A DPLL Algorithm for Solving DQBF

Andreas Fröhlich, Gergely Kovásznai, and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria*

Abstract. Dependency Quantified Boolean Formulas (DQBF) comprise the set of propositional formulas which can be formulated by adding Henkin quantifiers to Boolean logic. We are not aware of any published attempt in solving this class of formulas in practice. However with DQBF being NEXPTIME-complete, efficient ways of solving it would have many practical applications. In this paper we describe a DPLL-style approach (DQDPLL) for solving DQBF. We show how methods successfully applied in similar algorithms for SAT/QBF can be lifted to this richer logic. This enables to reuse efficient SAT and QBF solving techniques.

1 Introduction

Dependency Quantified Boolean Formulas (DQBF), as first defined in [1], are obtained by adding Henkin quantifiers [2] to Boolean formulas. In contrast to QBF, the dependencies of a variable in DQBF are not implicitly defined by the order of the quantifier prefix but are explicitly specified. The dependencies therefore are not forced to represent a total order but only a partial one.

While QBF is PSPACE-complete [3], DQBF can be shown to be NEXPTIME-complete [1,4]. Because of this, DQBF offers more succinct descriptions than QBF, provided that the two classes do not collapse. Apart from DQBF, many practical problems are known to be NEXPTIME-complete, e.g. partial information non-cooperative games [4] or certain bit-vector logics [5,6] used in the context of Satisfiability Modulo Theories (SMT).

There have been theoretical results on succinct formalizations using DQBF and certain subclasses, e.g. DQBF-Horn has been shown to be solvable in polynomial time [7]. However, we are not aware of any description on solving DQBF problems in practice, nor any actual implementation of a decision procedure for DQBF. More recently, formula expansion and transformations specific to QBF have been discussed [8], which stayed on only the theoretical side but might yield an expansion-based DQBF solver similar to those existing for QBF [9].

Effectively Propositional Logic (EPR) is another class of problems, for which the decision problem is NEXPTIME-complete. Thus there exist polynomial reductions from DQBF to EPR and vice versa. Consequently, it is also possible to use EPR solvers, e.g. iProver [10] being the currently most successful one, to solve DQBF, given some translation from DQBF to EPR. However, since EPR

* This work is partially supported by FWF, NFN Grant S11408-N23 (RiSE).

solvers in general have to reason with predicates and larger domains, solvers directly working on the propositional level should have advantages when DQBF formalizations of a problem are more natural.

Implementations of the DPLL algorithm [11], and improved variants, commonly known as CDCL solvers [12], are successfully used in many industrial applications. Inspired by the success of effective techniques used in SAT-solving, similar algorithms have been developed for QBF extending the algorithm by quantifier-reasoning and new concepts like cube learning. Although modern QBF solvers do not reach the performance of their SAT-counterparts yet, their capability also increased considerably in recent years.

This success of DPLL-style algorithms in the context of SAT and QBF gives reason to investigate how a similar algorithm could be adapted to DQBF. In the following we propose a DPLL-style [11] algorithm (DQDPLL) for solving DQBF.

2 Definitions

Let V be a set of propositional variables. A *literal* l is a variable $x \in V$ or its negation $\neg x$, and let $x = var(l)$ denote its variable. A *clause* C is a disjunction of literals. A propositional formula ϕ is in *conjunctive normal form (CNF)*, if it is a conjunction of clauses. A DQBF ψ can always be expressed as

$$\psi = Q.\phi = \forall u_1, \ldots, u_m \exists e_1(u_{1,1}, \ldots, u_{1,m_1}), \ldots, e_n(u_{n,1}, \ldots, u_{n,m_n}).\phi$$

with Q being the quantifier prefix and ϕ being a propositional formula (matrix) in CNF over the variables $V := U \cup E$ and $U = \{u_1, \ldots, u_m\}$, $E = \{e_1, \ldots, e_n\}$, $u_{i,j} \in U \ \forall\, i \in \{1, \ldots, n\}, j \in \{1, \ldots, m_i\}$. In DQBF, existential variables can always be placed after all universal variables in the quantifier prefix, since the dependencies of a certain variable are explicitly given and not implicitly defined by the order of the prefix (in contrast to QBF).

