Journal on Satisfiability, Boolean Modeling, and Computation 11 (2019) 261-272

Qute in the QBF Evaluation 2018

Tomáš Peitl* peitl@ac.tuwien.ac.at
Friedrich Slivovsky† fs@ac.tuwien.ac.at
Stefan Szeider sz@ac.tuwien.ac.at
Algorithms and Complexity Group,
TU Wien
Austria

Abstract

QUTE is a solver for Quantified Boolean Formulas (QBFs) based on Quantified Conflict-Driven Constraint Learning (QCDCL). Its main distinguishing feature is dependency learning, a lazy technique for relaxing restrictions on the order of variable assignments imposed by nested quantifiers. In this short note, we describe the configurations of QUTE submitted to QBFEval'18, along with the parameter tuning process that went into creating them.

KEYWORDS: *QBF*

Submitted November 2018; revised September 2019; published September 2019

1. Introduction

Conflict-Driven Clause Learning (CDCL) is the dominant architecture of modern (complete) SAT solvers [14]. Efficient implementations of CDCL combine fast unit propagation and clever heuristics (such as VSIDS [15]) for branching with clause learning. The success of CDCL for SAT has lead to a generalization of this algorithm to the satisfiability problem (or *evaluation problem*) of Quantified Boolean Formulas (QBFs) under the name of Quantified CDCL (QCDCL) [7, 19].

In spite of significant improvements over the years, it is fair to say that QCDCL has not resulted in a breakthrough in QBF solving in the way that CDCL has lead to a breakthrough in SAT solving. Two principal obstacles to lifting CDCL to QBF can be identified:

1. Prenex Conjunctive Normal Form (PCNF), the default encoding format for Quantified Boolean Formulas, is biased towards proving unsatisfiability and does not do justice to the symmetry of truth and falsity in QBF satisfiability, which can lead to unnecessarily long proofs of satisfiability (and thus unnecessarily long solving times for satisfiable formulas) [1].

2. Heuristics for branching must respect the order of quantification in the prefix (or the nesting of quantifiers in non-prenex formulas) and therefore cannot unleash their full potential. In the worst case, heuristics are forced into a fixed order of assignments.

∗ Supported by the Austrian Science Fund (FWF) under grant W1255-N23
† Supported by the Austrian Science Fund (FWF) under grant P 27721.

The first of these problems—which is not unique to QCDCL solving—has been addressed by the introduction of a new encoding standard for QBFs called QCIR [11] and techniques for *dual propagation* [12, 9]. A number of solutions have been proposed for the second problem, most of which are subsumed by the application of *dependency schemes* [17]. While the use of dependency schemes can speed up solving times [2] and has the potential to dramatically decrease proof sizes in some cases [4], it has distinct disadvantages, such as the fact that it changes the underlying proof system and thus complicates strategy extraction.

We recently proposed *dependency learning* as an alternative solution to the second problem [16]. QCDCL with dependency learning (for details see Section 2) maintains set of dependencies (which is empty initially) and permits variable assignments in any order that is consistent with these dependencies. A dependency is added whenever clause learning fails due to an assignment that does not respect the quantifier prefix. This idea has been implemented in Qute (`https://github.com/perebor/qute`).

The remainder of this note is structured as follows. In Section 2, we describe the core algorithm underlying Qute as well as some implementation details, with an emphasis on the changes required from "vanilla" QCDCL. Section 3 describes the submission to QBFEval'18 based on Qute. Section 3.1 details the preprocessing pipeline used for the submission to the Prenex Conjunctive Normal Form (PCNF) track. Section 3.2 describes the automated configuration process we used to tune the parameters of our submissions. The (non-default) parameter settings for all submitted configurations are listed in Appendix A.

2. Qute in a Nutshell

We first present the nuts and bolts of QCDCL following the presentation of Peitl, Slivovsky, and Szeider [16]. QCDCL simultaneously works on two dual sets of *constraints*: a set of *clauses* and a set of *terms*. Each clause is a disjunction of literals and each term (or *cube*) is a conjunction of literals. One can think of clauses as encoding obligations of the existential player and of terms as encoding obligations of the universal player. For PCNF formulas, the set of clauses initially consists of clauses in the matrix and the set of terms is populated on the fly using the *model generation* rule [7]. For QCIR formulas, Qute uses Tseitin conversion to obtain initial sets of clauses and terms.

Starting from this initial set of constraints, QCDCL generates ("learns") new constraints until it learns an empty constraint, outputting TRUE if the empty term has been learned and FALSE if the empty clause has been learned. The algorithm is sound because every clause learned by QCDCL can be derived from the input formula in the Q-resolution proof system and every term learned by QCDCL can be derived by the dual proof system known as Q-consensus [7, 6].

QCDCL solving can be seen as proceeding in rounds. The solver maintains a partial truth assignment σ of the given formula's variables (called the *trail*) which is extended in each round by quantified Boolean constraint propagation (QBCP) and—possibly—branching. QBCP consists of the exhaustive application of universal reduction in combination with unit assignments. Universal reduction is an operation from Q-resolution that removes a universal variable u from a clause C if the clause C does not contain an existential variable e such that e is quantified after u in the prefix.

