Integrating Dependency Schemes in Search-Based QBF Solvers

Florian Lonsing and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria
`http://fmv.jku.at/`

Abstract. Many search-based QBF solvers implementing the DPLL algorithm for QBF (QDPLL) process formulae in prenex conjunctive normal form (PCNF). The quantifier prefix of PCNFs often results in strong variable dependencies which can influence solver performance negatively. A common approach to overcome this problem is to reconstruct quantifier structure e.g. by quantifier trees. Dependency schemes are a generalization of quantifier trees in the sense that more general dependency graphs can be obtained. So far, dependency graphs have not been applied in QBF solving. In this work we consider the problem of efficiently integrating dependency graphs in QDPLL. Thereby we generalize related work on integrating quantifier trees. By analyzing the core parts of QDPLL, we report on modifications necessary to profit from general dependency graphs. In comprehensive experiments we show that QDPLL using a particular dependency graph, despite of increased overhead, outperforms classical QDPLL relying on quantifier prefixes of PCNFs.

1 Introduction

The satisfiability problem of *quantified boolean formulae (QBF)* is the canonical PSPACE-complete decision problem. QBF often allows many practically relevant-problems from the domains of model checking or automated planning to be encoded succinctly. As propositional logic (SAT), which is widely applied for modelling NP-complete problems in practice, QBF requires efficient and scalable decision procedures to be accepted for practical application.

Many QBF solvers process formulae in *prenex conjunctive normal form (PCNF)*, hence QBF encodings of problems have to be converted into PCNF first. Such conversion often comes with a loss of structural properties of the original formula. This can influence solver performance negatively.

Structure can be partially recovered to tackle this problem. A special case in this respect is the analysis of quantifier structure in QBFs, either before [8, 16] or after [2] conversion to PCNF. Such approaches allow a QBF solver to overcome the restrictions of linear quantifier prefixes in PCNFs to some extent. This applies to search- and elimination-based solvers, e.g. [4, 7, 12, 19, 20, 28, 32].

Exploiting tree-shaped quantifier structure is well-known and has been applied in different contexts. This can be achieved either by reconstructing *quantifier trees* from PCNFs [2], which is closely related to minimizing quantifier scopes by miniscoping [1], or by analyzing tree structure present in non-PCNF formulae as e.g. in [8, 16]. The latter corresponds to directly considering the parse tree of a formula and can be integrated in non-PCNF solvers such as [9, 18, 28].

Dependency schemes [30] based on [4, 5], which are relations over variables, can be regarded as a generalization of tree-shaped quantifier structure. Given a dependency scheme D, a variable x is associated with all the variables y that "depend" on x with respect to D. Informally, if y depends on x, i.e. $y \in D(x)$, then the result obtained from assigning y before x in a search-based solver may not be sound in general. Quantifier prefixes of PCNFs as well as quantifier trees fit into that framework since dependency schemes can be obtained from the prefix or the tree, respectively. Sophisticated dependency schemes were introduced in [30], all of which can be computed efficiently by syntactically analyzing PCNFs.

1.1 From Quantifier Trees to Dependency Graphs

A well-known drawback when reconstructing quantifier trees in PCNFs is non-determinism [2, 8, 9, 16]. This is related to preferring some variable over another, which can result in different trees and hence in different sets of dependencies.

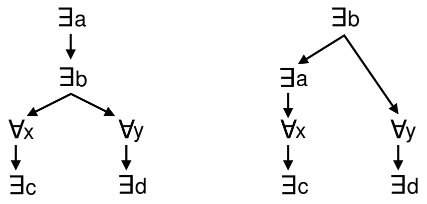

Fig. 1. Quantifier trees for the PCNF $\exists a, b \forall x, y \exists c, d.\ (a \vee b) \wedge (a \vee x \vee c) \wedge (b \vee c) \wedge (b \vee y \vee d)$. Minimizing the scope of $\exists a$ in the left tree yields the tree on the right. See also Ex. 1.

Example 1. Consider the PCNF $\exists a, b \forall x, y \exists c, d.\ (a \vee b) \wedge (a \vee x \vee c) \wedge (b \vee c) \wedge (b \vee y \vee d)$. Minimizing the scopes of $\exists c, \exists d, \forall x$ and $\forall y$ is deterministic and yields $\exists a, b.(a \vee b) \wedge (\forall x \exists c.(a \vee x \vee c) \wedge (b \vee c)) \wedge (\forall y \exists d.(b \vee y \vee d))$. Now there is the non-deterministic choice of whether to first minimize $\exists a$ and then $\exists b$ (right tree in Fig. 1) or vice versa (left tree in Fig. 1). Note that the left tree induces a dependency between a and y which is not the case in the right tree. Further, the left tree can be transformed into the tree on the right by first swapping $\exists a$ and $\exists b$ according to the rule $\exists a \exists b.\phi \equiv \exists b \exists a.\phi$ and then minimizing $\exists a$.

In addition to the problem described in Ex. 1, analyzing tree-shaped quantifier structure in general is not optimal among syntactic methods for structure analysis. This applies to reconstructed quantifier trees as well as considering tree-shaped structure present in non-PCNFs. For example, the *standard dependency scheme* D^{std} [30] is superior to tree-based approaches since it is deterministic and yields less dependencies. [1] This was pointed out in Ex. 2 in [24]. D^{std} can be efficiently constructed by analyzing connections between variables over clauses.