Given an existential variable e_i, we use $dep(e_i) := \{u_{i,1}, \ldots, u_{i,m_i}\}$ to denote its dependencies. For universal variables u, we define $dep(u) := \emptyset$. We also extend the notion of dependency to literals, defininig $dep(l) := dep(var(l))$ for any literal l. An *assignment* is a mapping $\alpha : V \to \{true, false\}$ from the variables of a formula to truth values. A *partial assignment* is a mapping $\beta : V \to \{true, false, undef\}$. To simplify the notation we extend the definition of assignments and partial assignments to literals, clauses and formulas in the natural way. In the rest of the paper $\alpha(l)$ (resp. $\alpha(C)$, resp. $\alpha(F)$) will denote the truth value a literal l (resp. a clause C, resp. a formula F) takes under the assignment α. We extend the notation for partial assignments β in the same way defining $undef \vee true := true$, $undef \wedge true := undef$, $undef \vee false := undef$, and $undef \wedge false := false$.

A propositional formula ϕ in CNF is satisfiable, iff all clauses in ϕ are satisfied by at least one assignment α. We then call α a model of ϕ. In QBF and DQBF a model can not be expressed by a single assignment. We use *assignment trees* [13] instead, more precisely the variant of [14]. Given a DQBF ψ, an assigment tree T is a tree with the following attributes: Every node N in T except the root

represents a truth assignment to a variable. A node has a sibling (exactly one representing the opposite truth value) if and only if it assigns a truth value to a universal variable.

Every path from the root to a leaf of T corresponds to an assignment α for the variables in ψ. In the same way a path from the root to an internal node corresponds to a partial assignment β. Compared with QBF there are two differences on the restrictions for possible trees:

Property 1: For every existential variable e and every universal variable u such that $u \in dep(e)$, the node N_u for u must be an ancestor of the node N_e for e. This ensures that for every possible path and every node N_e for an existential variable, the variable is allowed to take different values for different assignments to its dependencies, since the assignment tree splits in the corresponding node N_u.

Property 2: For each pair of paths with corresponding assignments α_1, α_2, it has to hold that $\alpha_1(e) = \alpha_2(e)$, if $\alpha_1(u) = \alpha_2(u)\ \forall\ u \in dep(e)$. This guarantees that an existential variable takes the same value in two distinct paths whenever its dependencies were assigned the same values in both paths.

A model for a DQBF $\psi = Q.\phi$ therefore is an assignment tree that fulfills both property 1 & 2 and at the same time for each path from the root to a leaf the corresponding assignment is a model for ϕ.

Actually property 1 is not needed to make sure that ψ has a solution: There is a model respecting property 1 & 2 iff there is a model respecting only property 2. This follows from the fact that removing property 1 allows existential variables to move up in the assignment tree and therefore to be assigned even before all their dependencies are assigned, i.e. to remove some dependencies. However removing dependencies makes a formula more difficult to satisfy, and therefore it is enough to consider satisfiability given property 1 & 2. This already rules out many assignment trees and yields a smaller search space.

3 DQDPLL Architecture

In the following we assume that the reader is familiar with the design of a DPLL solver for SAT/QBF. Figure 1 shows the typical pseudo-code for a QBF solver based on the DPLL algorithm. In Fig. 2 the pseudo-code of our adapted version for DQBF is presented. We will now discuss the DQDPLL algorithm in detail and point out the changes in specific methods compared to the original QBF-version.

The main underlying aspect when dealing with DQBF is the concept of dependency. As described in the previous section, a model for a DQBF formula exists iff there is an assignment tree where all paths satisfy the propositional matrix and, at the same time, the tree respects the restrictions defined by the underlying variable dependencies given in the prefix.

Instead of constructing arbitrary assignment trees and at the end checking whether they fulfill the dependency restrictions (property 1 & 2), our algorithm will only construct the subset of assignment trees that does.

```
QDPLL(F) {
  while(true) {
    state = checkState(beta);
    if (state == STATE_UNSAT) {
      level = analyseUNSAT();
      if (level == 0) return UNSAT;
      backtrack(level);
    } else if (state == STATE_SAT) {
      level = analyseSAT();
      if (level == 0) return SAT;
      backtrack(level);
    } else {
      literal = selectLiteral();
      beta = updateAssignment(literal);
      addDecision(literal);
    }
  }
}
```

Fig. 1. Main loop of QDPLL as pseudo-code

Given a partial assignment tree, *selectLiteral* decides on the next node to branch on. An arbitrary selection heuristic can be used for doing so as long as it preserves property 1 of our assignment tree. This means a universal variable can be chosen at any time and an existential variable e can be chosen whenever all $u \in dep(e)$ are already assigned in the current path of our tree. Compared to QDPLL this gives more possible decisions in each step, even given a QBF-formula as an input since decisions on existential variables may always be "delayed".

