Dynamically Partitioning for Solving QBF

Horst Samulowitz and Fahiem Bacchus
Department of Computer Science, University of Toronto, Canada.
[horst| fbacchus]@cs.toronto.edu

Abstract. In this paper we present a new technique to solve Quantified Boolean Formulas (QBF). Our technique applies the idea of dynamic partitioning to QBF solvers. Dynamic partitioning has previously been utilized in #SAT solvers that count the number of models of a propositional formula. One of the main differences with the #SAT case comes from the solution learning techniques employed in search based QBF solvers. Extending solution learning to a partitioning solver involves some considerable complexities which we show how to resolve. We have implemented our ideas in a new QBF solver, and demonstrate that dynamic partitioning is able to increase the performance of search based solvers, sometimes significantly. Empirically our new solver offers performance that is superior to other search based solvers and in many cases superior to non-search based solvers.

1 Introduction

The variables of a SAT problem are implicitly existentially quantified: SAT asks the question "does there exist a setting of these variables that satisfies the formula?" QBF is a generalization of SAT in which the variables are allowed to be universally as well as existential quantified: QBF asks the question "is the formula true for all settings of the universal variables." The ability to nest universal and existential quantification in arbitrary ways makes QBF considerable more expressive than SAT. While any NP problem can be encoded in SAT, QBF allows us to encode any PSPACE problem: QBF is PSPACE-complete.

This expressiveness opens a much wider range of potential application areas for a QBF solver, including areas like automated planning, non-monotonic reasoning, electronic design automation, scheduling, and model checking and verification, e.g., [1–3]. The difficulty, however, is that QBF is in practice a much harder problem to solve than SAT. (It is also much 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).

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. For example, in [4] the authors give an innovative application of QBF to hardware debugging, showing that the QBF encoding of the problem is many times smaller than an equivalent SAT encoding. Results like this demonstrate the potential of QBF and the importance of further improving QBF solvers.

In this paper we present a new technique for improving QBF solvers. Our technique extends the idea of dynamic partitioning, prominently utilized in #SAT solvers, to make it useful 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 [5]. 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 [6, 7] 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 deal with partitioning 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 [8] solver, which with the addition of some preprocessing techniques was the top scoring solver in the 2006 QBF competition. 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 [9] and Skizzo [10]. 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 background, setting the context for our methods. We then present the details of how clause and particularly cube 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.

2 Background

A quantified boolean formula has the form $\boldsymbol{Q}.F$, where F is a propositional formula expressed in CNF and $\boldsymbol{Q}$ is a sequence of quantified variables ($\forall x$ or $\exists x$). We require that no variable appear twice in $\boldsymbol{Q}$ and that all variables in F appear in $\boldsymbol{Q}$ (i.e., F contains no free variables). $\boldsymbol{Q}$ may have extra variables not mentioned in F. Such variables can be removed or retained—they do not affect the truth of the QBF.

QBF solvers are interested in answering the question of whether or not $\boldsymbol{Q}.F$ expresses a true or false assertion, i.e., whether or not $\boldsymbol{Q}.F$ is true or false. The **reduction** of a CNF formula F by a literal ℓ is the new CNF $F|_\ell$ which is F with all clauses containing ℓ removed and $\neg\ell$, the negation of ℓ, removed from all remaining clauses. The reduction of F by a **set** of literals L is defined to be the sequential reduction of F by each literal in L. It can easily be observed that the final reduced CNF is independent of the order these reductions are performed.

1. If F is the empty set of clauses then $\boldsymbol{Q}.F$ is *true*.
2. If F contains an empty clause then $\boldsymbol{Q}.F$ is *false*.

3. $\forall v \boldsymbol{Q}.F$ is *true* iff both $\boldsymbol{Q}.F|_v$ and $\boldsymbol{Q}.F|_{\neg v}$ are true.
4. $\exists v \boldsymbol{Q}.F$ is *true* iff at least one of $\boldsymbol{Q}.F|_v$ and $\boldsymbol{Q}.F|_{\neg v}$ is true.

A **quantifier block** qb of $\boldsymbol{Q}$ is a maximal contiguous subsequence of $\boldsymbol{Q}$ where every variable in qb has the same quantifier type. We order the quantifier blocks by their sequence of appearance in $\boldsymbol{Q}$: $qb_1 \leq qb_2$ iff qb_1 is equal to or appears before qb_2 in $\boldsymbol{Q}$. Each variable x in F appears in some quantifier block denoted by $qb(x)$.

