Technical Report AC-TR-17-011

December 2017

Dependency Learning for QBF

Tomáš Peitl, Friedrich Slivovsky, and Stefan Szeider

Dependency Learning for QBF

Tomáš Peitl, Friedrich Slivovsky( ), and Stefan Szeider

Algorithms and Complexity Group, TU Wien, Vienna, Austria
{peitl,fslivovsky,sz}@ac.tuwien.ac.at

Abstract. Quantified Boolean Formulas (QBFs) can be used to succinctly encode problems from domains such as formal verification, planning, and synthesis. One of the main approaches to QBF solving is Quantified Conflict Driven Clause Learning (QCDCL). By default, QCDCL assigns variables in the order of their appearance in the quantifier prefix so as to account for dependencies among variables. Dependency schemes can be used to relax this restriction and exploit independence among variables in certain cases, but only at the cost of nontrivial interferences with the proof system underlying QCDCL. We propose a new technique for exploiting variable independence within QCDCL that allows solvers to learn variable dependencies on the fly. The resulting version of QCDCL enjoys improved propagation and increased flexibility in choosing variables for branching while retaining ordinary (long-distance) Q-resolution as its underlying proof system. In experiments on standard benchmark sets, an implementation of this algorithm shows performance comparable to state-of-the-art QBF solvers.

1 Introduction

Conflict Driven Clause Learning (CDCL) represents the state of the art in propositional satisfiability (SAT) solving (see, e.g., [23]). Modern CDCL solvers are able to handle input formulas with thousands of variables and millions of clauses [22]. Their remarkable performance has led to the adoption of SAT solving in electronic design automation (for a survey, see [33]), it has turned algorithms relying on SAT oracles into viable tools for solving hard problems (see, e.g., [24]), and it has even helped resolve open questions in combinatorics [12].

Encouraged by this success, researchers are turning to an even harder problem: the satisfiability problem of Quantified Boolean Formulas (QSAT). Quantified Boolean Formulas (QBFs) enrich propositional formulas with universal and existential quantification over truth values and offer much more succinct encodings of problems from domains such as planning and synthesis [9]. This expressive power comes at a price: QSAT is complete for the complexity class PSPACE and thus believed to be significantly harder than SAT.

Quantified CDCL [7,34] is a natural generalization of CDCL and one of two dominant algorithmic paradigms in QSAT solving (the other being approaches broadly based on quantifier expansion [3,14,15,21,27,29,31]). While the performance of QCDCL solvers has much improved over the past years, they have so

© Springer International Publishing AG 2017
S. Gaspers and T. Walsh (Eds.): SAT 2017, LNCS 10491, pp. 298–313, 2017.
DOI: 10.1007/978-3-319-66263-3_19

far failed to replicate the success of CDCL in the domain of SAT. One of the main obstacles in lifting CDCL to QSAT is that the alternation of existential and universal quantifiers in the quantifier prefix of a QBF (we consider formulas in prenex normal form) introduces dependencies among variables that have to be respected by the order of variable assignments. The resulting constraints not only reduce the empirical effectiveness of branching heuristics but impose severe theoretical limits on the power of QCDCL [13]. By default, QCDCL only considers variables from the outermost quantifier block with unassigned variables for branching. In the worst case, this forces solvers into a fixed branching order. Among several techniques that have been introduced to relax this restriction, *dependency schemes* are arguably the most general. Given a QBF, a dependency scheme efficiently computes an overapproximation of its variable dependencies—that is, the result contains every pair of variables for which there is a "real" dependency, but it may contain "spurious" dependencies. Lonsing and Biere [5] introduced a generalization of QCDCL that uses dependency schemes to relax constraints on the order of variable assignments and implemented this algorithm in the solver DepQBF.

The use of dependency schemes within DepQBF often leads to performance improvements, but it has several drawbacks. First, it changes the proof system underlying constraint learning, and proving soundness of the resulting algorithm is nontrivial even for a simple version of QCDCL and common dependency schemes [25,30]. The continuous addition of solver features makes QCDCL a moving target, and the integration of dependency schemes with any new technique usually requires a new soundness proof. Second, even if soundness of the resulting proof system can be established, efficient (linear-time) strategy extraction from proofs—a common requirement for applications—does not follow. Third, and perhaps most importantly, the syntactic criteria for identifying dependencies used by common dependency schemes (such as the standard dependency scheme or the resolution-path dependency scheme) are fairly coarse, so that the set of dependencies computed by such schemes frequently coincides with the "trivial" dependencies implicit in the quantifier prefix (see Table 3 in Sect. 5).

In this paper, we describe a new approach to exploiting variable independence in QCDCL solvers we call *dependency learning*. The idea is that the solver maintains a set D of dependencies that is used in propagation and choosing variables for branching just like in QCDCL with dependency schemes: a clause is considered unit under the current assignment if it contains a single existential variable that does not depend, according to D, on any universal variable remaining in the clause; a variable is eligible for branching if it does not depend, according to D, on any variable that is currently unassigned (cf. [5]). But instead of initializing D using a dependency scheme, dependencies are added on the fly as needed. Initially, the set D is empty, so every clause containing a single existential variable is considered unit and variables can be assigned in any order. When propagation runs into a conflict, the solver attempts to derive a new clause by long-distance Q-resolution [1,8]. Because propagation implicitly performs universal reduction relative to D but Q-resolution applies universal reduction according to the prefix order, the solver may be unable to generate a learned clause. If this happens, a set of variable dependencies can be identified as the reason for this failure and added to D, preventing this situation from occurring in the future. The resulting version of QCDCL potentially improves on the flexibility provided by dependency schemes but retains long-distance Q-resolution as its underlying proof system and therefore supports linear-time strategy extraction [2].