Now we have to ensure that the constructed assignment tree also fulfills property 2 from the previous section. In our DQDPLL-approach it is possible that an existential variable is set after a universal variable on which it does not depend. This cannot be avoided since we enforce a total order on the variables by our assignment tree whereas the dependency scheme of a DQBF-formula is only partially ordered. To make sure that our assignment tree nevertheless fulfills property 2 we therefore have to "remember" the choice for an existential variable under a certain assignment of its dependencies. It will then be forced to take the same value in all other branches of the tree which imply the same assignment to those universals.

In our algorithm this happens in the *addDecision* method. While the QDPLL algorithm only has to save the literal that was assigned during a decision, the DQDPLL algorithm additionally saves a *Skolem clause* C_{sk} linked with the branch on the literal of an existential variable on the decision stack. For a decision on a universal variable no Skolem clause is added (i.e. $C_{sk} = true$ in the context of our pseudo-code). The Skolem clause added for an existential decision corresponds to the restriction implied for future branches due to property 2.

```
DQDPLL(F) {
  while (true) {
    state = checkState(beta);
    if (state == STATE_UNSAT) {
      level = analyseUNSAT();
      if (level == 0) return UNSAT;
      backtrack(level);
    } else if (state == STATE_SAT) {
      level = analyseSAT();
      if (level == 0) return SAT;
      restoreAssignment(level);
    } else {
      literal = selectLiteral();
      skolemClause = generateSkolemClause(beta, literal);
      beta = updateAssignment(literal);
      addDecision(beta, skolemClause);
    }
  }
}

backtrack(level) {
  while (stack.Size > level) popStack();
  (beta, _) = stack.Element(level);
}

restoreAssignment(level) { (beta, _) = stack.Element(level); }

addDecision(beta, skolemClause) { pushStack(beta, skolemClause); }
```

Fig. 2. Main methods of DQDPLL as pseudo-code

Note that in our pseudo-code for DQDPLL we actually do not push the branching literal on the decision stack but instead the current assignment β. Of course we could at any point reconstruct the branching literal from two consecutive assignments or the other way round, reconstruct an assignment from the sequence of branching literals. We have chosen to use the notation of storing assignments in our pseudo-code because this will simplify *backTrack* and *restoreAssignment*. In a real implementation however a version saving only the branching literals probably is a better choice since it reduces the memory requirement by a factor corresponding to the number of variables.

The Skolem clause C_{sk} linked with the decision can be constructed as follows: Let β be the partial assignment corresponding to the path from the root to the current branching node and let l_{e_i} be the branching literal with $var(l_{e_i}) = e_i$, $dep(e_i) = \{u_{i,1}, \ldots, u_{i,m_i}\}$. Then

$$C_{sk} := (l_{i,1}, \ldots, l_{i,m_i}, l_{e_i}),\ l_{i,j} = \begin{cases} u_{i,j}, & \text{if } \beta(u_{i,j}) = \mathit{false} \\ \neg u_{i,j}, & \text{if } \beta(u_{i,j}) = \mathit{true} \end{cases}$$

Since we only are allowed to branch on e_i if all $u_{i,j} \in dep(e_i)$ have already been assigned, we know that $\beta(u_{i,j}) \neq undef$, i.e. C_{sk} is well-defined. Adding this C_{sk} to the formula ensures that e_i will take the same value in all other paths of the tree where all $u_{i,j} \in dep(e_i)$ are assigned the same way as in the current path, i.e. property 2 is preserved. We decided to name this a Skolem clause because it corresponds to a partial definition of the Skolem function associated with an existential variable.

It is important to note that in each step the current set of clauses consists of the original matrix conjuncted with all Skolem clauses added so far. Depending on whether *checkState* returns the current set of clauses to be satisfied, unsatisfied or undecided under the partial assignment corresponding to the current path, the algorithm continues by conflict handling, solution handling or just assigning further literals.

Whenever the current set of clauses is discovered to be *UNSAT*, a call to *analyseConflict* returns an existental decision which can be flipped. In a naive implementation this could be simply the last existential variable that was picked by a call to *selectLiteral*. During the following call to *backTrack* all decisions up to that point are undone and the corresponding Skolem clauses are removed. The decision variable is set to the opposite value and a new Skolem clause representing the neccessary constraint is introduced.

If, on the other hand, the current set of clauses is *SAT* at some point, *analyseSolution* returns a previous decision on a universal variable that still has to consider the second branch. Again in a naive implementation this could be just the latest universal variable that was picked by a call to *selectLiteral*, for which the second branch has not been checked yet. This condition should actually be considered as part of β in the pseudo-code. This time, however, in contrast to QDPLL, no backtracking takes place. Instead *restoreAssignment* is called. This method restores the assignment to the one at the point of the decision but does not undo any decisions or remove any Skolem clauses. This is important because it means we keep the Skolem clauses over different universal branches and preserve property 2 of our assignment tree.

