DepQBF: A Dependency-Aware QBF Solver, System Description

Florian Lonsing and Armin Biere

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

Abstract. We present DepQBF 0.1, a new search-based solver for quantified boolean formulae (QBF). It integrates compact dependency graphs to overcome the restrictions imposed by linear quantifier prefixes of QBFs in prenex conjunctive normal form (PCNF). DepQBF 0.1 was placed first in the main track of QBFEVAL'10 in a score-based ranking. We provide a general system overview and describe selected orthogonal features such as restarts and removal of learnt constraints.

1 Introduction

Many QBF solvers process input formulae in PCNF. When converting an arbitrary QBF into PCNF, structural properties such as quantifier nestings can be lost. The quantifier prefix of PCNFs imposes a total ordering on the variables, resulting in variable dependencies which have to be respected by QBF solvers.

Dependency schemes [17] are a general formalism for expressing dependencies in QBFs in terms of binary relations over the set of variables. In [17], the *standard dependency scheme* D^{std} was introduced which is (potentially) less restrictive than the total variable ordering in PCNFs: variables are not totally but partially ordered. This can grant QBF solvers more freedom in the solution process.

DepQBF, a search-based QBF solver, features a compact representation of D^{std} as a directed-acyclic graph (DAG) over equivalence classes of variables [13], called dependency DAG. In the following, we provide a system overview of DepQBF 0.1 [12] which participated in QBFEVAL'10. Additionally, we describe selected features such as strategies for restarts and removal of learnt constraints. Implementing these techniques was inspired by ideas from the SAT domain. To our knowledge, restarts have not been applied in QBF solving so far. Further details and references to related work on the application of dependency information in search-based QBF solving may be found in [14].

2 Overview

DepQBF consists of a loosely coupled *solver* and *dependency manager*. The solver implements the DPLL algorithm for QBF (QDPLL) with conflict-driven clause and solution-driven cube learning [7, 11, 19], called constraint learning (*analyze*

in Fig. 1). It operates on formulae in PCNF. The CNF part $\phi := \phi_{OCL} \wedge \phi_{LCL} \vee \phi_{LCU}$ is represented as *augmented CNF* [19], where ϕ_{OCL}, ϕ_{LCL} and ϕ_{LCU} are the sets of original clauses, learnt clauses and learnt cubes, respectively. During the search, learnt constraints are added to ϕ_{LCL} and ϕ_{LCU}, which are initially empty. Different from the approach in [19], DepQBF does *not* learn constraints containing complementary literals. Instead, the learning procedure follows [9]. A brief description of QDPLL may be found in [14]. For a comprehensive treatment of learning in QBF solving, we refer to [7, 9, 11, 19].

The dependency manager (DM) maintains the dependency DAG representing D^{std}, which is extracted from the formula given in PCNF (*init-DAG* in Fig. 1). DM keeps track of the set of *decision candidates (DC)* with respect to the solver's current (partial) assignment. A variable x is in set DC if it is sound to assign x as decision under the current assignment. In DepQBF, DC is updated incrementally and lazily based on equivalence classes in the dependency DAG before decision making and during backtracking [14].

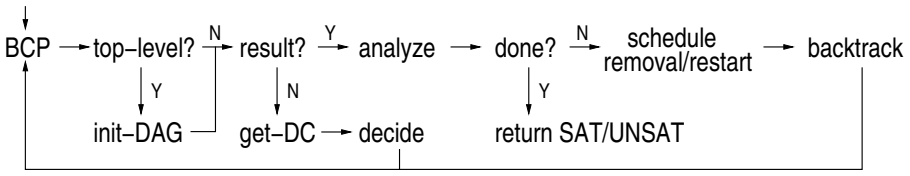

Fig. 1. DepQBF workflow.

For variables x and y, the solver queries DM to check whether y depends on x, written as $x \prec y$. Such checks are needed during learning and boolean constraint propagation (BCP), as pointed out in [14] in more detail. Generally, compact dependency DAGs in DepQBF allow efficient dependency checking.