To explore the effectiveness of this technique, we implemented *Qute*, a QCDCL solver that supports dependency learning. In experiments with benchmark instances from the 2016 QBF evaluation, Qute is competitive with state-of-the-art QBF solvers on formulas in prenex conjunctive normal form (PCNF). For formulas represented as quantified circuits in the QCIR format, Qute solves more instances than any other available solver. Additional experiments show that the number of dependencies learned by Qute on PCNF instances preprocessed by Bloqqer is typically only a fraction of those identified by the standard dependency scheme and even the (reflexive) resolution-path dependency scheme, and that dependency learning allows QCDCL to deal with formulas that are provably hard to solve for vanilla QCDCL [13].

The remainder of this paper is organized as follows. In Sect. 2, we cover basic definitions and notation. In Sect. 3, we introduce QCDCL and (long-distance) Q-resolution, its underlying proof system. In Sect. 4, we describe how to modify QCDCL to support dependency learning, and prove that the resulting algorithm is sound and terminating. In Sect. 5, we provide an experimental evaluation of Qute. In Sect. 6, we conclude with a discussion of our results and future research directions.

2 Preliminaries

We consider QBFs in Prenex Conjunctive Normal Form (PCNF), i.e., formulas $\Phi = \mathcal{Q}.\varphi$ consisting of a (quantifier) prefix $\mathcal{Q}$ and a propositional CNF formula φ, called the *matrix* of Φ. The *prefix* is a sequence $\mathcal{Q} = Q_1 x_1 \dots Q_n x_n$, where $Q_i \in \{\forall, \exists\}$ is a universal or existential quantifier and the x_i are variables. We write $x_i \prec_\Phi x_j$ if $1 \leq i < j \leq n$ and $Q_i \neq Q_j$, dropping the subscript if the formula Φ is understood. A *CNF formula* is a finite conjunction $C_1 \wedge \dots \wedge C_m$ of clauses, a *clause* is a finite disjunction $(\ell_1 \vee \dots \vee \ell_k)$ of literals, and a *literal* is a variable x or a negated variable $\neg x$. Dually, a *DNF formula* is a finite disjunction of $T_1 \vee \dots \vee T_k$ of terms, and a *term* is a finite conjunction $(\ell_1 \wedge \dots \wedge \ell_k)$ of literals. Whenever convenient, we consider clauses and terms as sets of literals, CNF formulas as sets of clauses, and DNF formulas as sets of terms. We assume that PCNF formulas are *closed*, so that every variable occurring in the matrix appears in the prefix, and that each variable appearing in the prefix occurs in the matrix. We write $var(x) = var(\neg x) = x$ to denote the variable associated with a literal and let $var(C) = \{\, var(\ell) : \ell \in C \,\}$ if C is a clause, $var(\varphi) = \bigcup_{C \in \varphi} var(C)$ if φ is a CNF formula, and $var(\Phi) = var(\varphi)$ if $\Phi = Q.\varphi$ is a PCNF formula.

An *assignment* is a sequence $\sigma = (\ell_1, \dots, \ell_k)$ of literals such that $var(\ell_i) \neq var(\ell_j)$ for $1 \leq i < j \leq n$. If S is a clause or term, we write $S[\sigma]$ for the the result of applying σ to S. For a clause C, we define $C[\sigma] = \top$ if $\ell_i \in C$ for some $1 \leq i \leq k$, and $C[\sigma] = C \setminus \{\overline{\ell_1}, \dots, \overline{\ell_k}\}$ otherwise, where $\overline{x} = \neg x$ and $\overline{\neg x} = x$. For a term T, we let $T[\sigma] = \bot$ if $\overline{\ell_i} \in T$ for some $1 \leq i \leq k$, and $T[\sigma] = T \setminus \{\ell_1, \dots, \ell_k\}$ otherwise. An assignment σ *falsifies* a clause C if $C[\sigma] = \emptyset$; it *satisfies* a term T if $T[\sigma] = \emptyset$. For CNF formulas φ, we let $\varphi[\sigma] = \{\, C[\sigma] : C \in \varphi, C[\sigma] \neq \top \,\}$, and for PCNF formulas $\Phi = \mathcal{Q}.\varphi$, we let $\Phi[\sigma] = \mathcal{Q}'.\varphi[\sigma]$, where $\mathcal{Q}'$ is obtained by deleting variables from $\mathcal{Q}$ not occurring in $\varphi[\sigma]$.

The semantics of a PCNF formula Φ are defined as follows. If Φ does not contain any variables then Φ is true if its matrix is empty and false if its matrix contains the empty clause $\emptyset$. Otherwise, let $\Phi = Qx\mathcal{Q}.\varphi$. If $Q = \exists$ then Φ is true if $\Phi[(x)]$ is true or $\Phi[(\neg x)]$ is true, and if $Q = \forall$ then Φ is true if both $\Phi[(x)]$ and $\Phi[(\neg x)]$ are true.

