besides reducing the run-time complexity of the classifier further, optimal feature selection might also shed some light on which features correlate with which technique. For instance, given only a limited amount of features is it possible to say that variable elimination is a better technique when some particular features are high/low? And do these particular features have any practical or theoretical meaning? In general, it would be interesting to determine which problem structures work well with which technique in a more focused, transparent manner. These insights might then also help us to proceed toward a theoretical comparison between for example search and variable elimination in the context of QBF.

The hardest challenge for the non-linear prediction models (e.g., neural networks [87], [51]) will be run-time efficiency, which might rule out kernel-based, and other non-parametric, methods.

Finally, the most important task remains developing novel solving techniques. Since we showed here that we can automatically predict the appropriate technique on a particular (sub-)instance it is now also more sensible to develop more specialized techniques that only work on a few benchmarks or even on only a single benchmark family. Then

every such technique can be employed in its appropriate context yielding an effective general purpose solver given a wide range of solving techniques. Clearly, the presented approach provides us with a powerful tool to advance our abilities to solve QBF and other related problems such as #SAT (e.g.,[28]), QCSP (e.g.,[40],[20]), SCSP (e.g.,[19]).

Chapter 7

Conclusions and Future Work

In this chapter we provide the reader with a summary and some conclusions of the work that has been done during this thesis. Furthermore, we also discuss directions for future research.

7.1 Summary

In this thesis we have presented a wide range of novel approaches to proceed toward an efficient and a practically applicable QBF solver. The first part of this thesis has provided a detailed and novel overview on solving QBF with search. In particular, we have illustrated how search relates to the semantics of QBF in a novel fashion. In addition, we presented a new formalization of solution learning in the context of a search-based QBF solver.

Subsequently we have introduced several new techniques based on inference that have been able to improve the state-of-the-art in QBF solving. The employment of the stronger rule of inference based on extended binary resolution has been particularly successful. Especially, the crafted preprocessor achieved outstanding empirical results (Chapter 3). These benchmark results were also independently verified by the QBF competition [49]. The employment of SAT as a powerful look-ahead with in QBF solving shows how two

different search-based algorithms can be tightly integrated in order to achieve a better performance (Chapter 4). The presented divide and conquer approach shows that it is worthwhile to employ dynamic partitioning in order to solve QBF. In this context we also provide another novel theoretical insight into QBF solving by showing how learning in a partitioning-based solver can be accomplished soundly. In general we show that all techniques introduced in this thesis are sound and complete extensions of the underlying DPLL framework.

Finally, we have presented the novel idea of dynamically adapting search during QBF solving by applying machine learning techniques. We have been able to show that it is possible to automatically adjust the solving strategy during search in order to improve the overall performance. The displayed initial results verify that this approach is very promising (Chapter 6).

7.2 Conclusions From This Work

There exist two main conclusions that can be drawn from the work presented here. First, search-based solvers provide a powerful machinery to tackle the intrinsically hard problem of QBF solving. However, in contrast to SAT there exist also other approaches besides search that are worthwhile in the context of QBF. Second, QBF is in practice a much harder problem to solve than SAT. Additionally, in order to catch up with the effectiveness of SAT solvers there still exists a need to elaborate further on the current techniques employed for QBF solving.

The presented empirical results—for instance, the benchmark results in Chapter 3—show that search-based solvers are among the best for QBF. More importantly, in many cases search-based solvers define the state-of-the-art. The newly introduced approaches based on stronger rules of inference (Chapter 3), SAT as a look-ahead (Chapter 4), and dynamic partitioning (Chapter 5) not only improve our ability to solve QBF (as we have

demonstrated in our empirical results), but also provide new insights into QBF solving. The technique discussed in Chapter 6 approaches such hard problems as QBF from a novel angle by employing machine learning to adapt the solving strategy in a dynamic fashion. Chapter 3, 4, 5, and 6 show quite convincingly that the basic DPLL framework can be effectively extended by a wide variety of techniques. For instance, the discussed machine learning approach shows how a technique quite different from DPLL and logical reasoning can be seamlessly integrated. Finally, it is also worthwhile to mention that other techniques that are for instance based on variable elimination or Skolemization are also powerful tools to solve QBF (see for instance Chapters 2 and 3). Consequently, the question of whether search will eventually be the best way to solve QBF remains open despite the results achieved by the techniques developed in this thesis.