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 $\boldsymbol{Q}$ is $\forall$ ($\exists$).*
3. *A variable x is **downstream** (**upstream**) of a set of variables V if (1) $x \notin V$ and (2) $\forall y. y \in V \rightarrow y \leq_q x$ ($\forall y. y \in V \rightarrow x \leq_q y$)). That is, x is not a member of V and appears no sooner (later) in the quantifier sequence $\boldsymbol{Q}$ 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) $\forall y. y \in V \rightarrow y \leq_q x$ ($\forall y. y \in V \rightarrow x \leq_q y$). That is x is a member of V and appears in the highest (lowest) quantifier block amongst all variables of V.*

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 $\boldsymbol{Q} = \exists e_1 e_2. \forall u_1 u_2. \exists e_3 e_4$ and 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 $\{u_1, e_3, e_4\}$.

We make two useful observations about QBFs (an easy proof is given, e.g., in [11]).

Observation 1

A. *If $F \models F'$ then $\boldsymbol{Q}.F \Rightarrow \boldsymbol{Q}.F'$. That is, if every SAT model of F is also a SAT model of F' then if $\boldsymbol{Q}.F$ is true $\boldsymbol{Q}.F'$ must also be true. Note that this holds even when F' contains a superset of F's variables.*

B. *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 [12] of a clause c is the process of removing all tailing universals from c. If F' is the result of applying universal reduction to some clause of F, then $\boldsymbol{Q}.F \Leftrightarrow \boldsymbol{Q}.F'$.*

2.1 Partitioning QBF

Now we discuss the conditions under which a QBF can be partitioned into a conjunction of independent sub-formulas. First we recall two standard logical laws for quantifiers. Let Φ_1 and Φ_2 be propositional formulas.

1. If Φ_1 does not contain x then $\exists x.(\Phi_1 \wedge \Phi_2) \Leftrightarrow (\Phi_1 \wedge \exists x.\Phi_2)$ 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 that can be divided into two CNF's F_1 and F_2 such that the clauses in F_1 and F_2 share no existential variables, then $\boldsymbol{Q}.F \Leftrightarrow \boldsymbol{Q}_1.F_1 \wedge \boldsymbol{Q}_2.F_2$, where $\boldsymbol{Q}_i$ is the subsequence of $\boldsymbol{Q}$ containing only the variables of F_i.*

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

For example,

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

2.2 Partitioning for a Search Based QBF Solver

Observation 2 immediately yields a partitioning search based QBF solver.

```
1  QBF-Prt($\boldsymbol{Q}.F$)
2  if $F$ contains an [empty clause/is empty] then
3      return([FALSE/TRUE])
4  for $\ell \in \{v, \bar{v}\}$ for some $v \in F$ with outermost scope in $\boldsymbol{Q}$ do
5      success = TRUE
7      Partitions = Partition($\boldsymbol{Q}.F|_\ell$)
8      foreach $\boldsymbol{Q}_i.P_i \in$ Partitions while success do
10         if QBF-Prt($\boldsymbol{Q}_i.P_i$) = FALSE then
11             success = FALSE
12     if [$\neg$success/success] AND $v$ is [universal/existential] then
13         return[FALSE/TRUE]
14 if $v$ is [universal/existential] then
15     return[TRUE/FALSE]
```

That is, we branch on variables respecting the order of quantification, just as in a standard search based QBF solver. However, after every variable has been instantiated (at this stage some propagation can also be performed 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 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 [13, 14]. Without these techniques a partitioning solver performs much worse than current search based solvers. One of the key contributions of our work is to show how learning can be extended so that it can be applied in the context of partitioning.

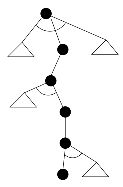
2.3 Quantifier Trees

In [15] 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 differences 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 $v = \text{TRUE}$ can be entirely different to the partitioning generated when $v = \text{FALSE}$. In a quantifier tree the partitioning will be the same for both truth values. Since this difference compounds as we set more variables this means that the partitions generated dynamically can be considerably more refined than those specified in a static quantifier tree.[1]

3 Learning with Partitioning

Search based QBF solvers employ the powerful techniques of clause and cube (solution) learning [13, 14]. 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.

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*.



3.1 Clause Learning

For the most part clause learning 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

[1] In [7] it was shown that for #SAT dynamic partitioning can yield a super-polynomial speedup over *any* static decomposition on some instances. We suspect that a similar result holds for QBF, but this is not yet proven.