3 QCDCL and Q-Resolution

We briefly review QCDCL and Q-resolution [17], its underlying proof system. More specifically, we consider *long-distance Q-resolution*, a version of Q-resolution that admits the derivation of tautological clauses in certain cases. Although this proof system was already used in early QCDCL solvers [34], the formal definition shown in Fig. 1 was given only recently [1]. A dual proof system called *(long-distance) Q-consensus*, which operates on terms instead of clauses, is obtained by swapping the roles of existential and universal variables (the analogue of universal reduction for terms is called *existential reduction*).

$$\frac{C_1 \vee e \qquad \neg e \vee C_2}{C_1 \vee C_2} \text{ (resolution)}$$

The *resolution* rule allows the derivation of $C_1 \vee C_2$ from clauses $C_1 \vee e$ and $\neg e \vee C_2$, provided that the *pivot* variable e is existential and that $e \prec var(\ell_u)$ for each universal literal $\ell_u \in C_1$ such that $\overline{\ell_u} \in C_2$. The clause $C_1 \vee C_2$ is called the resolvent of $C_1 \vee e$ and $\neg e \vee C_2$.

$$\frac{C}{C \setminus \{u, \neg u\}} \text{ (universal reduction)}$$

The *universal reduction* rule admits the deletion of a universal variable u from a clause C under the condition that $e \prec u$ for each existential variable e in C.

Fig. 1. Long-distance Q-resolution.

A (long-distance) Q-resolution *derivation* from a PCNF formula Φ is a sequence of clauses such that each clause appears in the matrix of Φ or can be derived from clauses appearing earlier in the sequence using resolution or universal reduction. A derivation of the empty clause is called a *refutation*, and one can show that a PCNF formula is false, if, and only if, it has a long-distance Q-resolution refutation [1]. Dually, a PCNF formula is true, if, and only if, the empty term can be derived from a DNF representation of its matrix by Q-consensus.

Starting from an input PCNF formula, QCDCL generates ("learns") *constraints*—clauses and terms—until it produces an empty constraint. Every clause learned by QCDCL can be derived from the input formula by Q-resolution, and every term learned by QCDCL can be derived by Q-consensus [8,10]. Accordingly, the solver outputs TRUE if the empty term is learned, and FALSE if the empty clause is learned.

One can think of QCDCL solving as proceeding in rounds. Along with a set of clauses and terms, the solver maintains an assignment σ. During each round, this assignment is extended by quantified Boolean constraint propagation (QBCP) and—possibly—branching.

Quantified Boolean constraint propagation consists in the exhaustive application of universal and existential reduction in combination with unit assignments.[1] More specifically, QBCP reports a clause C as falsified if $C[\sigma] \neq \top$ and universal reduction can be applied to $C[\sigma]$ to obtain the empty clause. Dually, a term T is considered satisfied if $T[\sigma] \neq \bot$ and $T[\sigma]$ can be reduced to the empty term. A clause C is *unit* under σ if $C[\sigma] \neq \top$ and universal reduction yields the clause $C' = (\ell)$, for some existential literal ℓ such that $var(\ell)$ is unassigned. Dually, a term T is *unit* under σ if $T[\sigma] \neq \bot$ and existential reduction can be applied to obtain a term $T' = (\ell)$ containing a single universal literal ℓ. If $C = (\ell)$ is a unit clause, then the assignment σ has to be extended by ℓ in order not to falsify C, and if $T = (\ell)$ is a unit term, then σ has to be extended by $\overline{\ell}$ in order not to satisfy T. If several clauses or terms are unit under σ, QBCP nondeterministically picks one and extends the assignment accordingly. This is repeated until a constraint is empty or no unit constraints remain.

If QBCP does not lead to an empty constraint, the assignment σ is extended by *branching*. That is, the solver chooses an unassigned variable x such that every variable y with $y \prec x$ is assigned, and extends the assignment σ by x or $\neg x$.

The resulting assignment can be partitioned into so-called *decision levels*. The decision level of an assigment σ is the number of literals in σ that were assigned by branching. The decision level of a literal ℓ in σ is the decision level of the prefix of σ that ends with ℓ. Note that each decision level greater than 0 can be associated with a unique variable assigned by branching.

Eventually, the assignment maintained by QCDCL must falsify a clause or satisfy a term. When this happens (this is called a *conflict*), the solver proceeds to *conflict analysis* to derive a learned constraint C. Initially, C is the falsified clause (satisfied term), called the *conflict clause (term)*. The solver finds the existential (universal) literal in C that was assigned last by QBCP, and the antecedent clause (term) R responsible for this assignment. A new constraint is derived by resolving C and R and applying universal (existential) reduction.

[1] We do not consider the pure literal rule as part of QBCP.

This is done repeatedly until the resulting constraint C is *asserting*. A clause (term) S is asserting if there is a unique existential (universal) literal $\ell \in S$ with maximum decision level among literals in S, the corresponding decision variable is existential (universal), and every universal (existential) variable $y \in \mathit{var}(S)$ such that $y \prec \mathit{var}(\ell)$ is assigned at a lower decision level (an asserting constraint becomes unit after backtracking). Once an asserting constraint has been found, it is added to the solver's set of constraints. Finally, QCDCL *backtracks*, undoing variable assignments until reaching a decision level computed from the learned constraint.

4 QCDCL with Dependency Learning

We now describe how to modify QCDCL to support dependency learning. First, the solver maintains a set $D \subseteq \{\, (x, y) : x \prec y \,\}$ of variable dependencies. Second, both QBCP and the decision rule must be modified as follows:

– QBCP() uses universal and existential reduction relative to D. Universal reduction relative to D removes each universal variable u from a clause C such that there is no existential variable $e \in \mathit{var}(C)$ with $(u, e) \in D$ (existential reduction relative to D is defined dually).