For those reasons and given the drawbacks of tree-shaped quantifier structure, we suggest to apply the more general concept of dependency schemes for analyzing quantifier structure in PCNFs. Similar to quantifier trees, which have already been implemented in QBF solvers, we apply *directed acyclic dependency graphs (DAGs)* in QBF solving. This generalizes quantifier trees. Dependency DAGs can be obtained from dependency schemes such as the ones introduced in [30]. When integrating dependency DAGs in QBF solvers, the drawbacks of tree-shaped quantifier structure as pointed out above can be overcome.

The core parts of our work presented here are as follows. We focus on search-based QBF solvers for PCNF which implement the *DPLL algorithm for QBF* [7] *(QDPLL)* with learning like [12, 32, 33]. By considering the main parts of QDPLL such as boolean constraint propagation, decision making or learning, we show how to integrate dependency DAGs into QDPLL in order to profit from dependency schemes in practice (Sec. 3 and 4). This analysis is closely related to [16] which aims at exploiting tree-based quantifier structure in QDPLL. Our work generalizes observations made in [16] to *arbitrary* dependency schemes. Further we address implementation-related issues indispensable for practical efficiency of dependency DAGs. Although we focus on PCNF and QDPLL, our results are, just as quantifier trees, relevant for any QBF solver.

We provide a comprehensive experimental evaluation (Sec. 5) of dependency DAGs in practice. For this purpose we have implemented QDPLL with learning in a new QBF solver DepQBF [23] which tightly integrates dependency DAGs. We analyze the costs of moving from relatively simple structures like linear quantifier prefixes of PCNFs or trees to more general dependency DAGs. This is closely related to practical applicability. Finally, we evaluate dynamic effects on QDPLL when using dependency DAGs for different dependency schemes.

In DepQBF we implemented a common framework for dependency DAGs which can represent linear quantifier prefixes and trees as well, thus enabling us to compare these approaches. Apart from that, we have implemented D^{std} as suggested in [30]. The remarks on how to profit from dependency schemes in QDPLL (Sec. 4) are general and hold independently from our implementation.

We show in experiments (Sec. 5) that, despite increased overhead, QDPLL with a DAG representation of D^{std} outperforms QDPLL relying on quantifier prefixes and trees. Our results indicate the potential of using dependency schemes in QDPLL in terms of more powerful rules for detecting unit literals and learning.

2 Preliminaries

For a set of propositional variables V, a *literal* is either a variable $x \in V$ or its negation $\neg x$ where $v(x) = x$ and $v(\neg x) = x$ denotes the variable of a literal. A *clause* (*cube*) is a disjunction (conjunction) over literals. A propositional formula

[1] Note that here we ignore variable polarities in dependency analysis. Otherwise, quantifier trees would have to be compared to the polarity-aware triangle scheme $D^{\triangle}$ [30].

is in *conjunctive normal form (CNF)* if it consists of a conjunction over clauses. A quantified boolean formula (QBF) $F = S_1 \dots S_n.\, \phi$ in *prenex conjunctive normal form (PCNF)* consists of a propositional formula ϕ in CNF over a set of variables V and a *quantifier prefix* $S_1 \dots S_n$. The quantifier prefix is a linearly ordered set of *scopes* S_i forming a partition on V. A scope S_i is *existential* ($q(S_i) = \exists$) if it is associated with an existential quantifier and *universal* ($q(S_i) = \forall$) otherwise. For scopes S_i and S_{i+1}, $q(S_i) \neq q(S_{i+1})$ for $1 \leq i < n$. The set of existential and universal variables is denoted by $V_\exists = \bigcup S_i$ for $q(S_i) = \exists$ and $V_\forall = \bigcup S_i$ for $q(S_i) = \forall$. For a literal x with $v(x) \in S_i$, $q(x) = q(S_i)$ is the *type* of x. For a clause (cube) C and $Q \in \{\forall, \exists\}$, $L_Q(C) := \{l \in C \mid q(l) = Q\}$. For literals l, k with $v(l) \in S_i$ and $v(k) \in S_j$, $l \leq k$ if, and only if $i \leq j$ for $1 \leq i, j \leq n$.

3 Dependency Schemes in Theory

Due to space limitations, we introduce dependency schemes only informally and refer to the original definition in [30]. As we focus on QDPLL with learning [12, 22, 32, 33] (see Sec. 4.1), we confine the theoretical framework in that respect.

Definition 1. *For a PCNF F, a* dependency scheme *is a relation $D \subseteq ((V_\exists \times V_\forall) \cup (V_\forall \times V_\exists))$ with the following property when applied in QDPLL: for variables x and y with $y \notin D(x)$, the result of QDPLL when assigning y before x will be sound. The* inverse *of D is $\overline{D} := \{(y, x) \mid (x, y) \in D\}$.*

Def. 1 is related to the semantical evaluation of a QBF by QDPLL and corresponds to *cumulative* dependency schemes as defined in [30], which guarantees soundness of assigning y before x if $y \notin D(x)$. It is based on independence ($y \notin D(x)$), rather than dependence ($y \in D(x)$). In practice, independence between x and y with respect to D allows y to be assigned earlier. Consequently, if $y \in D(x)$ then the result of QDPLL when assigning y before x as a decision variable is *not* sound in general[2]. At the same time it is *not always* unsound. This is due to different amounts of independence identified by different dependency schemes. For a PCNF there could be dependency schemes D and D' such that $y \in D'(x)$ but $y \notin D(x)$. Hence dependency schemes can be compared according to the amount independence.