Note that after calling *backTrack* as well as after calling *restoreAssignment* the second branch at the corresponding level has to be checked. This is not explicitely specified in our pseudo-code but for simplicity just is assumed to be part of *selectLiteral*.

Soundness and completeness of the algorithm can be checked easily:

Soundness: Altogether the given specifications of the methods guarantee that every constructed assignment tree will fulfill property 1 and property 2. Furthermore, the algorithm only returns *SAT* when all possible universal branches have been visited. This shows soundness of the DQDPLL-approach.

Completeness: Backtracking occurs as long as an existential variable can take a different value. The algorithm only returns *UNSAT* if no more backtracking is possible. Thus in the worst case all possible Skolem functions for all existential variables are enumerated, which implies completeness.

Apart from this it is also easy to check runtime and space requirements of the proposed algorithm. Due to the fact that all possible Skolem functions are enumerated in the worst case, the runtime is double-exponential. This is no surprise considering that DQBF is NEXPTIME-complete. The space required is bounded exponentially. This corresponds to the size of the current assignment tree being checked for whether it is a solution to the formula.

There are several optimizations one can consider when implementing the proposed algorithm. E.g. as already mentioned it is not necessary to save the whole assignment on the stack for each decision but instead one can only use the decision literal and later reconstruct previous assignments during *backTrack* and *restoreAssignment*. This is a bit more complicated as it is in QBF since sometimes several universal branches have to be considered and therefore variables might first get unassigned and then reassigned again to exactly reconstruct the assignment in a certain state. Still this is quite straightforward to implement but was neglected here in order to keep the pseudo-code easier to read.

Further optimizations are possible which do not backtrack in a linear way, but take advantage of the underlying tree-structure, instead of iterating through the whole stack. This again is neglected here to improve readability. As a low level optimization it is not the focus of this paper. In the next section we will look at different concepts used in DPLL algorithms for SAT/QBF and describe how they can be adapted to be used in the DQDPLL-framework.

4 Conversion of Concepts from SAT/QBF

Having described the general design of DQDPLL we now want to investigate if and how several techniques used in DPLL algorithms for SAT/QBF can be converted to the DQBF context. During the last decades many concepts have been introduced to speed up DPLL algorithms for SAT, and many of those concepts have later been adapted to QBF. Some of these are *unit propagation*, *pure literal reduction* and *clause learning*. Additionally there were also concepts especially defined for QBF, e.g. *universal reduction* and *cube learning*. Apart from these, *selection heuristics* and *watched literal schemes* also play an important role in the performance of various solvers in those domains. In this section we will describe how the abovementioned concepts can be used for DQBF.

Unit Propagation. As one of the most important techniques used in DPLL-style algorithms for SAT and QBF, unit propagation is usually implemented as part of *checkState*, which is then often referred to as *Boolean Constraint Propagation (BCP)*.

Consider a clause $C = (l_1, \ldots, l_k)$ and a partial assignment β, so that $\exists! j \in \{1, \ldots, k\} : \beta(l_j) = \mathit{undef}, \beta(l_i) = \mathit{false} \forall\, i \in \{1, \ldots, k\} \backslash \{j\}$. For SAT, $\beta(l_j)$ can then be set to *true*. For QBF, $\beta(l_j)$ can be set to *true*, if $var(l_j)$ is an existential variable, and *checkState* returns *STATE_UNSAT* otherwise. The latter one also trivially holds for DQBF, following the arguments used in the QBF version.

However in the case of an existential variable being assigned because of unit propagation, there are the following aspects we have to consider: In contrast

to selecting an existential variable e due to a decision, it is possible that not all $u \in dep(e)$ have been assigned yet when it gets propagated. Assigning e before all $u \in dep(e)$ are assigned actually violates property 1 defined in Sect. 2. Nevertheless it is still sound to do so and will help pruning the search tree.

We already argued in Sect. 2 that property 1 only was added to prevent the algorithm from constructing irrelevant assignment trees since an assignment tree not respecting property 1 corresponds only to an under-approximation of the original formula and does not preserve satisfiability. In the case of unit propagation the last observation is not true any more. If unit propagation on e is possible under a certain partial assignment β, then the same unit propagation step is possible under all possible partial assignments β' which can be constructed from β by assigning all remaining variables $u \in dep(e), \beta(u) = undef$. This means that assigning a unit e earlier, i.e. before all the universals on which it depends are assigned, does not violate any dependency restrictions of e. Actually the same effect occurs in QBF during propagation on an existential unit, if not all universals in outer scopes are assigned yet.