Algorithm 1. QCDCL with Dependency Learning

```
procedure SOLVE( )
    $D = \emptyset$
    while TRUE do
        conflict = QBCP()
        if conflict == NONE then
            DECIDE()
        else
            $\mathit{constraint}, \mathit{btlevel}$ = ANALYZECONFLICT($\mathit{conflict}$)
            if $\mathit{constraint}$ != NONE then
                if ISEMPTY($\mathit{constraint}$) then
                    if ISTERM($\mathit{constraint}$) then
                        return TRUE
                    else
                        return FALSE
                    end if
                else
                    ADDLEARNEDCONSTRAINT($\mathit{constraint}$)
                end if
            end if
            BACKTRACK($\mathit{btlevel}$)
        end if
    end while
end procedure
```

- DECIDE() may assign any variable y such that there is no unassigned variable x with $(x, y) \in D$ (note that $(x, y) \in D$ implies $x \prec y$).

This is how DepQBF uses the dependency relation D computed by a dependency scheme in propagation and decisions [5]. Unlike DepQBF, QCDCL with dependency learning does *not* use the generalized reduction rules during conflict analysis (RESOLVE and REDUCE in lines 7 and 8 refer to resolution and reduction as defined in Fig. 1). As a consequence, the algorithm cannot always construct a learned constraint during conflict analysis (see the example below). Such cases are dealt with in lines 9 through 12 of ANALYZECONFLICT (Algorithm 2):

- EXISTSRESOLVENT(*constraint*, *reason*, *pivot*) determines whether the resolvent of *constraint* and *reason* exists.
- If this is not the case, there has to be a variable v (universal for clauses, existential for terms) satisfying the following condition: $v \prec pivot$ and there exists a literal $\ell \in constraint$ with $var(\ell) = v$ and $\overline{\ell} \in reason$. The set of such variables is computed by ILLEGALMERGES. For each such variable, a new dependency is added to D. No learned constraint is returned by conflict analysis, and the backtrack level (*btlevel*) is set so as to cancel the decision level at which *pivot* was assigned.

The criteria for a constraint to be asserting must also be slightly adapted: a clause (term) S is asserting with respect to D if there is a unique existential (universal) literal $\ell \in S$ with maximum decision level among literals in S, the corresponding decision variable is existential (universal), and every universal (existential) variable $y \in var(S)$ such that $(y, var(\ell)) \in D$ is assigned (again, this corresponds to the definition of asserting constraints used in DepQBF [19, p. 119]). Finally, in the main QCDCL loop, we have to implement a case distinction to account for the fact that conflict analysis may not return a constraint (line 9).

Algorithm 2. Conflict Analysis with Dependency Learning

```
 1: procedure ANALYZECONFLICT($conflict$)
 2:     $constraint$ = GETCONFLICTCONSTRAINT($conflict$)
 3:     while NOT ASSERTING($constraint$) do
 4:         $pivot$ = GETPIVOT($constraint$)
 5:         $reason$ = GETANTECEDENT($pivot$)
 6:         if EXISTSRESOLVENT($constraint, reason, pivot$) then
 7:             $constraint$ = RESOLVE($constraint, reason, pivot$)
 8:             $constraint$ = REDUCE($constraint$)
 9:         else
10:             $illegal\_merges$ = ILLEGALMERGES($constraint, reason, pivot$)
11:             $D = D \cup \{ (v, pivot) : v \in illegal\_merges \}$
12:             return NONE, DECISIONLEVEL($pivot$)
13:         end if
14:     end while
15:     $btlevel$ = GETBACKTRACKLEVEL($constraint$)
16:     return $constraint, btlevel$
17: end procedure
```

4.1 Examples

We now illustrate the two possible outcomes of conflict analysis with simple examples. First, take the QBF

$$\Phi = \forall u \exists e.(u \vee e).$$

Starting from an empty set D of dependencies, QCDCL with dependency learning initially assumes that e is independent of u. By applying universal reduction relative to D to the clause $(u \vee e)$, one derives the unit clause (e). Accordingly, the solver appends e to its current assignment and finds the matrix satisfied. Since (e) is a term in the DNF representation of Φ's matrix and (e) can be reduced to the empty term by existential reduction, QCDCL learns the empty term and correctly reports that Φ is true. Now consider

$$\Psi = \forall u \exists e.(u \vee e) \wedge (\neg u \vee \neg e).$$

Again, QCDCL with dependency learning starting with empty D considers the first clause unit and appends e to its assignment. Propagating this assignment to the second clause results in a conflict, as $(\neg u \vee \neg e)[e \mapsto 1] = (\neg u)$, which simplifies to the empty clause by universal reduction. During conflict analysis, the solver attempts to construct a learned clause by resolving the conflict clause $(\neg u \vee \neg e)$ with the clause $(u \vee e)$ responsible for propagating e. But since $u \prec e$ is universal and appears negated in the first and unnegated in the second clause, these two clauses do not have a resolvent in long-distance Q-resolution, and the solver is unable to learn a clause. Instead, it adds the dependency (u, e) to D and backtracks until e is unassigned. Now that D contains the dependency (u, e), the universal variable u can no longer be reduced from a clause that contains e, so neither clause is unit. Moreover, the solver cannot branch on e while u is unassigned. It is easy to see that from this point on, the solver behaves just like ordinary QCDCL on this example.

4.2 Soundness and Termination

Soundness of QCDCL with dependency learning is an immediate consequence of the following observation.

Observation 1. Every constraint learned by QCDCL with dependency learning can be derived from the input formula by long-distance Q-resolution or Q-consensus.

To prove termination, we argue that the algorithm learns a new constraint or a new dependency after each conflict. Just as in QCDCL, every learned constraint is asserting, so learning does not introduce duplicate constraints.

Observation 2. QCDCL with dependency learning never learns a constraint already present in the database.

The only additional argument required to prove termination is one that tells us that the algorithm cannot indefinitely "learn" the same dependencies.