Definition 2. *For a PCNF F and dependency schemes D and D', D is* less restrictive *if, and only if $|D| \subseteq |D'|$.*

Example 2. For the QBF from Ex. 1, let D^{triv} be the trivial dependency scheme obtained from the prefix of F: $y \in D^{\text{triv}}(x)$ if, and only if $q(x) \neq q(y)$ and $x \leq y$. Let D^{tree} be obtained from the left tree in Fig. 1: $y \in D^{\text{tree}}(x)$ if, and only if $q(x) \neq q(y)$ and y is a successor of x in the tree. Then D^{tree} is less restrictive than D^{triv} since $D^{\text{tree}} \subseteq D^{\text{triv}}$. For example, $d \in D^{\text{triv}}(x)$ but $d \notin D^{\text{tree}}(x)$.

A dependency scheme induces a partial order on the set of variables V which can be represented as a directed acyclic graph (DAG) over V.

[2] Assignments by unit and pure literals [7] are always sound independently from D.

Definition 3. *Given a dependency scheme D, the* dependency graph *for D is a DAG $G(D)$ with vertices V and edges $E := \{(x, y) \mid y \in D(x)\}$.*

4 Dependency Schemes in QDPLL

Many implementations of QDPLL rely on the quantifier prefixes of PCNFs, which corresponds to D^{triv} as defined in Ex. 2. In this section we analyze the core parts of QDPLL. We point out how to modify those parts in order to profit from less restrictive dependency schemes other than D^{triv} in QDPLL. In our analysis, we generalize observations from using quantifier trees in QDPLL [16]. Our results are *independent* from a particular dependency scheme (Def. 1).

In the following, let D be an arbitrary dependency scheme for a PCNF. For literals x, y, we write $x \prec y$ if $v(y) \in D(v(x))$. G denotes the dependency graph for D. D is integrated into QDPLL by means of G, which is used to check if $x \prec y$. This corresponds to checking if there is an edge (x, y) in G. However, D has $\mathcal{O}(V^2)$ elements and storing all edges of G can be prohibitive. Instead, transitive edges are discarded and variables are merged into equivalence classes. Checking $x \prec y$ is done by checking successor relation between x and y in G. As shown (Sec. 5), these optimizations are indispensable for efficiency in practice.

4.1 QDPLL with Learning

We briefly introduce QDPLL with *conflict-driven clause and solution-driven cube learning* based on [33]. For a PCNF $S_1 \dots S_n.\ \phi$, an additional disjunction ψ over learnt cubes is stored: $S_1 \dots S_n.\ (\phi \vee \psi)$, also called *augmented CNF*. Fig. 2 shows a high-level view. Clauses and cubes (*constraints*) are derived by clause resolutions [6] and cube resolutions [13, 22, 33] (*constraint resolutions*). Cube "resolution" is actually consensus. Different from [32, 33] we do *not* consider to learn constraints containing complementary literals and rather follow the algorithms from [15]. Further details can also be found in e.g. [13, 22].

The core of algorithm `qdpll` in Fig. 2 is propagation of *implications* (unit and pure literals) which is carried out in `bcp` until saturation. If neither a conflicting (*conflict*), nor a satisfying assignment (*solution*) was found, i.e. the formula state is undefined under the current assignment, then a variable x is assigned as next *decision* in `select_dec_var`. Decisions are numbered ascendingly by *decision levels* $dl(x)$, starting at 1. Having assigned x as decision, all implications y are propagated by `bcp` again, where $dl(y) := dl(x)$.

Otherwise the solver has either derived a conflict or a solution. This situation corresponds to a leaf in the search tree enumerated by QDPLL. For conflicts the formula contains an *empty clause* (see Def. 5) returned by `get_initial_reason` in `analyze_leaf`. By means of successive clause resolutions, the backtrack level and a learnt clause (called *asserting clause*, see Def. 8) containing a *forced literal* are computed which is unit at the backtrack level (also called *asserting level*). We assume that `qdpll` learns asserting clauses only. The current clause R is resolved (`constraint_res`) with the *antecedent clause* (`get_antecedent`) of an existential unit literal (`get_pivot`) in R. The antecedent clause is the clause where that literal became unit. If R is asserting then resolution stops (`stop_res`).

For handling solutions, `get_initial_reason` either returns a *satisfied learnt cube* (see Def. 5) already present in the cube set ψ of the formula or a new one generated from the current assignment. Dually to clauses, an asserting cube is learnt by cube resolutions using antecedent cubes of universal unit literals.

After backtracking and unassigning variables (`backtrack`), the forced literal is assigned as unit at the backtrack level and the learnt constraint is added to the formula. Again `bcp` propagates all implications. If an empty clause or satisfied cube is derived by resolutions then `qdpll` terminates (`btlevel == INVALID`).

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

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

Fig. 2. Pseudo-code of QDPLL with conflict-driven clause and solution-driven cube learning [13, 22, 33]. Code blocks are indicated by indentation level. See also Sec. 4.1.

In the following, we generalize unit literals and learning (Def. 4, 6, 8) to arbitrary dependency schemes. Soundness is explained by reordering the quantifiers in the prefix of a PCNF F based on D by Def. 1 to obtain an equivalent PCNF F' [30]. This is possible, as Def. 1 corresponds to cumulative schemes [30]. Then original versions of Def. 4, 6, 8 in the context of prefixes (i.e. D^{triv} in Def. 2) apply to F'. Finally, F' can be converted back to F by reordering.