In order to ensure property 2 of the assignment tree, a Skolem clause needs to be added for all possible remaining assignments of $\{u \in dep(e) \mid \beta(u) = undef\}$. Using resolution and subsumption, this can be expressed by adding only one clause:

$$C_{sk} := (l_{i,1}, \ldots, l_{i,m_i}, l_{e_i}),\ l_{i,j} = \begin{cases} u_{i,j}, & \text{if } \beta(u_{i,j}) = false \\ \neg u_{i,j}, & \text{if } \beta(u_{i,j}) = true \\ false, & \text{if } \beta(u_{i,j}) = undef, \end{cases}$$

assuming $var(l_{e_i}) = e_i,\ dep(e_i) = \{u_{i,1}, \ldots, u_{i,m_i}\}$.

Pure Literal Reduction. For universal variables, pure literal reduction can be implemented exactly as it is done for QBF. Whenever a pure universal literal l_u is found, it can be set to *false*. To see that this procedure is sound, one can move the concerned universal variable outwards and expand it [8]. It is enough to consider the part where the literal is set to *false* since it subsumes the other part.

For existential variables this becomes more complicated and there is no dual version as it is the case for QBF. The reason for this is the following: setting a pure existential literal to *true* does not guarantee to preserve satisfiability since it adds a new Skolem clause to the formula (i.e. restricts the solutions), which might force the literal to the same value in some later branch of the assignment tree, although the literal is not pure there anymore. In QBF this was possible because all branches of the decision tree were independent of each other.

To guarantee that pure literal reduction on existential variables remains sound for DQBF, it can only be applied under certain conditions: An existential literal l_{e_i} can be set to *true* if every clause containing $\neg l_{e_i}$ is already satisfied by at least one l_u, $var(l_u) \in dep(l_{e_i})$ or by an existential literal l_{e_j}, $dep(l_{e_j}) \subseteq dep(l_{e_i})$. In this case we know that all clauses containing $\neg l_{e_i}$ are already satisfied whenever the newly added Skolem clause propagates l_{e_i}. This means l_{e_i} is still pure

whenever the Skolem clause propagates, and therefore the Skolem clause does not put an additional restriction on the original formula.

Clause Learning. Adding clause learning to DPLL-based SAT algorithms is responsible for a huge performance improvement of SAT solvers during the last two decades, particularly in the combination with *conflict driven clause learning (CDCL)* solvers [12]. Clause learning was then also applied to QBF [15,16,17]. In SAT as well as in QBF, it often allows to prune large parts of the search tree.

It turns out that conflict clauses in DQDPLL can be generated in the same way as it was done for QBF, and originally for SAT. The simple reason is that clause learning is based on (propositional) resolution and therefore can be applied on the matrix level, totally ignoring variable dependencies. Any resolvent of two clauses can be added to a formula without affecting satisfiability. In SAT/QBF it is common to perform resolution with clauses on the decision stack while backtracking. It can be shown that, like this, the conflict can be resolved and the new clause is asserting under the current assignment after backtracking.

However if clause learning is applied in the same way in DQDPLL, it is possible that Skolem clauses are used for resolution. The resulting resolvent therefore is only valid as long as all Skolem clauses used to create it are still part of the formula. Because of this, we need to differentiate between *temporary learned clauses* and *permanent learned clauses*.

Any learned clause created by resolution with at least one Skolem clause or with a temporary learned clause is only valid as long as all clauses participating in the resolution steps are still part of the formula, and will be a temporary learned clause itself. It therefore will be linked with the latest such clause and is removed whenever the linked clause is. A permanent learned clause is created when no Skolem clause and no temporary learned clause was part of the resolution process applied during backtracking. A clause like this can be kept or removed at any point in the same way as it is done in SAT/QBF.

Apart from this, it is also possible to create a permanent clause during backtracking if there are Skolem clauses or temporary learned clauses participating in the conflict. The algorithm can just skip the resolution steps with those clauses and, of course, ends up with a permanent clause. However in this case it is not guaranteed that the resulting clause is asserting under the current assignment after backtracking, and the permanent learned clause is less restrictive than the corresponding temporary learned clause.

It is therefore reasonable to generate both types of clauses in order to profit from the individual advantages. The temporary clause will prune larger parts of the current search tree, while the permanent clause can still affect other parts of the search tree whenever the temporary clause gets removed during further backtracking. If the permanent learned clause is too weak and does not contribute, it can be automatically deleted if removal schemes like those proposed in [18] are used.