Lemma 1. *If QCDCL with dependency learning does not learn a constraint during conflict analysis, it learns a new dependency.*

Proof. To simplify the presentation, we are only going to consider clause learning (the proof for term learning is analogous). We first establish an invariant of intermediate clauses derived during conflict analysis: they are empty under the partial assignment obtained by backtracking to the last literal in the assignment that falsifies an existential literal in the clause. Formally, let C be a clause and let $\sigma = (\ell_1, \dots, \ell_k)$ be an assignment. We define $\mathit{last}_C(\sigma) = \max(\{\, i \in [k] : \overline{\ell_i} \in C, \mathit{var}(\ell_i) \in \mathit{var}_\exists \,\} \cup \{0\})$ and let $\sigma_C = (\ell_1, \dots, \ell_{\mathit{last}_C(\sigma)})$. In particular, if $\mathit{last}_C(\sigma) = 0$ then σ_C is empty.

We now prove the following claim: if σ is an assignment that falsifies a clause, then, for every intermediate clause C constructed during conflict analysis, $C[\sigma_C]$ is empty after universal reduction. The proof is by induction on the number of resolution steps in the derivation of C. If C is the conflict clause then $C[\sigma]$ reduces to the empty clause. That means $C[\sigma_C]$ can only contain universal literals and can also be reduced to the empty clause by universal reduction. Suppose C is the result of resolving clauses C' and R and applying universal reduction, where C' is an intermediate clause derived during conflict analysis and R is a clause that triggered unit propagation. The induction hypothesis tells us that $C'[\sigma_{C'}]$ reduces to the empty clause. Since the pivot literal ℓ is chosen to be the last existential literal falsified in C', we must have $\sigma_{C'} = (\ell_1, \dots, \ell_k)$ where $\ell_k = \overline{\ell}$. Let $\tau = (\ell_1, \dots, \ell_{k-1})$. We must have $C'[\tau] = U' \cup \{\ell\}$, where U' is a purely universal clause. Because R is responsible for propagating $\overline{\ell}$, we further must have $R[\tau] = U'' \cup \{\overline{\ell}\}$, where U'' again is a purely universal clause. It follows that their resolvent $C[\tau] = (C' \setminus \{\ell\})[\tau] \cup (R \setminus \{\overline{\ell}\})[\tau] = U' \cup U''$ is a purely universal clause. Since τ is a prefix of σ, it follows that $C[\sigma_C]$ is a purely universal clause as well and therefore empty after universal reduction. This proves the claim.

We proceed to prove the lemma. If the algorithm does not learn a clause during conflict analysis, this must be due to a failed attempt at resolving an intermediate clause C with a clause R responsible for unit propagation. That is, if e is the existential pivot variable, there must be a universal variable $u \prec e$ such that $u \in \mathit{var}(C) \cap \mathit{var}(R)$ and $\{u, \neg u\} \subseteq C \cup R$. Towards a contradiction, suppose that $(u, e) \in D$. Let σ denote the assignment that caused the conflict and assume without loss of generality that $\{u, e\} \subseteq R$ and $\{\neg u, \neg e\} \subseteq C$. Since R caused propagation of e but $(u, e) \in D$, the variable u must have been assigned before e and $\neg u \in \sigma$. As the pivot $\neg e$ is the last existential literal falsified in C, it follows that $\neg u \in \sigma_C$. Because $\neg u \in C$, this implies that the assignment σ_C satisfies C, in contradiction with the claim proved above.

The number of dependencies and constraints is bounded by a function of the number n of variables, and QCDCL runs into a conflict at least every n variable assignments, so Observation 2 and Lemma 1 imply termination.

Theorem 1. *QCDCL with dependency learning is sound and terminating.*

5 Experiments

To see whether dependency learning works in practice, we implemented a QCDCL solver that supports this technique named Qute.[2] We evaluated the performance of Qute in several experiments. First, we measured the number of instances solved by Qute on benchmark sets from the 2016 QBF evaluation [26]. We compare these numbers with those of the best performing publicly available solvers for each benchmark set. In a second experiment, we computed the dependency sets given by the standard dependency scheme [4,28] and the reflexive resolution-path dependency scheme [30,32] for preprocessed instances, and compared their sizes to the number of dependencies learned by Qute. Finally, we revisit an instance family which is known to be hard to solve for QCDCL [13] and show they pose no challenge to Qute. For our experiments, we used a cluster with Intel Xeon E5649 processors at 2.53 GHz running 64-bit Linux.

5.1 Solved Instances for QBF Evaluation 2016 Benchmark Sets

In our first experiment, we used the prenex non-CNF (QCIR [16]) benchmark set from the 2016 QBF evaluation consisting of 890 formulas. Time and memory limits were set to 10 min and 4 GB, respectively. The results are summarized in Table 1 and Fig. 2. Qute is able to solve signficantly more instances within the timeout than the other solvers, and this appears to be in large part due to dependency learning: when dependency learning is deactivated, the number of solved instances drops significantly.

Table 1. Instances from the 2016 QBF evaluation prenex non-CNF (QCIR) benchmark set solved within 10 min.

Solver	Total	Sat	Unsat
Qute+dl	581	274	307
GhostQ	524	228	296
QuAbS	515	225	290
Qute	494	228	266
RAReQS	403	174	229

For our second experiment, we used the prenex CNF (PCNF) benchmark set from the 2016 QBF evaluation consisting of 825 instances. Time and memory limits were again set to 10 min and 4 GB. We performed this experiment twice: with and without preprocessing using bloqqer [6]. In order not to introduce variance in overall runtime through preprocessing, each instance was preprocessed only once and solvers were run on the preprocessed instances with a timeout corresponding to the overall timeout minus the time spent on preprocessing.