4.2 Unit Literal Detection

Unit literals were introduced in [7] for clauses and extended to cubes in [13, 33]. The original definition is based on quantifier prefixes of PCNFs, i.e. on D^{triv} as defined in Ex. 2, and can be generalized to arbitrary dependency schemes.

Definition 4. *A clause (cube) C is* unit *if, and only if no $l \in C$ is assigned true (false), exactly one $l_e \in L_\exists(C)$ ($l_u \in L_\forall(C)$) is unassigned, and for all unassigned $l_u \in L_\forall(C)$ ($l_e \in L_\exists(C)$): $l_u \not\prec l_e$ ($l_e \not\prec l_u$).*

Analogously, Def. 4 generalizes the definition based on quantifier trees from [16]. If a clause (cube) C is unit according to Def. 4 then l_e (l_u) can be assigned as a unit literal (`bcp` in Fig. 2). Detecting unit literals involves checking dependencies. Using a two-literal watching scheme based on [10, 26], this can be achieved lazily as follows. In each clause two unassigned literals l_1 and l_2 are watched such that either $q(l_1) = q(l_2) = \exists$ or $q(l_1) = \forall$, $q(l_2) = \exists$ and $l_1 \prec l_2$. If watched literals are updated during BCP then condition $l_1 \prec l_2$ needs to be checked in the latter case only. Literal watching in cubes can be handled dually.

4.3 Constraint Learning

In QDPLL as shown in Fig. 2, new constraints are added to the formula whenever a conflicting or satisfying assignment was found. These constraints are derived by successive resolutions, each potentially eliminating literals from the resolvent.

Definition 5. *A clause (cube) C is* empty *(*satisfied*) if, and only if no $l \in C$ is assigned true (false), and all $l \in L_\exists(C)$ ($l \in L_\forall(C)$) are assigned false (true).*

Definition 6. Universal reduction *(*existential reduction*) eliminates from a clause (cube) C all $l_u \in L_\forall(C)$ ($l_e \in L_\exists(C)$) for which there is no $l_e \in L_\exists(C)$ ($l_u \in L_\forall(C)$) with $l_u \prec l_e$ ($l_e \prec l_u$).*

Definition 7. *For universally-reduced clauses (existentially-reduced cubes) C_1 and C_2 with $v \in C_1$ and $\neg v \in C_2$ for a variable v, let $C := (C_1 \cup C_2) \setminus \{v, \neg v\}$. If C does not contain complementary literals then let C' be the result of applying universal (existential) reduction to C; C' is the* resolvent *of C_1 and C_2 on v.*

Soundness of universal reduction as part of clause resolution for QBF was proved in [6]. Existential reduction for cube resolution was applied in [13, 33]. Def. 6 generalizes the reduction rules from D^{triv} to arbitrary dependency schemes. In [16] such generalization was given for quantifier trees.

Among several learning strategies which add and remove learnt constraints according to particular quality measures [13, 22, 33], QDPLL as shown in Fig. 2 learns exactly one constraint for each conflict or solution. The learnt constraint is asserting, i.e. it is unit at the level QDPLL backtracks to, and hence will trigger a unit literal to be assigned at the backtrack level. Resolution continues until the current resolvent is asserting. This is controlled by a stop criterion.

Definition 8. *Let R denote the clause (cube) derived after some resolution steps in the learning process. For $Q := \exists$ ($Q := \forall$), let $d := max(\{dl(l) \mid l \in L_Q(R)\})$. Then R is* asserting *at level $a := max(\{dl(l) \mid l \in R \wedge dl(l) < d\})$ if, and only if*

1. *the decision variable at level d is existential (universal).*
2. *there is exactly one $l \in L_\exists(R)$ ($l \in L_\forall(R)$) with $dl(l) = d$*
3. *for all $l_u \in L_\forall(R)$ ($l_e \in L_\exists(R)$) where $l_u \prec l$ ($l_e \prec l$): l_u (l_e) must be assigned false (true) with $dl(l_u) < d$ ($dl(l_e) < d$).*

Def. 8 generalizes the stop criteria for generating asserting constraints given in [15, 33] from D^{triv} to arbitrary dependency schemes. This affects condition 3 in Def. 8 only, where dependency has to be checked. In practice, this check is deferred as far as possible by checking conditions 1 and 2 before condition 3.

4.4 Decision Making

The quantifier prefix of PCNFs restricts the freedom of QDPLL to select decision variables, as variables must be assigned "from left to right" according to the prefix (i.e. D^{triv}). In the context of dependency schemes (see Def. 1), a variable y may be assigned as decision as soon as all variables in $\overline{D}(y)$ have been assigned.

Definition 9. *A variable y is* enabled *in QDPLL if, and only if all variables in $\overline{D}(y)$ are assigned. Otherwise, y is* disabled*. A variable is a* (decision) candidate *if, and only if it is unassigned and enabled.*

Example 3. For the PCNF $\exists a \forall x, y \exists b.\ \phi$, $\overline{D^{\text{triv}}}(a) = \emptyset$, $\overline{D^{\text{triv}}}(x) = \overline{D^{\text{triv}}}(y) = \{a\}$ and $\overline{D^{\text{triv}}}(b) = \{x, y\}$. Variable a is always enabled, b is enabled as soon as both x and y are assigned and if a is assigned then both x and y are enabled.