The results presented in this thesis also indicate that QBF is not only in theory a much harder problem to solve than SAT, but that it also appears to be much harder in practice. Although standard QBF solvers already utilize nearly all major and highly-developed techniques employed in todays SAT solvers (e.g., data structures, learning techniques, etc.) the performance of QBF solvers clearly lags behind the effectiveness common to SAT solvers. Only recently have benchmark families been presented that encode competitive real-world problems and that can be efficiently solved by QBF solvers (e.g., see [52]). However, the hardness of QBF also has its upsides. As shown in Chapter 6 it appears to be worthwhile to apply for instance online classification due to the complexity of QBF. In contrast, this technique seems to cause too much overhead in the context of SAT solving [55]. In summary, although we advanced research on QBF in this thesis there still exists quite some room to improve on the approaches for QBF solving.

7.3 Future Work

The techniques introduced in this thesis can be extended and further improved by a wide variety of approaches. In this section we focus on the most important ideas for future work. Concerning the preprocessor presented in Chapter 3 there exist several obvious extensions. For instance, the approach presented here could be combined with the preprocessor based on variable elimination and universal expansion ([21] and Chapter 2) in order to achieve a even more effective, complete preprocessor. In addition, the preprocessor PreQuel now only outputs the reduced original theory. However, for some benchmarks and/or solvers it might be also useful to additionally output the computed newly inferred binary clauses since they can potentially cause more propagations. For instance, a search-based solver might be guided by these additional constraints towards a solution much more rapidly.

The QBF-solvers SQBF, 2clsQ, and 2clsP (see Chapters 3, 4, and 5) can also be improved in several different ways. First, one composed solver that would tightly integrate all the techniques employed in each of these solvers (e.g., SAT as look-ahead, dynamic partitioning, etc.) and other additional techniques (e.g., incomplete strategies [41]) would clearly achieve a better performance. Furthermore, this solver could be equipped with the machine learning approach presented in Chapter 6 in order to be able to predict when to apply which technique. This would also eliminate the major drawback that is caused by these techniques when they are not effective. For instance, as shown in Chapter 5 applying dynamic partitioning on an instance that does not break up into existentially independent subcomponents can be quite harmful with respect to run-time performance. The ability to predict these cases allows us to turn off dynamic partitioning and consequently to remove this overhead from the solving procedure. Second, in order to verify the soundness of the different implementations it would be highly beneficial to either integrate a QBF verifier or at least to output a certificate of the discovered model or a refutation. The research on the verification of QBF solutions is quite recent and still in

progress [59] [107]. Note, that in contrast to the polynomial time verification of a SAT solution the verification of QBF certificates is PSPACE-complete as well [107].

The adaptive search approach presented in Chapter 6 not only shows promising results but also opens up a rich area of research. While there exist many possible technical extensions to the work presented here (see Chapter 6, e.g., non-linear prediction models, multi-label classification, etc.) it is important to realize that this approach is not restricted to search nor to QBF. For instance, one could think of a solver that combines search and a variable elimination based scheme to solve QBF. Then in this solver at each search node we could predict whether or not to proceed with search or to employ variable elimination. Given the orthogonality of search and variable elimination with respect to run-time performance and the results achieved by the portfolio-approach of those two techniques (see Chapter 6) a dynamic combination of these two approaches appear to be a very promising strategy to solve QBF. In addition, online prediction could also be integrated in a solver solely based on variable elimination in order to guide the procedure for instance towards resolving variables that cause minimal growth in the resulting theory minimally. Furthermore, the adaptive strategy can not only be utilized within different approaches for QBF solving but obviously it can also be employed in other hard combinatorial problems like for instance #SAT (e.g.,[28]), QCSP (e.g.,[40], [20]), SCSP (e.g.,[19]).