Universal Reduction. This can be adopted for DQBF in a straightforward way. Consider a universal variable u and a clause $C = (l_u, l_1, \ldots, l_k)$, and let

$var(l_u) = u$, $\beta(l_i) \neq \mathit{true}\ \forall\, l_i \in C$. If $u \notin \bigcup_{\beta(l_i)=\mathit{undef}} \{dep(l_i)\}$, l_u can be set to *false*.

This can be seen when considering the universal expansion of C considering u. Let $C_{l_u=v}$ be the clause obtained from C by setting l_u to $v \in \{\mathit{true}, \mathit{false}\}$. A solution for F has to satisfy $C_{l_u=\mathit{true}}$ and $C_{l_u=\mathit{false}}$. Since all variables that are contained in C and are still unassigned at the current node in the assignment tree do not depend on u, they have to take the same value in both $C_{l_u=\mathit{true}}$ and $C_{l_u=\mathit{false}}$. Since $C_{l_u=\mathit{true}}$ is already satisfied by l_u, only $C_{l_u=\mathit{false}}$ needs to be considered instead of C, i.e. l_u can be removed from C.

Cube Learning. Introduced for QBF in [15,16,19], cube (goods / solution) learning is used to prune satisfied branches of the assignment tree. It can be considered as the dual concept to clause (no goods) learning, creating so-called *cubes*, i.e. a subset of literals already satisfying the formula. A cube therefore is a conjunction of literals and is added to the formula by disjunction. *Initial cubes* are created from a satisfying assignment by extracting a minimal subset of literals necessary to satisfy it. Later further cubes can be generated by using resolution on existing cubes, similar to the way new clauses are created when a conflict occurs. The same principle can still be applied to DQBF since all reasoning for creating cubes is done on the matrix level. However, similar to the reasoning necessary for adapting clause learning, a cube in DQBF is not permanent in a certain sense. When a Skolem clause is added during a decision, the set of satisfying assignments for the formula matrix shrinks. Because of this, it is possible that a cube which was added to the DQBF in a previous step does not represent a satisfying assignment for the formula matrix anymore after adding additional Skolem clauses. Whenever a Skolem clause is added to the formula, the algorithm therefore has to check whether it is satisfied by the existing cubes. Cubes not satisfying the new clause are linked with the Skolem clause and get flagged "inactive". They are not removed from the formula because they can be flagged "active" again if the Skolem clause later gets removed during backtracking.

An important point to note is that reasoning with cubes changes compared to QBF. While unit propagation on universal variables in cubes is still sound, a cube only consisting of existential variables cannot considered to be satisfied in DQBF. The reasoning behind this is the same as for pure literal reduction. Setting the remaining existential variables in a cube to *true* implies restricting the formula by Skolem clauses, i.e. it might rule out solutions and therefore does not preserve satisfiability.

Selection Heuristics. An important aspect determining the performance of a SAT solver is given by its selection heuristic. A selection heuristic determines the order of the variables getting assigned and the value they first get assigned to. In SAT there is a huge choice of different heuristics. Recently the most common heuristics are VSIDS [20] and phase saving [21]. QBF solvers suffer from the fact that variable selection is much more restricted due to the total order defined by the quantifier prefix. Only variables from the current quantifier

scope can be chosen. Sometimes this constraint can be reduced by explicitly checking for dependencies between the different variables on the matrix level, as done for example by DepQBF [22]. Note that this is a different concept. While independence on the matrix level means that the result of the formula will be the same no matter which ordering for the variables is chosen, independence in the context of DQBF is a constraint forcing a variable to take consistent values on different branches of the assignment tree.

Since variable dependencies in DQBF are less strict and the design of DQDPLL allows to "delay" decisions on existential variables, this offers more freedom on the selection of variables compared to QBF. We therefore suggest that selection heuristics have more influence in the DQBF-case. For our implementation, we used VSIDS [20] and phase saving [21] in the same way it is done in SAT, but restricted to the set of possible candidates defined by the properties of our assignment trees. It might however also be interesting to extend heuristics for DQBF by incorporating information specified on the quantifier-level, e.g. prefering existential variables over universals or picking those existential variables with dependencies most "similar" to the current universal assignment.

Watched Literal Schemes. The watched literal scheme, as a lazy data structure for unit literal detection, has proved itself to be efficient in SAT solving [23,20]. The basic idea is that the clauses are kept untouched (i.e., no literals are ever removed), and furthermore, the data structure does not require any update during backtracking. The watched literal scheme has been adapted also to QBF [24,22]. In the *two literal watching scheme*, in each clause two literals l_1 and l_2 are watched, fulfilling the following invariant: l_1 is existential, and if l_2 is universal then $var(l_2) \in dep(l_1)$. Notice that in QBF this latter condition about dependency only requires to check whether $var(l_2)$ is quantified before $var(l_1)$ in the prefix. This can be adapted to DQBF in a straightforward way, by checking the explicit dependencies of $var(l_1)$. It is important to initialize watchers on the fly for each fresh clause (i.e. conflict clause or Skolem clause). The detection of falsified, satisfied and unit clauses can be done in the same way just like in QBF.