Following from Def. 1 and 9, assigning disabled variables as decisions is not sound in general. Using less restrictive dependency schemes (see Def. 2) than e.g. D^{triv} allows more freedom to select candidates in QDPLL because $\overline{D}$ is smaller and hence variables become enabled earlier.

One candidate is heuristically selected as next decision by `select_dec_var` in Fig. 2. In practice, it is prohibitive to maintain the exact candidate set explicitly. First, this set is needed precisely in `select_dec_var` and not e.g. in `bcp`. Further, not every assignment enables, not every backtrack disables new variables.

Based on these observations, we apply the dependency graph G and maintain the set of decision candidates (DC) incrementally as follows. Before QDPLL starts, $DC := \{x \in V \mid \overline{D}(x) = \emptyset\}$, i.e. DC corresponds to the roots of G. Each time a decision is made (i.e. each time `select_dec_var` in Fig. 2 is called), DC is updated by taking into account the effects of assignments $l_1, \ldots, l_k$ made since the previous decision only. Each l_i in $l_1, \ldots, l_k$ is processed one after the other. The assignment l_i possibly enables some, not necessarily all variables in $D(v(l_i))$. There might be other variables $x \neq v(l_i)$ with $D(x) = D(v(l_i))$. If any such x is still unassigned at the time l_i is processed then l_i will *not* enable any variable in $D(v(l_i))$. This observation can be exploited by constructing G as a graph [24] over equivalence classes $[x]$ of variables: $x \approx y \Leftrightarrow D(x) = D(y)$ for $x, y \in V$. Assuming $l_i \in [x]$, *no* variable will be enabled unless *all* variables in $[x]$ are assigned. Only if this is the case, set $D(v(l_i))$ is inspected by traversing successors of $[x]$ in G and new candidates are added to DC. If successor $[y]$ with $D(y) \subseteq D(v(l_i))$ is visited, then it is checked if $[y]$ is fully assigned.

When backtracking in `backtrack`, assignments l_i in $l_1, \ldots, l_k$ made between the backtrack level and the current decision level are cleared one after the other, which possibly disables variables in $D(v(l_i))$. Assuming $l_i \in [x]$, this can happen only if *all* variables in $[x]$ are assigned at the time l_i is cleared. Only if this is the case, set $D(v(l_i))$ is inspected and disabled variables are removed from DC.

Maintaining DC as described is independent from any decision heuristic for QBF and therefore can be integrated in any implementation of `select_dec_var`. Furthermore, this approach generalizes quantifier watching [10] from quantifier prefixes to arbitrary dependency schemes.

5 Experimental Results

We have implemented QDPLL with dependency schemes as described in Sec. 4 in our PCNF-based solver DepQBF [23], which also participated in QBFEVAL'10 [11]. It differs from other search-based solvers mainly in a tight integration of dependency schemes. Apart from that, approaches implemented comprise watched data structures for detection of unit and pure literals [10, 14, 26], conflict-driven clause and solution-driven cube learning [13, 22, 32, 33], assignment caching [29], activity heuristics based on VSIDS [26] and partial restarts based on [3].

As pointed out in Sec. 4, dependency graphs G in DepQBF are represented as compact graphs over equivalence classes of variables. The data structure evolved from previous work in [24]. Although originally being tailored to the standard dependency scheme D^{std}, we also implemented dependency graphs for D^{triv} and D^{tree} (see Ex. 2) within the *same* framework. This enables us to directly compare QDPLL using those three schemes without changing any other part of the solver. To build one out of possibly many non-deterministic dependency graphs (i.e. trees) for D^{tree}, we adapted the approach from [2] to our framework.

	D^{triv}	D^{tree}	D^{std}	*QuBE6.6-np*	*QuBE6.6*
QBFEVAL'08 (3326 formulae)					
solved	1223	1221	1252	1106	2277
time	579.94	580.64	572.31	608.97	302.49
QBFEVAL'07 (1136 formulae)					
solved	533	548	567	458	734
time	497.12	484.69	469.97	549.29	348.05
Herbstritt (478 formulae)					
solved	321	357	357	296	395
time	316.06	248.20	248.07	357.52	173.53

Table 1. Performance comparison of DepQBF with quantifier prefixes (D^{triv}), quantifier trees (D^{tree}) and the standard dependency scheme (D^{std}), which is less restrictive than the other two. Average run times are given in seconds. Benchmarks include all structured formulae from *QBFEVAL'07*, *QBFEVAL'08* and from set *Herbstritt* [11]. The three versions of DepQBF do *not* apply preprocessing and differ only in the integrated dependency schemes, all other parts are *exactly* the same. For external reference, statistics of PCNF-based QuBE6.6 [12] with and without preprocessing (*QuBE6.6-np*) are listed. We did not add other solvers as we focus on evaluating QDPLL with dependency schemes and, given the results of QBF competitions [11], QuBE6.x is the state-of-the-art QDPLL-based solver.