Beyond the scope of QBF the conducted research in QBF solving could also be applied to improve solving closely related problems. For instance, in #SAT the introduced technique of solution learning in QBF could be applied as well in order to determine a count on the number of solutions [29]. Similarly, in the area of QCSPs there already exists a quite active transfer from technologies developed for QBF to QCSPs (e.g., [105]) . However, there still exist powerful techniques like solution learning that can be adapted to QCSPs. Furthermore, it would be also interesting to extend the research on areas even more closely related to QBF. For instance, the problem of $k-$QBF provides an interesting

modeling framework to cover a even wider range of real-world problems. With $k-$QBF we try to answer the question of whether there exists a Q-model that satisfies along each complete path at least k clauses ([81]). This derivate of QBF could be for instance used to model preferences in a quite natural fashion (e.g., planning with preferences [13]). Note, that $k-$QBF is PSPACE-complete as well.

However, as already pointed out earlier further research should first probably focus on QBF itself in order to develop a more practically viable solver before spreading out to other sometimes even harder problems.

Bibliography

[1] M.F. Ali, S. Safarpour, A. Veneris, M.S. Abadir, and R. Drechsler. Post-verification debugging of hierarchical designs. In *International Conf. on Computer Aided Design (ICCAD)*, pages 871–876, 2005.

[2] Carlos Anstegui, Carla P. Gomes, and Bart Selman. The achilles' heel of QBF. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings of the 20st AAAI Conference on Artificial Intelligence (AAAI-05)*, pages 275–281. AAAI Press / The MIT Press, 2005.

[3] B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithms for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8:121–123, 1979.

[4] Gilles Audemard and Lakhdar Sais. A symbolic search based approach for quantified boolean formulas. In *Eigth International Conference on Theory and Applications of Satisfiability Testing (SAT 05)*, pages 16–30, 2005.

[5] Fahiem Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In *Proceedings of the 18st AAAI Conference on Artificial Intelligence (AAAI-02)*, pages 613–619, 2002.

[6] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and Complexity Results for #SAT and Bayesian Inference. In *44th Symposium on Foundations of Computer Science (FOCS)*, pages 340–351, 2003.

[7] Fahiem Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, pages 341–355, 2003.

[8] P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[9] M. Benedetti. sKizzo: a QBF decision procedure based on propositional skolemization and symbolic reasoning. Technical Report TR04-11-03, 2004.

[10] M. Benedetti. Evaluating QBFs via Symbolic Skolemization. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR04)*, number 3452 in LNCS. Springer, 2005.

[11] M. Benedetti. Extracting Certificates from Quantified Boolean Formulas. In *Proc. of 9th International Joint Conference on Artificial Intelligence (IJCAI05)*, 2005.

[12] M. Benedetti. Quantifier Trees for QBFs. In *Proc. of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT05)*, 2005.

[13] M. Bienvenu, C. Fritz, and S. McIlraith. Planning with qualitative temporal preferences. In *International Conference on Principles of Knowledge Representation and Reasoning (KR06)*, pages 134–144, Lake District, UK, 2006.

[14] A. Biere. Resolve and expand. In *Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT2004)*, pages 238–246, 2004.

[15] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*, 1999.

[16] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1579, pages 193–207, 1999.

[17] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC'99)*, 1999.

[18] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 454–464, London, UK, 2001. Springer-Verlag.

[19] Lucas Bordeaux and Horst Samulowitz. On the stochastic constraint satisfaction framework. In *SAC 2007 (The 22nd Annual ACM Symposium on Applied Computing)*, 2007.

[20] Lucas Bordeaux and Lintao Zhang. A solver for quantified boolean and linear constraints. In *SAC 2007: Proceedings of the 2007 ACM symposium on Applied computing*, pages 321–325, New York, NY, USA, 2007. ACM Press.

[21] U. Bubeck and H. Kleine Buning. Bounded universal expansion for preprocessing QBF. In *Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2007.

[22] H. K. Büning, M. Karpinski, and A. Flügel. Resolution for quantified boolean formulas. *Inf. Comput.*, 117(1):12–18, 1995.

[23] Joshua Buresh-Oppenheim and Toniann Pitassi. The complexity of resolution refinements. In *IEEE Symposium on Logic in Computer Science*, pages 138–147, 2003.

[24] S.R. Buss. An introduction to proof theory. In S.R. Buss, editor, *Handbook of Proof Theory*, pages 1–78. Elsevier Science Publishers B. V., 1998.

[25] M. Cadoli, A. Giovanardi, and M. Schaerf. An algorithm to evaluate quantified boolean formulae. In *Proceedings of the 13st AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 262–267, 1998.