However, a special situation, right after backtracking, has to be considered: l_1 is assigned and $l_2 = \mathit{undef}$ is universal. In QBF solvers or even in DQBF solvers respecting property 1 this situation cannot occur. However, when neglecting property 1, *backtracking to a previous path* might result in such a situation. Nevertheless, it is easy to improve the solver to avoid this situation: update the watchers of all the literals which are assigned by β, provided by the *backTrack* method. We would like to point out that this update could be highly optimized by the implementation optimization mentioned in Sect. 3, namely that only the branching literals should be saved on the decision stack instead of assignments. Given the current node n and the node n' to jump back to, let $lca(n, n')$ denote the lowest common ancestor of n and n'. During traversing the path from $lca(n, n')$ to n', update the watchers of the literals assigned by the touched nodes.

5 Preliminary Results

We implemented a prototype of our DQDPLL algorithm as introduced in Sect. 3 and added all the concepts described in Sect. 4. Testing was rather difficult since there is no DQBF library yet nor any other DQBF solver to compare results with.

Since EPR is also NEXPTIME-complete, we used EPR formulas from the TPTP and converted those formulas to DQBF. Unfortunately the conversion caused a large blow-up in the formula size. Bit-blasting of the domain, introduction of Ackermann constraints when removing predicates [25, Chapter 3.3.1], inverse destructive equality reasoning [6] to remove dependencies on other existential variables (which are not defined in DQBF) and final transformation to CNF led to an explosion in formula size. This blow-up though being polynomial produced formulas which were too large for our algorithm to solve.

Using QBF benchmarks as an input we then compared our solver with DepQBF [22]. As expected DepQBF was faster by several orders of magnitude since it is much more specialized while our solver has additional exponential overhead dealing with the stack of Skolem clauses which are not necessary for QBF. Nevertheless we could check that the returned satisfiability status of all instances solved by our algorithm was equal to the one returned by DepQBF, and therefore QBF seems to be solved correctly.

To check whether DQBF instances can be solved at all, we wrote a tool for generating random DQBF with different parameters, including number of clauses, number of existential variables, number of universal variables and expected number of dependencies per existential. We then used medium sized instances (10-50 variables, 100-1000 clauses) generated by our tool to check that our algorithm can deal with those problems and that it always produces consistent results during several hundred randomized runs, as well as very small sized instances (2-6 variables) to check correctness on this subset manually.

A further way to check correctness could be translating our randomly generated DQBF to EPR and then compare our results with the results of an EPR solver on the converted benchmark as done for QBF in [26].

6 Future Work

At the moment our algorithm is not able to solve translated EPR instances and therefore cannot compete with EPR solvers. One reason is that there is a huge blow-up during conversion. A second explanation could be the fact that those instances often were especially created using the properties of EPR. It might be interesting to look for problems which have a natural representation as DQBF instead. Maybe in domains that fit well to Boolean reasoning and do not directly suggest the usage of predicates the use of a low level DPLL-style approach is better suited and allows to profit from the well-established techniques already successful in SAT/QBF.

Apart from this, our solver is still a prototype and there are many possible optimizations regarding data structures and implementational details of our

techniques we should consider in the future. We also do not use restarts yet. Regarding the proposed concepts it will be interesting to analyze in detail, if and how each of them improves the performance of a DQBF solver based on our DQDPLL architecture.

It might also be of interest to create an expansion based solver for DQBF and see how it would compare to a DPLL-style solver such as the one we proposed. Additionally, expansion also could be used to construct a QBF out of a DQBF by expanding universal variables until the quantifiers can be totally ordered. A QBF generated this way can be given to any DPLL-based QBF solver to see if our approach of applying the concepts directly on the more succinct DQBF level gives any benefits over dealing with the less succinct QBF representation.

Finally, considering the increased complexity compared to QBF and SAT solvers, it becomes even more important to verify results. While the Skolem clauses on the decision stack after termination of our algorithm exactly define a Skolem function representing a solution, it might be interesting to check if certificates for conflicts can be generated similar to how it is done for QBF [27].

7 Conclusion

In this paper we described DQDPLL, a DPLL-style algorithm for DQBF. We have formally defined necessary conditions for assignment trees representing solutions for DQBF. Based on this, we have also shown what adaptations of the DPLL-architecture to DQBF are necessary and how they could be implemented by introducing a stack of Skolem clauses, representing partial definitions of the Skolem functions defining the existential variables.