[2] http://github.com/perebor/qute.

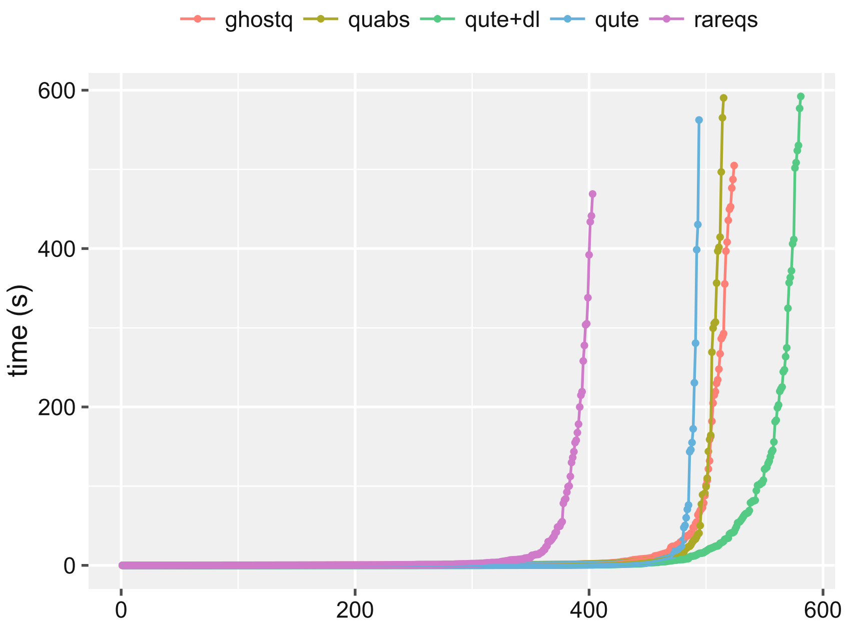

Fig. 2. Solved instances from the 2016 QBF evaluation prenex non-CNF (QCIR) benchmark set (x-axis) sorted by runtime (y-axis).

Since Qute shows good performance on QCIR instances, we included configurations that perform partial circuit reconstruction using *qcir-conv*[3] and then solve the resulting QCIR instance.

The results obtained without using bloqqer are shown on the left hand side of Table 2. When not using qcir-conv, Qute solves more instances with dependency learning than without dependency learning. Curiously, the opposite is the case when qcir-conv is used: in this case, Qute solves 3 more instances when dependency learning is turned off. Overall, we see that circuit reconstruction

Table 2. Instances from the QBF evaluation 2016 prenex CNF (PCNF) benchmark set solved within 10 min without preprocessing (left) and with preprocessing using bloqqer (right).

solver	total	sat	unsat
GhostQ	584	297	287
Qute+qcir-conv	538	283	255
Qute+dl+qcir-conv	535	277	258
DepQBF	451	200	251
Qute+dl	434	190	244
Qute	416	191	225
CAQE	358	167	191
RAReQS	335	128	207

solver	total	sat	unsat
RAReQS	615	299	316
DepQBF	585	294	291
CAQE	577	288	289
GhostQ	563	289	274
Qute+dl+qcir-conv	556	276	280
Qute+qcir-conv	541	266	275
Qute+dl	519	252	267
Qute	510	242	268

[3] http://www.cs.cmu.edu/~wklieber/qcir-conv.py.

(also used internally by GhostQ [18]) substantially increases the performance of Qute.

The results including preprocessing with bloqqer are shown on the right hand side of Table 2. With the exception of GhostQ, all solvers and configurations solve more instances when paired with bloqqer. However, the increase is less significant for Qute compared to other solvers, in particular in combination with qcir-conv. Notably, dependency learning increased the number of instances solved by Qute regardless of whether qcir-conv was used.

5.2 Learned Dependencies Compared to Dependency Relations

To get an idea of how well QCDCL with dependency learning is able to exploit independence, we compared the number of dependencies learned by Qute with the number of standard and resolution-path dependencies for instances from the PCNF benchmark set after preprocessing with bloqqer. We only considered instances with at least one quantifier alternation after preprocessing. Qute was run with a 10 min timeout (excluding preprocessing). If an instance was not solved we used the number of dependencies learned within that time limit.[4]

Summary statistics are shown in Table 3. On average, the standard dependency scheme does not provide a significant improvement over trivial dependencies. The reflexive resolution-path dependency scheme does better, but the high median shows that the set of trivial dependencies it can identify as spurious is still small in many cases. The fraction of learned dependencies is much smaller than either dependency relation on average, and the median fraction of trivial dependencies learned is even below 1%.

This indicates that proof search in QCDCL with dependency learning is less constrained than in QCDCL with either dependency scheme: since QCDCL is allowed to branch on a variable x only if every variable that x depends on has already been assigned, decision heuristics are likely to have a larger pool of variables to choose from if fewer dependencies are present.

Table 3. Learned dependencies, standard dependencies, and reflexive resolution-path dependencies for instances preprocessed by bloqqer, as a fraction of trivial dependencies.

Dependencies	Mean	Median	Variance
Learned	0.082	0.008	0.030
Standard	0.938	1.000	0.030
Resolution-path	0.660	0.942	0.172

[4] We cannot rule out that, for unsolved instances, Qute would have to learn a larger fraction of trivial dependencies before terminating. However, the solver tends to learn most dependencies at the beginning of a run, with the fraction of learned trivial dependencies quickly converging to a value that does not increase much until termination.