Tab. 1 shows a comparison[3] of DepQBF with D^{triv}, D^{tree} and D^{std} on structured formulae from previous QBF competitions [11]. Dependency checking as

[3] Setup for *all* experiments reported: Ubuntu Linux 9.04, Intel® Q9550@2.83 GHz, 3 GB/900 sec. mem and time limit. Data: `http://fmv.jku.at/papers/sat10qbf.7z`

needed in Def. 4, 6 and 8 was optimized in D^{triv}: for $x, y \in V$, $x \prec y$ if, and only if $x < y$, which can be checked in constant time. This is impossible for arbitrary schemes where $x \prec y$ if, and only if $q(x) \neq q(y)$ and y is a successor of x in G. Despite that additional overhead, QDPLL with D^{std} is best on *QBFEVAL'07* and *QBFEVAL'08* and is slightly faster than D^{tree} on set *Herbstritt*. There is a large performance gap to QuBE6.6 which, different from DepQBF, uses preprocessing. However, any version of DepQBF outperforms QuBE6.6 when preprocessing is disabled. Note that, in our terminology, QuBE6.6 uses D^{triv}.

A more detailed comparison of all three versions of DepQBF considering the intersection of solved formulae is shown in Tab. 2. D^{triv} is slightly faster on the

	$D^{\text{triv}} \cap D^{\text{tree}}$		$D^{\text{triv}} \cap D^{\text{std}}$		$D^{\text{tree}} \cap D^{\text{std}}$	
QBFEVAL'08 (solved only)						
solved	1172		1196		1206	
time	**23.15**	26.68	**23.73**	25.93	25.63	**22.37**
implied/assigned	90.4%	**90.7%**	88.6%	**90.5%**	90.9%	**92.1%**
backtracks	32431	**27938**	34323	**31085**	**25106**	26136
sat. cubes/sol.	1.8%	**2.9%**	1.8%	**2.6%**	**3.6%**	3.1%
learnt constr. size	157	**99**	150	**96**	102	**95**
QBFEVAL'07 (solved only)						
solved	501		513		537	
time	**31.22**	34.46	32.76	**32.66**	33.31	**28.33**
implied/assigned	89.0%	**89.2%**	87.7%	**89.5%**	89.9%	**91.9%**
backtracks	35131	**22334**	39906	**26362**	**21945**	22323
sat. cubes/sol.	4.0%	**10.0%**	4.0%	**9.5%**	**10.8%**	9.9%
learnt constr. size	150	**101**	134	**113**	100	**96**
Herbstritt (solved only)						
solved	312		308		348	
time	26.86	**19.28**	24.41	**19.28**	**20.46**	20.83
implied/assigned	96.6%	**97.4%**	96.2%	**97.4%**	97.4%	97.4%
backtracks	26565	**1329**	26733	**1482**	**1615**	1800
sat. cubes/sol.	0%	0%	0%	0%	0%	0%
learnt constr. size	**174**	306	**173**	323	**407**	410

Table 2. Comparing combinations of DepQBF with quantifier prefixes (D^{triv}), quantifier trees (D^{tree}) and the standard dependency scheme (D^{std}). Only formulae solved by *both* solvers ($\cap$) were considered. E.g. in section "$D^{\text{triv}} \cap D^{\text{std}}$", the left column reports statistics for D^{triv}, the right one for D^{std}. Average values are given for run time in seconds, ratio of implications among all assignments, number of backtracks, ratio of satisfied learnt cubes among all identified solutions and size (i.e. number of literals) of learnt constraints. See also Sec. 4.1 for terminology.

QBFEVAL sets. On the other hand, D^{triv} yields more backtracks than D^{tree} and D^{std} on all sets. On set *Herbstritt*, the difference in this respect is a factor of up to

20. D^{tree} and D^{std}, both being less restrictive than D^{triv}, produce smaller learnt constraints on the *QBFEVAL* sets. Furthermore, D^{std} triggers more implications on all sets and D^{triv} fewer satisfied learnt cubes. These effects can be attributed to more powerful rules for unit detection and constraint reduction (Def. 4, 6).

The results from Tab. 2 indicate that moving from D^{triv} to more sophisticated dependency DAGs incurs run time overhead (except on set *Herbstritt*), but also allows QDPLL to produce shorter runs in terms of backtracks. As mentioned above, checking if $x \prec y$, which is required in unit literal detection and constraint learning, is not a constant-time operation in general dependency DAGs. Instead, G must be inspected. However, QDPLL still seems to profit from using less restrictive dependency schemes such as D^{tree} and D^{std}, as indicated in Tab. 1.

In order to assess both the costs and benefits of integrating dependency DAGs in QDPLL in more detail, we carried out the following experiment. In addition to the dependency DAG which is used for dependency checking and constraint reduction in QDPLL, called primary DAG G_1, another dependency DAG, the secondary DAG G_2, is maintained *independently* and *in parallel* for statistical computations. The idea is to compare the effects of using different DAGs *dynamically*, i.e. during a solver run. This setup allows to compute more fine-grain

QBFEVAL'08 (3326 formulae)								
	$D^{\text{triv}} \ltimes D^{\text{std}}$		$D^{\text{std}} \ltimes D^{\text{triv}}$		$D^{\text{tree}} \ltimes D^{\text{std}}$		$D^{\text{std}} \ltimes D^{\text{tree}}$	
DC/d	13801.0	13801.6	11409.7	11409.0	8932.5	8933.0	15625.6	15625.3
DC-updt.	3.23	3.16	3.30	3.43	3.38	3.37	3.30	3.36
$\prec$	1	-	6.21	-	7.15	-	6.26	-
C-red.	1.18	-	535.62	-	538.30	-	540.94	-
Herbstritt (478 formulae)								
DC/d	21.3	26.55	20.14	20.13	20.67	20.67	20.16	20.16
Pan (384 formulae) $\cup$ *Sorting-Networks (84 formulae)*								
DC/d	75.81	93.87	117.50	109.66	86.89	86.90	120.03	119.98