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. The search will continue as before from that backtrack point. Similarly, the learnt clauses can then be used in unit propagation as they normally would be in a non-partitioning solver.

The main subtleties in using clause learning with partitioning have already been addressed in [16] 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. In particular, 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.

3.2 Cube Learning

In order to extend cube learning to allow partitioning we must first develop a new formalization of cube learning.

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, ***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, ***restrict***(F, V), by restricting each of its clauses. Note that if V contains all of the variables in c then ***restrict***$(c, V) = c$, and similarly ***restrict***$(F, V) = F$ if V contains all variables in F. We say that a QBF $\boldsymbol{Q}.F$ is **satisfied by the variables** V if the QBF $\boldsymbol{Q}$.***restrict***(F, V) is true.

We observe some facts about restriction and its relationship with reduction (setting a literal to be true).

Observation 3

1. *If $V \subseteq V'$, then **restrict**$(F, V) \models$ **restrict**(F, V').*
2. *If $\boldsymbol{Q}.F$ is satisfiable by any set of variables V, then it must also be true.*
3. *If $\ell \notin V$ then **restrict**$(F, V \cup \{\ell\})|_\ell$ is equal to **restrict**$(F|_\ell, V)$.*
4. *If $\ell \notin V$ then **restrict**$(F, V) \models$ **restrict**$(F|_\ell, V)$.*

Proof: For item 1, every clause of ***restrict***(F, V') is a superclause of a clause in ***restrict***(F, V). For item 2, this follows from item 1 and Observation 1.A by taking V' to be any 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. There are three cases to consider (a) $\ell \in c$, (b) $\neg\ell \in c$ and (c) all other clauses. For item 4, using the same three cases it can be shown that ***restrict***$(F|_\ell, V)$ contains a subset of the clauses of ***restrict***(F, V). ■

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

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

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

The following theorem justifies standard cube learning in a non-partitioning QBF solver.

Theorem 1.

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 ℓ is existential and maximal in ρ, then $\mathbf{cube}[\rho - \{\ell\}, V \cup \{\ell\}, F]$*
3. *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 $\{\ell, \neg\ell\} \subseteq \rho_1 \cup \rho_2$, (2) this clashing literal is universal, (3) ℓ is maximal in $\rho_1 \cup \rho_2$, and (4) $V_1 \cup V_2$ is downstream of $\rho_1 \cup \rho_1$, 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 $\boldsymbol{Q}.F|_\pi$ is an empty set of clause, thus it is satisfiable by any set of variables. **For item 2**, we know that $\boldsymbol{Q}$.***restrict***$(F|_\rho, V)$ is true, the claim is that $\Gamma = \boldsymbol{Q}$.***restrict***$(F|_{\rho-\{\ell\}}, V \cup \{\ell\})$ is true and that $V \cup \{\ell\}$ is downstream of $\rho - \{\ell\}$. Since $\ell \in \rho$ it must be upstream of all of the variables in V by the definition of a cube. Hence, ℓ appears in the outermost quantifier block among the variables in Γ and by definition Γ is true iff $\Gamma|_\ell$ or $\Gamma|_{\neg\ell}$ are true. In fact, $\Gamma|_\ell =$ is true: $\Gamma|_\ell = \boldsymbol{Q}$.***restrict***$(F|_{\rho-\{\ell\}}, V \cup \{\ell\})|_\ell = \boldsymbol{Q}$.***restrict***$(F|_\rho, V)$ by Observation 3.3. To see that $V \cup \{\ell\}$ is downstream of $\rho - \{\ell\}$ we observe that ℓ is maximal in ρ, so it must be downstream of $\rho - \{\ell\}$. **For item 3**, let $\rho = \rho_1 \cup \rho_2 - \{\ell, \neg\ell\}$ and let V be $V_1 \cup V_2$. We need to show that $\Gamma = \boldsymbol{Q}$.***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. $\Gamma|_\ell = \boldsymbol{Q}$.***restrict***$(F|_\rho, V \cup \{\ell\})|_\ell = \boldsymbol{Q}$.***restrict***$(F|_{\rho\cup\{\ell\}}, V)$ (Observation 3.3). Then we have that $\boldsymbol{Q}$.***restrict***$(F|_{\rho_1}, V_1)$ is true by assumption, that $\boldsymbol{Q}$.***restrict***$(F|_{\rho_1}, V_1) \Rightarrow \boldsymbol{Q}$.***restrict***$(F|_{\rho_1}, V)$ (since $V_1 \subseteq V$ and Observation 3.1), and that $\boldsymbol{Q}$.***restrict***$(F|_{\rho_1}, V) \Rightarrow \boldsymbol{Q}$.***restrict***$(F|_{\rho\cup\{\ell\}}, V)$ ($\rho \cup \{\ell\} = \rho_1 \cup (\rho_2 - \{\neg\ell\})$, all of the literals in $\rho_2 - \{\neg\ell\}$ are upstream of V, i.e., not in V, and thus Observation 3.4 applies). The proof for $\Gamma|_{\neg\ell}$ is similar. ■