5.3 Dependency Learning on Hard Instances for QCDCL

For our third experiment, we chose a family of instances $\mathtt{CR}_n$ recently used to show that ordinary QCDCL does not simulate tree-like Q-resolution [13]. Since the hardness of these formulas is tied to QCDCL not propagating across quantifier levels, they represent natural test cases for QCDCL with dependency learning. We recorded the number of backtracks required to solve $\mathtt{CR}_n$ by Qute with and without dependency learning, for $n \in \{1, ..., 50\}$. As a reference, we used DepQBF.[5] For this experiment, we kept the memory limit of 4 GB but increased the timeout to one hour. The results are summarized in Fig. 3. As one would expect, Qute without dependency learning and DepQBF were only able to solve instances up to $n = 7$ and $n = 8$, respectively. Furthermore, it is evident from the plot that the number of backtracks grows exponentially with n for both solvers. By contrast, Qute with dependency learning was able to solve all instances within the timeout.

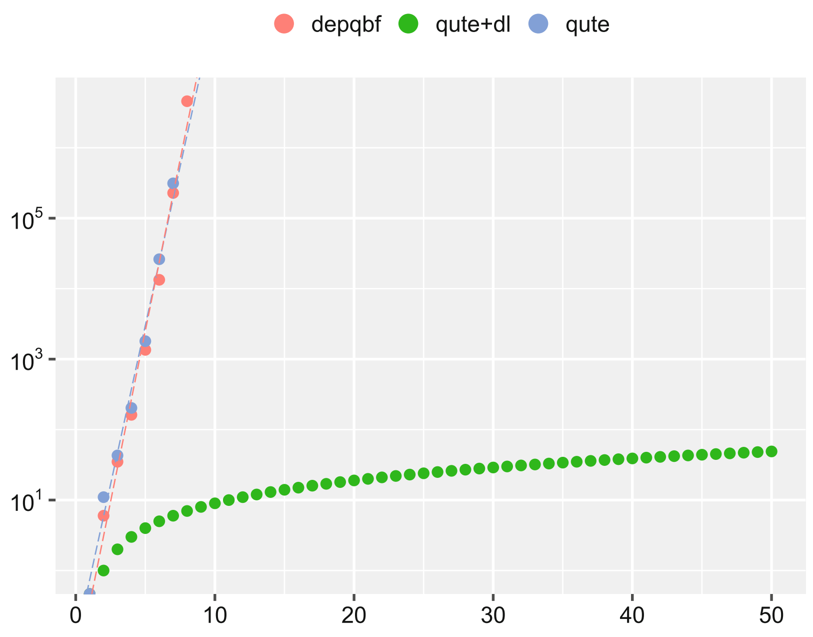

Fig. 3. Backtracks for instances $\mathtt{CR}_n$ based on the completion principle [13], as a function of n.

6 Discussion

In our experiments, Qute performed much better when presented with non-CNF input. In particular, dependency learning was most effective on the prenex non-CNF (QCIR) benchmark set, accounting for a 15% increase in the number of solved instances. Even for PCNF formulas, the best configuration(s)

[5] For sake of comparing with Qute in prefix mode, we disabled features recently added to DepQBF such as dynamic quantified blocked clause elimination [20] and oracle calls to the expansion-based solver Nenofex.

used tools for partially recovering circuit structure from CNF. This is consistent with the fact that Qute did not benefit from preprocessing nearly as much as other solvers, since preprocessing is known to adversely affect circuit reconstruction [11]. Whether this bias towards non-CNF representations is inherent to QCDCL with dependency learning or an artifact of other design choices implemented in our solver remains to be seen.

Dependency learning has several advantages over the use of dependency schemes within QCDCL: by retaining long-distance Q-resolution as its underlying proof system, the resulting algorithm is amenable to a simple correctness proof and supports linear-time strategy extraction. Moreover, our experiments indicate that proof search is less constrained with dependency learning, since typically only a small fraction of the dependencies computed by known dependency schemes has to be learned.

Sometimes, this additional freedom can be detrimental to performance, and a significant proportion of the overall runtime has to be spent on learning dependencies that are not spurious. To deal with such cases, we hope to find some middle ground between our current "tabula rasa" implementation of dependency learning and approaches that include too many spurious dependencies, by introducing a (small) set of dependencies that steer proof search in the right direction. For instance, Qute uses Tseitin conversion to obtain a set of initial clauses and terms from non-CNF (QCIR) instances. We found that assigning a Tseitin variable before a variable used in its definition often results in learning a dependency, so that it pays off to simply include dependencies of a Tseitin variable on the variables used in its definition from the start. For similar reasons, users may want to initialize D with pairs of variables that they know are dependent by construction. We hope to address this question by designing heuristics for "seeding" dependencies in a smart way as part of future work.

Acknowledgments. The authors thank Florian Lonsing for helpful discussions related to QCDCL. This research was kindly supported by FWF grants P27721 and W1255-N23.

References

1. Balabanov, V., Jiang, J.R.: Unified QBF certification and its applications. Formal Methods Syst. Des. **41**(1), 45–65 (2012)
2. Balabanov, V., Jiang, J.R., Janota, M., Widl, M.: Efficient extraction of QBF (counter)models from long-distance resolution proofs. In: Bonet, B., Koenig, S. (eds.) Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, Austin, Texas, USA, 25–30 January 2015, pp. 3694–3701. AAAI Press (2015)
3. Biere, A.: Resolve and expand. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 59–70. Springer, Heidelberg (2005). doi:10.1007/11527695_5
4. Lonsing, F., Biere, A.: A compact representation for syntactic dependencies in QBFs. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 398–411. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02777-2_37