Whenever the solver learns a unit clause or unit cube, it backtracks to the top-most decision level, called top-level (*top-level?* in Fig. 1). DM then re-initializes the dependency DAG, where clauses satisfied by top-level assignments are ignored. This can result in smaller DAGs and hence in smaller dependency sets, thus possibly influencing the solver positively. DAGs are initialized incrementally in DepQBF by inspecting clauses one after each other and updating classes using an efficient union-find data structure [18].

In the following sections, we describe selected features of DepQBF such as efficient detection of unit and pure literals, removal of learnt constraints, and restarts. Terminology and definitions used may be found in [14].

3 Boolean Constraint Propagation (BCP)

The core of QDPLL is propagation of unit and pure literals [3]. For the benchmark set used in QBFEVAL'08 (3326 formulae), we observed that 88% of total

assignments in DepQBF were due to BCP, 59% were unit, and 29% were pure literals. DepQBF implements watched data structures [5, 15] for BCP as follows.

3.1 Unit Literals

Two unassigned literals l_1 and l_2 are watched in each constraint C. For a literal l, $q(l) \in \{\forall, \exists\}$ denotes the quantifier of the variable of l. If C is a clause, then either (1) $q(l_1) = q(l_2) = \exists$ or (2) $q(l_1) = \forall$, $q(l_2) = \exists$ and $l_1 \prec l_2$. Otherwise, C is a cube and either (1) $q(l_1) = q(l_2) = \forall$ or (2) $q(l_1) = \exists$, $q(l_2) = \forall$ and $l_1 \prec l_2$.

If variable x is assigned in QDPLL, then the watchers of all constraints C where a literal of x is watched will be updated. Whenever a current watcher already disables C, i.e. satisfies clause C or falsifies cube C, then no update is made. Similarly, if a disabling literal l is found in C while updating watchers of C, then l is watched under the restrictions stated above. Note that dependency checking for watcher updates is needed in case (2) only.

3.2 Pure Literals

For a variable x, let $C(x), C(\overline{x}) \subseteq \phi_{OCL}$ denote the set of *original* (i.e. non-learnt) clauses in ϕ containing positive and negative literals of x, respectively. Two unsatisfied clauses $C_x \in C(x)$ and $C_{\overline{x}} \in C(\overline{x})$ are watched for each variable x [5, 8]. Let A be a variable assignment where, for an unassigned variable x, either C_x or $C_{\overline{x}}$ is satisfied under A. If there is no $C'_x \in C(x)$ (respectively $C'_{\overline{x}} \in C(\overline{x})$) currently unsatisfied under A, then x is considered to be pure.

Assigning a variable x will satisfy all clauses in $C(x)$ (or $C(\overline{x})$, respectively). Further, variables y watching clauses in $C(x)$ (or $C(\overline{x})$) will have to update their watcher C_y (or $C_{\overline{y}}$). Each variable x maintains two *notification lists*, one for x and one for $\overline{x}$, which exactly contain references to all variables y watching clauses in $C(x)$ (or $C(\overline{x})$). After some y has found a new watcher C'_y (respectively $C'_{\overline{y}}$), notification lists of variables occurring in the old and new watcher C_y and C'_y (or $C_{\overline{y}}$ and $C'_{\overline{y}}$) have to be updated.

Clause watching for pure literal detection as implemented in DepQBF was introduced in [5], yet without providing details of how to update watchers. The implementation of notification lists is optimal in the sense that no search is required to find all variables which need an update of their watcher list.

Similarly to the common technique of two-literal watching in SAT solvers [15], watched data structures for unit and pure literal detection as in DepQBF do *not* require additional maintenance work during backtracking.

The idea to ignore learnt constraints and only consider ϕ_{OCL} in pure literal detection was introduced in [8]. This approach can yield *spurious pure literals*, i.e. literals which are detected as pure according to ϕ_{OCL}, but not pure when also taking learnt constraints in ϕ_{LCL} and ϕ_{LCU} into account. Such literals are handled lazily as suggested in [8] by ignoring any learnt empty clause in ϕ_{LCL} or satisfied cube in ϕ_{LCU} containing a variable assigned as spurious pure literal. Further, unit literals triggered by spurious pure literals are ignored as well.