Cubes are used to perform non-chronological solution backtracking that can skip large parts of the search space. They can also be stored and triggered to short-circuit the search of a subtree. In particular, if all of the literals in the cube are true, then the remaining theory is true and we need not descend in the search further.

Partial Cubes. With partitioning the 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 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 of the original clauses F_1 have been made true by the literals set along the path to n_ℓ, and we can use item 1 of Theorem 1 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, F_1 does include all clauses in the active partition P_1.

Now, we continue the search using this cube to backtrack to undo the most deeply assigned literal in π. If this literal is existential, we use item 2 of Theorem 1 to construct a new cube and backtrack further. If it is a universal we solve the other side, obtain another cube, combine the two cubes using item 3, and continue to backtrack further. At each node n we obtain a $\mathbf{cube}[\rho, V_1, F_1]$ such that ρ is a subset of the literals set along the path to n, F_1 includes all clauses in the active partition P_1 along with all other clauses made true along the path to n, and V_1 contains only variables instantiated below n (only variables backtracked over can be added into the cube). All of these variables are downstream of the variable instantiated at n which ensures that the conditions of item 3 of Theorem 1 are meet whenever it is to be applied. We also note that V_1 is always a subset of the variables of P_1 as these are the only variables branched on while solving P_1.

With partitioning, however, we cannot backtrack past node n_2 where the active partition 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.

If P_2 is true, the search in P_2's subtree will yield another cube, $\mathbf{cube}[\rho_2, V_2, F_2]$, such that F_2 includes all of the clauses of P_2 and shares with F_1 all clauses made true along the path to n_2, while V_2 is a subset of 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 2. *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: Since $V_1 \cup V_2$ is downstream of $\rho_1 \cup \rho_2$ by assumption we only need to prove that $\boldsymbol{Q}.\textbf{\textit{restrict}}(F_1 \cup F_2|_{\rho_1 \cup \rho_2}, V_1 \cup V_2)$ 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.

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

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 so the formula can be partitioned. And finally line 5 is justified by Observation 3.4: none of the literals in $\rho_1 \cup \rho_2$ 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 in P_1 and P_2 ($P_i \subseteq F_i$), 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 in the partition Q_0 and we can now utilize that cube to backtrack further within the subtree solving Q_0.

Note also that (1) all of the literals of ρ_i 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 a subset of the variables in P_i.

Finally, we note that we can also trigger cubes $[\rho, V, F']$ by storing both ρ and V. In particular, it can be shown that it is sound to trigger a cube $[\rho, V, F']$ (and terminate the search of a subtree), if (a) all of the literals in ρ are true, (b) none of the existential variables of V are assigned, and (c) the existential variables of V form a partition at the current node of the search space. Note that we do not need to keep track of the clauses the cube covers F'. However, space precludes proving this result.

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.

4 Implementation

We have implemented dynamic partitioning within the DPLL based QBF solver 2clsQ [17, 8]. 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 [17]. 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 and straight forward 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 to which a partition is constrained. The lower the ratio the more constrained are the existentials. The aim was to try to solve the most constrained partition first: if a partition failed we do not have to solve any of the other partitions 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 consists of only 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 Experimental Results

To evaluate the empirical effect of our new approach we considered all of the non-random benchmark instances from QBFLib (2005) [18] (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 our program). The time limit for a run of any solver was set to 5,000 seconds.

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 1 shows the comparison between these two

Benchmark Families	*2clsQ*		*2clsP*	
(# instances)	*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 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. The summary line shows the average success rate over all benchmark families and the total time taken (on solved instances only).

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 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%.