[26] P. Chatalic and L. Simon. Multi-resolution on compressed sets of clauses. In *ICTAI '00: Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'00)*, page 2, Washington, DC, USA, 2000. IEEE Computer Society.

[27] Adnan Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, 11(1-2):11–34, 2001.

[28] J. Davies and F. Bacchus. Using more reasoning to improve #sat solving. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*. AAAI Press, 2007.

[29] J. Davies, E. Hsu, and H. Samulowitz. Solution backjumping for #sat. NESCAI 2007 (North East Student Colloquium on Artificial Intelligence), 2007.

[30] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 4:394–397, 1962.

[31] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[32] James P. Delgrande, Torsten Schaub, Hans Tompits, and Stefan Woltran. On Computing Belief Change Operations using Quantified Boolean Formulas. *Journal of Logic Computation*, 14(6):801–826, 2004.

[33] A. Tacchella E. Giunchiglia, M. Narizzano. Clause/term resolution and learning in the evaluation of quantified boolean formulas. *Journal of Artificial Intelligence Research (JAIR)*, 36(4):345–377, 2006.

[34] N. Een and A. Biere. Effective Preprocessing in SAT through Variable and Clause Elimination. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*. 2005.

[35] Niklas Een and Niklas Srensson. An extensible sat-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[36] U. Egly, T. Eiter, V. Klotz, H. Tompits, and S. Woltran. Computing stable models with quantified boolean formulas: Some experimental results. In *Proceedings of the AAAI Spring Symposium*, pages 53–59, 2001.

[37] Uwe Egly, Thomas Eiter, Hans Tompits, and Stefan Woltran. Solving advanced reasoning tasks using quantified boolean formulas. In *AAAI*, pages 417–422, 2000.

[38] R. Feldmann, B. Monien, and S. Schamberger. A distributed algorithm to evaluate quantified boolean formulae. In *Proceedings of the AAAI National Conference (AAAI)*, pages 285–290, 2000.

[39] Rainer Feldmann, Burkhard Monien, and Stefan Schamberger. A distributed algorithm to evaluate quantified boolean formulae. In *AAAI*, pages 285–290, 2000.

[40] Ian P. Gent, Peter Nightingale, and Kostas Stergiou. QCSP-solve: A solver for quantified constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 138–143, 2005.

[41] I.P. Gent, H.H. Hoos, A.G.D. Rowley, and K. Smyth. Using stochastic local search to solve quantified boolean formulae. In *Principles and Practice of Constraint Programming — CP'2003*, pages 348–362, 2003.

[42] I.P. Gent and T. Walsh. The search for satisfaction. Internal Report, Department of Computer Science, University of Strathclyde, 1999.

[43] M. GhasemZadeh, V. Klotz, and Ch. Meinel. Zqsat: A useful tool for circuit verification. In *International Workshop on Logic and Synthesis (IWLS 2004), California, USA*, pages 135–142, 2004.

[44] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for quantified boolean logic satisfiability. In *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 275–281, 2001.

[45] E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. www.qbflib.org.

[46] E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: A system for deciding quantified boolean formulas satisfiability. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 364–369, 2001.

[47] E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for quantified boolean logic satisfiability. In *Eighteenth national conference on Artificial intelligence*, pages 649–654, 2002.

[48] E. Giunchiglia, M. Narizzano, and A. Tacchella. Monotone literals and learning in QBF reasoning. In *Principles and Practice of Constraint Programming (CP2004)*, pages 260–273. Springer-Verlag, New York, 2004.

[49] E. Giunchiglia, M. Narizzano, and A. Tacchella. The QBF2006 competition, 2006. available on line at http://www.qbflib.org/.

[50] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability Solvers. Elsevier Science Publishers B. V., to appear.

[51] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 2001.

[52] H.Mangassarian, A.Veneris, and M.Benedetti. Fault diagnosis using quantified boolean formulas. In *IEEE Silicon Debug and Diagnosis Workshop (SDD)*, 2007.

[53] Holger H. Hoos and Thomas Stützle. Satlib: An online resource for research on sat. In *Third International Conference on Theory and Applications of Satisfiability Testing (SAT 2000)*, pages 283–292, 2000. SATLIB is available online at http://www.satlib.org.