With the main reason for the success of DPLL algorithms in SAT and QBF being found in various techniques such as *unit propagation*, *pure literal reduction* and *clause learning*, *universal reduction*, *cube learning*, *selection heuristics* and *watched literal schemes*, we also discussed how these can be translated to DQBF.

Our implementation shows that it is indeed possible to solve DQBF with this approach, at the same time, however, it does not perform very well. We have given reasons for why this is the case for EPR formulas, and suggested to find problems which can be formalized in DQBF more naturally.

Since the introduction of DQBF in [1], this paper is the first detailed description of an algorithm to solve this class of problems. While still a lot of progress has to be made in this field, we hope that our contribution helps getting a better insight on the topic of DQBF, and possibilities and pitfalls on the way of practically solving it.

References

1. Peterson, G.L., Reif, J.H.: Multiple-person alternation. In: FOCS, IEEE Computer Society (1979) 348–363
2. Henkin, L.: Some remarks on infinitely long formulas. In: Infinistic Methods, Pergamon Press (1961) 167–183

3. Papadimitriou, C.H.: Computational complexity. Addison-Wesley (1994)
4. Peterson, G., Reif, J., Azhar, S.: Lower bounds for multiplayer noncooperative games of incomplete information (2001)
5. Kovásznai, G., Fröhlich, A., Biere, A.: On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In: Proc. SMT'12. (2012) to appear.
6. Wintersteiger, C.M., Hamadi, Y., de Moura, L.: Efficiently solving quantified bit-vector formulas. In: Proc. FMCAD'10. (2010)
7. Bubeck, U., Büning, H.K.: Dependency quantified horn formulas: Models and complexity. In: Proc. SAT'06. (2006)
8. Balabanov, V., Chiang, H.J.K., Jiang, J.H.R.: Henkin quantifiers and boolean formulae. In: Proc. SAT'12. (2012) to appear.
9. Biere, A.: Resolve and expand. In: Proc. SAT'04, Springer (2004) 238–246
10. Korovin, K.: iProver — an instantiation-based theorem prover for first-order logic (system description). In: Proc. IJCAR'08. IJCAR '08, Springer (2008)
11. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. CACM **5**(7) (1962) 394–397
12. Marques-Silva, J.P., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. on Computers **48** (1999) 506–521
13. Samulowitz, H., Davies, J., Bacchus, F.: Preprocessing QBF. In: Proc. CP'06. (2006) 514–529
14. Lonsing, F., Biere, A.: Failed literal detection for QBF. In: Proc. SAT'11. SAT'11, Springer (2011) 259–272
15. Giunchiglia, E., Narizzano, M., Tacchella, A.: Learning for quantified boolean logic satisfiability. In: Proc. AAAI'02, American Association for Artificial Intelligence (2002) 649–654
16. Letz, R.: Lemma and model caching in decision procedures for quantified boolean formulas. In: Proc. TABLEAUX'02. TABLEAUX'02, Springer (2002) 160–175
17. Zhang, L.: Conflict driven learning in a quantified boolean satisfiability solver. In: Proc. ICCAD'02, ACM Press (2002) 442–449
18. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust sat-solver. In: Proc. DATE'02, IEEE Comp. Soc. (2002)
19. Zhang, L., Malik, S.: Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In: Proc. CP'02, Springer (2002) 200–215
20. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. DAC. DAC '01, ACM (2001) 530–535
21. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Proc. SAT'07. SAT'07, Springer (2007) 294–299
22. Lonsing, F., Biere, A.: Integrating dependency schemes in search-based QBF solvers. In: Proc. SAT'10, Springer (2010) 158–171
23. Zhang, H.: Sato: An efficient propositional prover. In: Proc. CADE'97. Volume 1249 of Lecture Notes in Computer Science., Springer (1997) 272–275
24. Gent, I.P., Giunchiglia, E., Narizzano, M., Rowley, A.G.D., Tacchella, A.: Watched data structures for QBF solvers. In: Selected Papers SAT'03. Lecture Notes in Computer Science, Springer (2003) 25–36
25. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Texts in Theoretical Computer Science. Springer (2008)
26. Seidl, M., Lonsing, F., Biere, A.: qbf2epr: A tool for generating epr formulas from qbf. In: Proc. PAAR'12. (2012) to appear.
27. Niemetz, A., Preiner, M., Lonsing, F., Seidl, M., Biere, A.: Resolution-based certificate extraction for QBF (tool presentation). In: Proc. SAT'12. (2012) to appear.