4 Variable Activities

After the solver has applied BCP until saturation, a variable is selected heuristically and assigned as decision. Variables are kept on a priority queue in descending order of their *activities*, which are implemented as described in [4].

Before a decision is made (*decide* in Fig. 1), the solver retrieves all decision candidates from the dependency manager and puts them on the priority queue (*get-DC* in Fig. 1). The variable with highest activity is then taken from the queue for the next decision. The content of the queue is *not* continuously kept up to date. Particularly, there might be variables which are already assigned as unit or pure literal or which are actually no decision candidates under the current assignment. When being removed, such variables are simply discarded.

5 Learnt Constraint Removal

Each time the solver encounters a conflict, i.e. an empty clause in $\phi_{OCL} \cup \phi_{LCL}$, or a solution, i.e. a satisfied cube in ϕ_{LCU} or an assignment satisfying all clauses in ϕ_{OCL}, one newly learnt constraint C is added to ϕ [9]. For conflicts, C is a clause and is produced by a sequence of clause resolutions. For solutions, C is a cube either produced by cube resolutions or a cube comprising a subset of the current assignment's literals sufficient to satisfy all clauses in ϕ_{OCL}. In DepQBF, C is always asserting [9, 14, 19], i.e. after backtracking to the asserting level, C will trigger a unit literal (*backtrack*, *BCP* in Fig. 1).

Learnt constraints in ϕ_{LCL} and ϕ_{LCU} are periodically removed according to the following strategy (*schedule removal* in Fig. 1). Set ϕ_{LCL} is empty before solving starts, i.e. $|\phi_{LCL}| = 0$, having an initial capacity $cap(\phi_{LCL}) = |\phi_{OCL}|$ based on the size of the original formula, but not less than 2500 and not more than 10000. Clauses are added to ϕ_{LCL} during learning. If $|\phi_{LCL}| = cap(\phi_{LCL})$, then half of the clauses in ϕ_{LCL} are removed. However, clauses which triggered a unit literal in the current assignment are never removed. Then, the capacity is increased by a constant $inc(\phi_{LCL}) = 500$ to $cap(\phi_{LCL}) + inc(\phi_{LCL})$. Removing cubes is handled analogously by ϕ_{LCU}, $|\phi_{LCU}|$, $cap(\phi_{LCU})$, $inc(\phi_{LCU}) = 500$. All constant values were chosen heuristically based on experimental data.

Sets ϕ_{LCL} and ϕ_{LCU} are implemented as doubly-linked lists. New constraints are always added to the head of the list. If a learnt clause (cube) C becomes empty (satisfied), triggers a unit literal during the search or is used in the deduction of learnt constraints, then C is moved to the head of the list. The idea is to make frequently used constraints appear at the head. When removing constraints, the lists are processed starting from the tail, thus removing least-recently used and possibly less important constraints. In contrast to the clause-move-to-front decision heuristics in HaifaSAT [6] this policy is only used for constraint removal, and in effect very similar to techniques based on clause activities as implemented in SAT solvers like e.g. [4, 10].

6 Restarts and Assignment Caching

DepQBF implements an inner-outer restart schedule inspired by PicoSAT [2] which is based on ideas from [1]. Separate inner and outer restart distances i and o, respectively, are maintained. Before solving starts, distances are initialized with $i = 100$ and $o = 10$, where values were determined experimentally. Assume that the solver performed $i - 1$ backtracks (i.e. conflict or solutions, see also Sec. 5). Before actually backtracking the ith time to the backtracking level b computed from the current learnt constraint (*schedule restart* in Fig. 1), the solver "restarts" (called inner restart) by backtracking to the largest decision level d of a universally quantified decision variable x, provided that $d < b$. Otherwise, it backtracks to b as usual. If there is no such x, then d is the top-level by definition. After each inner restart, the inner distance i is incremented by 10. Then, the next inner restart happens after i backtracks. After o inner restarts have been carried out, i is reset to its initial value 100 and o is incremented by 5 (called an outer restart). Thus, the next outer restart happens after o inner restarts. From 568 benchmarks used in the main track of QBFEVAL'10, DepQBF 0.1 solves 360 (370) instances in 352.33 (337.10) seconds average run time including time-outs when disabling (enabling) restarts.[1]