Benchmark Families	*Skizzo*		*Quantor*		*2clsP*		*Quaffle*		*Qube*		*SQBF*	
(# instances)	*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 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).

The total 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.

5.2 2clsP vs. Other solvers

We also compared our new solver 2clsP to five other state of the art QBF solvers **Quaffle** [13] (version as of Feb. 2005), **Quantor** [9] (version as of 2004), **Qube** (release 1.3) [19], **Skizzo** [10] (release 0.82), **SQBF** [20].

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 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.

5.3 State of the art solver

The results of the QBF competition 2006 [21] 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 hyperbinary preprocessor (PreQuel [11, 22]), 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 [11] a very promising strategy in the competition would 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 [11]. It is not clear if the employment of Quantor in the context of Skizzo is 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).

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 paper 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. Also the theory behind partial cubes can probably be elaborated further and perhaps used to obtain further algorithmic insights.

References

1. Bryant, R., Lahiri, S., Seshia, S.: Convergence testing in term-level bounded model checking. Technical Report CMU-CS-03-156, Carnegie Mellon University (2003)
2. Egly, U., Eiter, T., Tompits, H., Woltran, S.: Solving advanced reasoning tasks using quantified boolean formulas. In: AAAI/IAAI. (2000) 417–422
3. Rintanen, J.: Constructing conditional plans by a theorem-prover. Journal of Artificial Intelligence Research **10** (1999) 323–352
4. Ali, M., Safarpour, S., Veneris, A., Abadir, M., Drechsler, R.: Post-verification debugging of hierarchical designs. In: International Conf. on Computer Aided Design (ICCAD). (2005) 871–876
5. Jr., R.J.B., Pehoushek, J.D.: Counting models using connected components. In: Proceedings of the AAAI National Conference (AAAI). (2000) 157–162
6. Darwiche, A.: 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) (2001) 11–34
7. Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and Complexity Results for #SAT and Bayesian Inference. In: 44th Symposium on Foundations of Computer Science (FOCS). (2003) 340–351
8. Samulowitz, H., Bacchus, F.: QBF Solver 2clsQ (2006) available at http://www.cs.toronto.edu/˜fbacchus/sat.html.
9. Biere, A.: Resolve and expand. In: Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT). (2004) 238–246
10. Benedetti, M.: sKizzo: a QBF decision procedure based on propositional Skolemization and symbolic reasoning. Technical Report TR04-11-03 (2004)
11. Samulowitz, H., Davies, J., Bacchus, F.: Preprocessing QBF. In: Principles and Practice of Constraint Programming, Springer-Verlag, New York (2006)
12. Büning, H.K., Karpinski, M., Flügel, A.: Resolution for quantified boolean formulas. Inf. Comput. **117**(1) (1995) 12–18
13. Zhang, L., Malik, S.: Towards symmetric treatment of conflicts and satisfaction in quantified boolean satisfiability solver. In: Principles and Practice of Constraint Programming (CP2002). (2002) 185–199
14. Giunchiglia, E., Narizzano, M., Tacchella, A.: Learning for quantified boolean logic satisfiability. In: Eighteenth national conference on Artificial intelligence. (2002) 649–654
15. Benedetti, M.: Quantifier Trees for QBFs. In: Proc. of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT05). (2005)
16. Sang, T., Bacchus, F., Beame, P., Kautz, H., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: SAT. (2004)
17. Samulowitz, H., Bacchus, F.: Binary clause reasoning in qbf. In: Ninth International Conference on Theory and Applications of Satisfiability Testing (SAT 2006), Lecture Notes in Computer Science 2919. (2006)
18. Giunchiglia, E., Narizzano, M., Tacchella, A.: Quantified Boolean Formulas satisfiability library (QBFLIB) (2001) www.qbflib.org.
19. Giunchiglia, E., Narizzano, M., Tacchella, A.: QUBE: A system for deciding quantified boolean formulas satisfiability. In: International Joint Conference on Automated Reasoning (IJCAR). (2001) 364–369
20. Samulowitz, H., Bacchus, F.: Using SAT in QBF. In: Principles and Practice of Constraint Programming, Springer-Verlag, New York (2005)
21. Giunchiglia, E., Narizzano, M., Tacchella, A.: The qbf2006 competition (2006) available on line at http://www.qbflib.org/.
22. Samulowitz, H., Davies, J., Bacchus, F.: QBF Preprocessor Prequel (2006) available at http://www.cs.toronto.edu/˜fbacchus/sat.html.