[54] Jinbo Huang and Adnan Darwiche. A structure-based variable ordering heuristic for sat. In *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 1167–1172, 2003.

[55] F. Hutter. Personal communication. University of British Columbia, Vancouver, hutter@cs.ubc.ca, 2007.

[56] Frank Hutter, Youssef Hamadi, Holger H. Hoos, and Kevin Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *Principles and Practice of Constraint Programming*, 2006.

[57] Shigeki Iwata and Takumi Kasai. The othello game on an nxn board is pspace-complete. *Theor. Comput. Sci.*, 123(2):329–340, 1994.

[58] Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. Counting models using connected components. In *Proceedings of the AAAI National Conference (AAAI)*, pages 157–162, 2000.

[59] T. Jussila, A. Biere, C. Sinz, D. Kroening, and C. Wintersteiger. A first step towards a unified proof checker for QBF. In *Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2007.

[60] Toni Jussila and Armin Biere. Compressing bmc encodings with QBF. *Electron. Notes Theor. Comput. Sci.*, 174(3):45–56, 2007.

[61] Jacob Katz. Model checking with quantified boolean formulas. Master Thesis, Tel-Aviv University, Raymond and Beverly Sackler Faculty of Exact Sciences The School of Computer Science, 2005.

[62] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In Howard Shrobe and Ted Senator, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201. AAAI Press, 1996.

[63] Henry Kautz and Bart Selman. The state of SAT. *Discrete Appl. Math.*, 155(12):1514–1524, 2007.

[64] M. Lagoudakis and M. Littman. Learning to select branching rules in the dpll procedure for satisfiability. In *Workshop on Theory and Applications of Satisfiability Testing (SAT 01)*, 2001.

[65] G. Lakemeyer and H. Levesque. Planning with qualitative temporal preferences. In *International Conference on Principles of Knowledge Representation and Reasoning (KR02)*, pages 14–23, Lake District, UK, 2002.

[66] Reinhold Letz. Lemma and model caching in decision procedures for quantified boolean formulas. In *TABLEAUX '02: Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 160–175, 2002.

[67] Wei Li and Peter van Beek. Guiding real-world sat solving with dynamic hypergraph separator decomposition. In *ICTAI '04: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'04)*, pages 542–548, Washington, DC, USA, 2004. IEEE Computer Society.

[68] A.C. Ling, D.P. Singh, and S.D. Brown. FPGA PLB evaluation using quantified boolean satisfiability. In *Field Programmable Logic and Applications*, 2005.

[69] Inês Lynce and ao P. Marques-Silva Jo˙An overview of backtrack search satisfiability algorithms. *Annals of Mathematics and Artificial Intelligence*, 37(3):307–326, 2003.

[70] Roland Memisevic. An introduction to structured discriminative learning. Technical report, University of Toronto, Toronto, Canada, 2006.

[71] S. Minato. Binary decision diagrams and applications to VLSI CAD. Kluwer, 1996.

[72] R. C. Moore. The role of logic in knowledge representation and common sense reasoning. In *Proceedings of the AAAI National Conference (AAAI)*, pages 428–433, 1982.

[73] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. of the Design Automation Conference (DAC)*, 2001.

[74] M. Narizzano, L. Pulina, and A. Taccchella. QBF solvers competitive evaluation (QBFEVAL), 2006. http://www.qbflib.org/qbfeval.

[75] M. Narizzano, L. Pulina, and A. Tacchella. The third QBF solvers comparative evaluation. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:145–164, 2006.

[76] M. Narizzano and A. Tacchella. QBF evaluation, 2005. http://www.qbflib.org/qbfeval/2005.

[77] M. Narizzano and A. Tacchella. QBF competition, 2006. http://www.qbflib.org/qbfeval/2006.

[78] Eugene Nudelman, Kevin Leyton-Brown, Alex Devkar, Yoav Shoham, and Holger Hoos. Satzilla: An algorithm portfolio for sat. In *Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT2005)*, pages 13–14, 2004.

[79] G. Pan and M. Y. Vardi. Symbolic decision procedures for QBF. In *Principles and Practice of Constraint Programming*, number 3258 in Lecture Notes in Computer Science, pages 453–467. Springer-Verlag, New York, 2004.

[80] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2007.

[81] T. Pitassi. Personal communication. University of Toronto, Toronto, toni@cs.toronto.edu, 2007.