As many SAT solvers, DepQBF combines assignment caching [16] with restarts. Initially, the assignment cache $cached(x)$ is empty for all variables x. Each time a variable x is assigned, the assigned value is stored in $cached(x)$, replacing the old entry. Decision variables are always assigned the value in $cached(x)$, if not empty, otherwise the assigned value is chosen heuristically.

7 Conclusion

We have presented DepQBF, a search-based QBF solver which integrates dependency schemes [17] as compact dependency graphs. We described selected features such as watched data structures for BCP, removal of learnt constraints and restarts. These techniques are largely independent from the specific system architecture of DepQBF and therefore may be relevant for arbitrary QBF solvers. Strategies for restarts and constraint removal are our main contributions.

DepQBF version 0.1 [12] participated in QBFEVAL'10 [2] among 11 other, search- and elimination-based solvers. In the main track of the competition, it was placed first according to a score-based ranking. As the only solver, DepQBF solved all 136 formulae from the newly submitted benchmark suite *mqm*[3].

We believe that many implementation-related techniques applied in SAT solvers could also be successfully integrated in search-based QBF solvers. As future work, we want to improve DepQBF in this respect.

[1] Setup: Ubuntu 9.04, Intel® Q9550@2.83 GHz, 7 GB/900 sec. mem and time limit.
[2] `http://www.qbflib.org/index_eval.php`
[3] `http://www.qbflib.org/family_solvers.php?idFamily=723&year=2010`

References

1. 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.
2. A. Biere. PicoSAT Essentials. *JSAT*, 4(2-4):75–97, 2008.
3. M. Cadoli, A. Giovanardi, and M. Schaerf. An Algorithm to Evaluate Quantified Boolean Formulae. In *AAAI/IAAI*, pages 262–267, 1998.
4. N. Eén and N. Sörensson. An Extensible SAT-Solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.
5. I. P. Gent, E. Giunchiglia, M. Narizzano, A. G. D. Rowley, and A. Tacchella. Watched Data Structures for QBF Solvers. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 25–36. Springer, 2003.
6. R. Gershman and O. Strichman. HaifaSat: A SAT Solver Based on an Abstraction/Refinement Model. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:33–51, 2008.
7. E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for Quantified Boolean Logic Satisfiability. In *AAAI/IAAI*, pages 649–654, 2002.
8. E. Giunchiglia, M. Narizzano, and A. Tacchella. Monotone Literals and Learning in QBF Reasoning. In M. Wallace, editor, *CP*, volume 3258 of *LNCS*, pages 260–273. Springer, 2004.
9. 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.
10. E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust SAT-Solver. In *Proc. DATE*, pages 142–149. IEEE Computer Society, 2002.
11. 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.
12. F. Lonsing. DepQBF 0.1 Source Code, 2010. `http://fmv.jku.at/depqbf/`.
13. F. Lonsing and A. Biere. A Compact Representation for Syntactic Dependencies in QBFs. In O. Kullmann, editor, *SAT*, volume 5584 of *LNCS*, pages 398–411. Springer, 2009.
14. F. Lonsing and A. Biere. Integrating Dependency Schemes in Search-Based QBF Solvers. In O. Strichman and S. Szeider, editors, *SAT (accepted for publication)*, LNCS. Springer, 2010.
15. 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.
16. K. Pipatsrisawat and A. Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In J. Marques-Silva and K. A. Sakallah, editors, *SAT*, volume 4501 of *LNCS*, pages 294–299. Springer, 2007.
17. M. Samer and S. Szeider. Backdoor Sets of Quantified Boolean Formulas. *Journal of Automated Reasoning (JAR)*, 42(1):77–97, 2009.
18. R. E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM*, 22(2):215–225, 1975.
19. 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.