5. Lonsing, F., Biere, A.: Integrating dependency schemes in search-based QBF solvers. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 158–171. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14186-7_14
6. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 101–115. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22438-6_10
7. Cadoli, M., Schaerf, M., Giovanardi, A., Giovanardi, M.: An algorithm to evaluate Quantified Boolean Formulae and its experimental evaluation. J. Autom. Reason. **28**(2), 101–142 (2002)
8. Egly, U., Lonsing, F., Widl, M.: Long-distance resolution: proof generation and strategy extraction in search-based QBF solving. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 291–308. Springer, Heidelberg (2013). doi:10.1007/978-3-642-45221-5_21
9. Faymonville, P., Finkbeiner, B., Rabe, M.N., Tentrup, L.: Encodings of bounded synthesis. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 354–370. Springer, Heidelberg (2017). doi:10.1007/978-3-662-54577-5_20
10. Giunchiglia, E., Narizzano, M., Tacchella, A.: Clause/term resolution and learning in the evaluation of Quantified Boolean Formulas. J. Artif. Intell. Res. **26**, 371–416 (2006)
11. Goultiaeva, A., Bacchus, F.: Recovering and utilizing partial duality in QBF. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 83–99. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39071-5_8
12. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 228–245. Springer, Cham (2016). doi:10.1007/978-3-319-40970-2_15
13. Janota, M.: On Q-resolution and CDCL QBF solving. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 402–418. Springer, Cham (2016). doi:10.1007/978-3-319-40970-2_25
14. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 114–128. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31612-8_10
15. Janota, M., Marques-Silva, J.: Solving QBF by clause selection. In: Yang, Q., Wooldridge, M. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, pp. 325–331. AAAI Press (2015)
16. Jordan, C., Klieber, W., Seidl, M.: Non-cnf QBF solving with QCIR. In: Darwiche, A. (ed.) Beyond NP, Papers from the 2016 AAAI Workshop. AAAI Workshops, vol. WS-16-05. AAAI Press (2016)
17. Kleine Büning, H., Karpinski, M., Flögel, A.: Resolution for quantified Boolean formulas. Inf. Comput. **117**(1), 12–18 (1995)
18. Klieber, W., Sapra, S., Gao, S., Clarke, E.: A non-prenex, non-clausal QBF solver with game-state learning. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 128–142. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14186-7_12
19. Lonsing, F., Schemes, D., Solving, Search-Based QBF: Theory and Practice. PhD thesis, Johannes Kepler University, Linz, Austria, April 2012
20. Lonsing, F., Bacchus, F., Biere, A., Egly, U., Seidl, M.: Enhancing search-based QBF solving by dynamic blocked clause elimination. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 418–433. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48899-7_29

21. Lonsing, F., Biere, A.: Nenofex: expanding NNF for QBF solving. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 196–210. Springer, Heidelberg (2008). doi:10.1007/978-3-540-79719-7_19
22. Malik, S., Zhang, L.: Boolean satisfiability from theoretical hardness to practical success. Commun. ACM **52**(8), 76–82 (2009)
23. Marques-Silva, J.P., Lynce, I., Malik, S.: Conflict-driven clause learning sat solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, pp. 131–153. IOS Press (2009)
24. Meel, K.S., Vardi, M.Y., Chakraborty, S., Fremont, D.J., Seshia, S.A., Fried, D., Ivrii, A., Malik, S.: Constrained sampling and counting: universal hashing meets SAT solving. In Darwiche, A. (ed.) Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA. AAAI Workshops, February 12, 2016, vol. WS-16-05. AAAI Press (2016)
25. Peitl, T., Slivovsky, F., Szeider, S.: Long distance Q-resolution with dependency schemes. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 500–518. Springer, Cham (2016). doi:10.1007/978-3-319-40970-2_31
26. Pulina, L.: The ninth QBF solvers evaluation - preliminary report. In: Lonsing, F., Seidl, M. (eds.) Proceedings of the 4th International Workshop on Quantified Boolean Formulas (QBF 2016). CEUR Workshop Proceedings, vol. 1719, pp. 1–13. CEUR-WS.org (2016)
27. Rabe, M.N., Tentrup, L.: CAQE: a certifying QBF solver. In: Kaivola, R., Wahl, T. (eds.) Formal Methods in Computer-Aided Design - FMCAD 2015, pp. 136–143. IEEE Computer Soc. (2015)
28. Samer, M., Szeider, S.: Backdoor sets of quantified Boolean formulas. J. Autom. Reason. **42**(1), 77–97 (2009)
29. Scholl, C., Pigorsch, F.: The QBF solver AIGSolve. In: Lonsing, F., Seidl, M. (eds.) Proceedings of the 4th International Workshop on Quantified Boolean Formulas (QBF 2016). CEUR Workshop Proceedings, vol. 1719, pp. 55–62. CEUR-WS.org (2016)
30. Slivovsky, F., Szeider, S.: Soundness of Q-resolution with dependency schemes. Theoret. Comput. Sci. **612**, 83–101 (2016)
31. Tentrup, L.: Non-prenex QBF solving using abstraction. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 393–401. Springer, Cham (2016). doi:10.1007/978-3-319-40970-2_24
32. Gelder, A.: Variable independence and resolution paths for quantified boolean formulas. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 789–803. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23786-7_59
33. Vizel, Y., Weissenbacher, G., Malik, S.: Boolean satisfiability solvers and their applications in model checking. Proc. IEEE **103**(11), 2021–2035 (2015)
34. Zhang, L., Malik, S.: Conflict driven learning in a quantified Boolean satisfiability solver. In: Pileggi, L.T., Kuehlmann, A. (eds.) Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2002, San Jose, California, USA, 10–14 November 2002, pp. 442–449. ACM/IEEE Computer Society (2002)