[82] Ch. Pollet and Jeffrey B. Remmel. Non-monotonic reasoning with quantified boolean constraints. In *LPNMR '97: Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 18–39, London, UK, 1997. Springer-Verlag.

[83] Anja Remshagen and Klaus Truemper. An effective algorithm for the futile questioning problem. *Journal of Automated Reasoning*, 34(1):31–47, 2005.

[84] J. Rintanen. Asymptotically optimal encodings of conformant planning in qbf. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 1045–1050. AAAI Press, 2007.

[85] Jussi Rintanen. Constructing conditional plans by a theorem-prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

[86] Jussi Rintanen. Improvements to the evaluation of quantified boolean formulae. In *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, pages 1192–1197, 1999.

[87] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by back-propagating errors. *Nature*, 1986.

[88] Ashish Sabharwal. *Algorithmic Applications of Propositional Proof Complexity*. PhD thesis, University of Washington, 2005.

[89] H. Samulowitz. The efficiency and implementation of an evaluation-based reasoning procedure with disjunctive information in first-order knowledge bases. Master Thesis, University of Technology, Aachen (RWTH), Department of Computer Science, 2003.

[90] H. Samulowitz and F. Bacchus. Using SAT in QBF. In *Principles and Practice of Constraint Programming*. Springer-Verlag, New York, 2005.

[91] H. Samulowitz and F. Bacchus. Dynamically partitioning for solving QBF. In *Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2007.

[92] H. Samulowitz, J. Davies, and F. Bacchus. Preprocessing QBF. In *Principles and Practice of Constraint Programming*. Springer-Verlag, New York, 2006.

[93] H. Samulowitz, J. Davies, and F. Bacchus. QBF Preprocessor Prequel, 2006. available at http://www.cs.toronto.edu/~fbacchus/sat.html.

[94] H. Samulowitz and R. Memisevic. Learning to solve QBF. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*. AAAI Press, 2007.

[95] Horst Samulowitz and Fahiem Bacchus. Binary clause reasoning in QBF. In *Ninth International Conference on Theory and Applications of Satisfiability Testing (SAT 06)*, 2006.

[96] Horst Samulowitz and Fahiem Bacchus. QBF solver 2clsQ, 2006. available at http://www.cs.toronto.edu/˜fbacchus/sat.html.

[97] T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[98] SHARCNET. Shared hierarchical academic research computing network, 2007. http://www.sharcnet.ca.

[99] S.K. Shevade and S.S. Keerthi. A simple and efficient algorithm for gene selection using sparse logistic regression. *Bioinformatics*, 19(17):2246–2253, 2003.

[100] T. Skolem. über die mathematische logik, 1928.

[101] S. Staber and R. Bloem. Fault localization and correction with QBF. In *Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2007.

[102] L.J. Stockmeyer and A.R. Meyer. Word problems requiring exponential time. *Journal of the ACM*, pages 1–9, 1973.

[103] Miroslav N. Velev and Randal E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 226–231, New York, NY, USA, 2001. ACM Press.

[104] Guillaume Verger and Christian Bessiere. The complexity of theorem proving procedures. In *Proceedings of the 33rd Annual ACM Symposium on the Theory of Computation*, pages 151–158, 1971.

[105] Guillaume Verger and Christian Bessiere. Blocksolve: a bottom-up approach for solving quantified CSPs. In *Proceedings of CP'06*, pages 635–649, Nantes, France, 2006.

[106] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla-07: The design and analysis of an algorithm portfolio for SAT. In *Principles and Practice of Constraint Programming (CP'07)*, 2007.

[107] Y. Yu and S. Malik. Validating the result of a quantified boolean formula (QBF) solver: theory and practice. In *Proceedings, ASPDAC'05*, pages 1047–1051, January 2005.

[108] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4/5/6):543–560, April, May & June 1996.

[109] L. Zhang. *Searching for truth: Techniques for Satisfiability of Boolean Formulas.* PhD thesis, Princeton University, 2003.

[110] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, November 2001.

[111] L. Zhang and S. Malik. Towards symmetric treatment of conflicts and satisfaction in quantified boolean satisfiability solver. In *Principles and Practice of Constraint Programming (CP2002)*, pages 185–199, 2002.