Table 3. Comparing costs and benefits of different dependency schemes in DepQBF (all benchmarks, time out 900 sec.). The solver maintains *two* dependency DAGs G_1 (primary) and G_2 (secondary) in parallel. E.g. in section "$D^{\text{triv}} \ltimes D^{\text{std}}$", G_1 is obtained from D^{triv} (left column), G_2 from D^{std} (right column). Note that columns "D^{std}" in "$D^{\text{std}} \ltimes D^{\text{triv}}$" and "$D^{\text{std}} \ltimes D^{\text{tree}}$" are incomparable since G_2 influences run time, i.e. "$D^{\text{std}} \ltimes D^{\text{triv}}$" and "$D^{\text{std}} \ltimes D^{\text{tree}}$" may run at different speeds. Numbers of decision candidates (*DC*, see Sec. 4.4, Def. 9) when using different DAGs are compared. Each time before decision making, the number of *DC* under the current assignment is computed. Row "*DC/d*" shows the total sum of *DC* over the total number of decisions in the benchmark set after max. 900 sec. run time. Average costs are listed for (un)assigning an l_i as defined in Sec. 4.4 for updating *DC* (*DC-updt.*), dependency checks ($\prec$) as needed in unit detection (Def. 4) and for the stop criterion (Def. 8), and constraint reduction (*C-red.*) per resolution. The latter are irrelevant for G_2 ("-").

statistics than overall run time or number of backtracks, as listed in Tab. 1 and 2. During a run of QDPLL, it is interesting to compare the numbers of decision candidates (*DC*) with respect to G_1 and G_2 under the current assignment. These numbers are computed each time before a decision is made and reflect the degree of freedom resulting from less restrictive dependency schemes (see Sec. 4.4). E.g. we expect D^{std} to allow more candidates than D^{triv} and D^{tree}. Apart from that, we want to measure average costs of dependency checking and candidate maintenance for DAGs resulting from different dependency schemes.

Tab. 3 shows results of the experiments described above. For G_1 and G_2, we compared D^{std} to D^{triv} and D^{tree} , where all four combinations were run to even out biased solver behaviour. Due to limited computational resources, we did not compare D^{triv} to D^{tree} and omitted *QBFEVAL'07*. As indicated for sets *QBFEVAL'08* and *Herbstritt*, the difference in *DC* statistics is very small in general, sometimes less than 1 candidate on average per decision. However, it seems that this is already enough for QDPLL with D^{std} to outperform D^{triv} and D^{tree} by Tab. 1. Further, *DC* statistics are also family-dependent, as shown by the results for sets *Pan* and *Sorting-Networks* in Tab. 3.

Cost statistics in Tab. 3 (rows "*DC-updt.*", "$\prec$", "*C-red.*") are correlated to the number of variables that have to be visited (i.e. pointer dereferences in our implementation) when inspecting a dependency DAG. Average costs for dependency checking and (un)assigning variables for updating *DC* before decisions or during backtracking are small. This is due to the class-based approaches described in Sec. 4. On the other hand, costs of constraint reduction are very high for D^{tree} and D^{std}. These effects are closely related to implementation. When using D^{triv}, all constraints C can be kept sorted according to scope order, which allows efficient reduction. This was implemented in DepQBF with D^{triv} and is reflected by low costs in Tab. 3. In general, such an approach is not possible and we rather reduce constraints based on classes in the dependency DAG for D^{tree} and D^{std}. Classes are collected for all literals in C before reduction, where the size of C (particularly for cubes) can be large. The statistics in Tab. 3 also include that effort. Instead of collecting from scratch, the set of classes could also be maintained incrementally for all constraints, which is currently not implemented in DepQBF. However, despite that overhead in D^{tree} and D^{std}, overall performance by Tab. 1 is still better than with D^{triv} .

6 Conclusion

Structure analysis of formulae can improve QBF solvers considerably. A common approach is the analysis of quantifier structure in PCNFs by quantifier trees. Dependency schemes generalize trees and allow to overcome related drawbacks.

In this work, we considered the problem of efficiently integrating dependency DAGs into search-based QBF solvers (QDPLL) for PCNFs. Dependency DAGs result from dependency schemes and, just as trees, represent quantifier structure. By analyzing core parts of QDPLL, we have pointed out how to profit from DAGs. Thereby we generalized related work on quantifier trees in QDPLL. The results of our analysis are independent from a particular dependency scheme. Further, quantifier DAGs are relevant for QBF solvers of any kind.

Our experiments demonstrate that a careful implementation of QDPLL integrating the standard dependency scheme D^{std} outperforms classical approaches based on quantifier prefixes and trees. Despite increased overhead, our results indicate the potential of using less restrictive dependency schemes in QDPLL, which is supported by DepQBF's performance in QBFEVAL'10 [11]. More powerful unit literal detection and constraint reduction produce more implications and shorter learnt constraints. However, we also argue that the effects to a large extent differ with respect to problem domains and QBF encodings.

As future work we want to extend our implementation to arbitrary dependency schemes. Particularly, the *triangle dependency scheme* seems to be promising since it is provably less restrictive than D^{std} [27, 30].

Finally, we want to thank Paolo Marin and Enrico Giunchiglia for providing us with a version of QuBE6.6 without preprocessing.

References

1. A. Ayari and D. A. Basin. QUBOS: Deciding Quantified Boolean Logic Using Propositional Satisfiability Solvers. In M. Aagaard and J. W. O'Leary, editors, *FMCAD*, volume 2517 of *LNCS*, pages 187–201. Springer, 2002.
2. M. Benedetti. Quantifier Trees for QBFs. In F. Bacchus and T. Walsh, editors, *SAT*, volume 3569 of *LNCS*, pages 378–385. Springer, 2005.
3. A. Bhalla, I. Lynce, J. T. de Sousa, and J. Marques-Silva. Heuristic-Based Backtracking Relaxation for Propositional Satisfiability. *Journal of Automated Reasoning (JAR)*, 35(1-3):3–24, 2005.
4. A. Biere. Resolve and Expand. In H. H. Hoos and D. G. Mitchell, editors, *SAT (Selected Papers)*, volume 3542 of *LNCS*, pages 59–70. Springer, 2004.
5. U. Bubeck and H. Kleine Büning. Bounded Universal Expansion for Preprocessing QBF. In Marques-Silva and Sakallah [25], pages 244–257.
6. H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for Quantified Boolean Formulas. *Inf. Comput.*, 117(1):12–18, 1995.
7. M. Cadoli, A. Giovanardi, and M. Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *AAAI/IAAI*, pages 262–267, 1998.
8. U. Egly, M. Seidl, H. Tompits, S. Woltran, and M. Zolda. Comparing Different Prenexing Strategies for Quantified Boolean Formulas. In Giunchiglia and Tacchella [17], pages 214–228.
9. U. Egly, M. Seidl, and S. Woltran. A Solver for QBFs in Nonprenex Form. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *ECAI*, volume 141 of *FAIA*, pages 477–481. IOS Press, 2006.
10. I. P. Gent, E. Giunchiglia, M. Narizzano, A. G. D. Rowley, and A. Tacchella. Watched Data Structures for QBF Solvers. In Giunchiglia and Tacchella [17], pages 25–36.
11. E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas Satisfiability Library (QBFLIB), 2001. `http://www.qbflib.org`.
12. E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: A System for Deciding Quantified Boolean Formulas Satisfiability. In R. Goré, A. Leitsch, and T. Nipkow, editors, *IJCAR*, volume 2083 of *LNCS*, pages 364–369. Springer, 2001.

13. E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for Quantified Boolean Logic Satisfiability. In *AAAI/IAAI*, pages 649–654, 2002.
14. E. Giunchiglia, M. Narizzano, and A. Tacchella. Monotone Literals and Learning in QBF Reasoning. In Wallace [31], pages 260–273.
15. E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. *J. Artif. Intell. Res. (JAIR)*, 26:371–416, 2006.
16. E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantifier Structure in Search-Based Procedures for QBFs. *TCAD*, 26(3):497–507, 2007.
17. E. Giunchiglia and A. Tacchella, editors. *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *LNCS*. Springer, 2004.
18. A. Goultiaeva, V. Iverson, and F. Bacchus. Beyond CNF: A Circuit-Based QBF Solver. In Kullmann [21], pages 412–426.
19. G.Pan and M. Y. Vardi. Symbolic Decision Procedures for QBF. In Wallace [31], pages 453–467.
20. T. Jussila, A. Biere, C. Sinz, D. Kröning, and C. M. Wintersteiger. A First Step Towards a Unified Proof Checker for QBF. In Marques-Silva and Sakallah [25], pages 201–214.
21. O. Kullmann, editor. *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *LNCS*. Springer, 2009.
22. R. Letz. Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In U. Egly and C. G. Fermüller, editors, *TABLEAUX*, volume 2381 of *LNCS*, pages 160–175. Springer, 2002.
23. F. Lonsing. DepQBF 0.1 Source Code, 2010. `http://fmv.jku.at/depqbf/`.
24. F. Lonsing and A. Biere. A Compact Representation for Syntactic Dependencies in QBFs. In Kullmann [21], pages 398–411.
25. J. Marques-Silva and K. A. Sakallah, editors. *Proceedings SAT'07*, volume 4501 of *LNCS*. Springer, 2007.
26. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535. ACM, 2001.
27. M.Samer. Variable Dependencies of Quantified CSPs. In I. Cervesato, H. Veith, and A. Voronkov, editors, *LPAR*, volume 5330 of *LNCS*, pages 512–527. Springer, 2008.
28. F. Pigorsch and C. Scholl. Exploiting structure in an AIG based QBF solver. In *DATE*, pages 1596–1601. IEEE, 2009.
29. K. Pipatsrisawat and A. Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In Marques-Silva and Sakallah [25], pages 294–299.
30. M. Samer and S. Szeider. Backdoor Sets of Quantified Boolean Formulas. *Journal of Automated Reasoning (JAR)*, 42(1):77–97, 2009.
31. M. Wallace, editor. *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *LNCS*. Springer, 2004.
32. L. Zhang and S. Malik. Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. In L. T. Pileggi and A. Kuehlmann, editors, *ICCAD*, pages 442–449. ACM, 2002.
33. L. Zhang and S. Malik. Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In P. Van Hentenryck, editor, *CP*, volume 2470 of *LNCS*, pages 200–215. Springer, 2002.