QBCP reports a clause C as falsified if it is not satisfied by the current trail σ and universal reduction can be applied to $C[\sigma]$ to obtain the empty clause. A clause C is *unit* under σ if it is not satisfied and universal reduction applied to $C[\sigma]$ yields a clause (ℓ), for some existential literal ℓ. If (ℓ) is a unit clause then the assignment σ has to be extended by ℓ in order not to falsify (ℓ). (The dual versions of these notions for propagation of terms are defined in a straightforward way.) If several clauses or terms are unit under the current trail assignment, 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 where $Q'y$ precedes Qx in the quantifier prefix and $Q' \neq Q$ has already been assigned. The resulting assignment can be partitioned into so-called *decision levels*. The decision level of an assignment σ is the number of literals in σ that were assigned by branching. 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 (we focus on clauses, the process for terms is dual), called the *conflict clause*. The solver finds the existential literal in C that was assigned last by QBCP, and the antecedent clause R responsible for this assignment. A new constraint is derived by resolving C and R and applying universal reduction. This is done repeatedly until the resulting constraint C is *asserting*. A clause (term) C is asserting if there is a unique existential (universal) literal $\ell \in C$ with maximum decision level among literals in C, its decision level is greater than 0, the corresponding decision variable is existential (universal), and every universal (existential) variable $y \in \mathit{var}(C)$ such that y precedes $\mathit{var}(\ell)$ in the quantifier prefix 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 it reaches a decision level computed from the learned constraint.

We now describe how QCDCL is modified in QUTE to support dependency learning. The solver maintains a set D of dependencies consisting of pairs of variables that is used to generalize both QBCP and the decision rule:

- QBCP performs 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).

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

This is how DEPQBF uses the dependency relation D computed by a dependency scheme in propagation and decisions [2]. Unlike DEPQBF, QUTE does *not* use the generalized reduction rules during conflict analysis and instead sticks to the prefix order. As a consequence, it cannot always construct a learned constraint. Such cases are dealt with in lines 9 through 12 of ANALYZECONFLICT (Algorithm 1): EXISTSRESOLVENT(*constraint*, *reason*, *pivot*) determines whether *constraint* and *reason* can be resolved. If this is not the case, there has to

Algorithm 1 Conflict Analysis with Dependency Learning

```
procedure analyzeConflict(conflict)
    constraint = getConflictConstraint(conflict)
    while not asserting(constraint) do
        pivot = getPivot(constraint)
        reason = getAntecedent(pivot)
        if existsResolvent(constraint, reason, pivot) then
            constraint = resolve(constraint, reason, pivot)
            constraint = reduce(constraint)
        else
            illegal_merges = illegalMerges(constraint, reason, pivot)
            D = D ∪ { (v, pivot) : v ∈ illegal_merges }
            return none, decisionLevel(pivot)
        end if
    end while
    btlevel = getBacktrackLevel(constraint)
    return constraint, btlevel
end procedure
```

be a variable v (universal for clauses, existential for terms) satisfying the following condition: v precedes *pivot* in the quantifier prefix and there exists a literal $\ell \in \mathit{constraint}$ with $\mathit{var}(\ell) = v$ and $\overline{\ell} \in \mathit{reason}$. 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 (by QBCP).

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, its decision level is greater than 0, the corresponding decision variable is existential (universal), and every universal (existential) variable $y \in \mathit{var}(S)$ such that $(y, \mathit{var}(\ell)) \in D$ is assigned (again, this corresponds to the definition of asserting constraints used in DepQBF [13, 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.

Representation of Learned Dependencies. Qute represents the set D of learned dependencies as follows. For each variable y, we record the set $D(y) = \{ x : (x, y) \in D \}$ of variables it depends on. These dependencies are relevant for propagation and determining variables for branching:

- If a clause C contains an unassigned existential variable e and an unassigned universal variable u such that $(u, e) \in D$, then generalized universal reduction cannot simplify C to a unit clause (or the empty clause) under the current trail assignment and literals over u and e can be used as watched literals for the clause C.

- A variable y is eligible for branching as soon as every variable $x \in D(y)$ has been assigned. In order to determine whether that is the case, we adopt a simple watcher scheme. We pick some unassigned variable $x \in D(y)$ that becomes the "watched dependency" of y. When x is assigned, we try to find a new watched dependency for y. If we cannot find a suitable variable, x remains the watcher of y. Testing whether a variable is eligible for branching thus boils down to checking whether its watched dependency (if there is one) is assigned.

For the purposes of searching for a watched dependency, each set $D(y)$ is internally represented as an array. Since maintenance of watched literals requires efficient membership tests for $D(y)$, we additionally store $D(y)$ as a hash set. This improves performance in cases where individual sets $D(y)$ grow large. At the same time, it does not significantly increase memory consumption since the size of D tends to remain small overall.

3. The Submissions

We submitted to two tracks of QBFEval'18: the *Prenex CNF* (PCNF) track and the *Prenex non-CNF* (QCIR) track. The corresponding configurations are unchanged from our submissions to QBFEval'17. For the QCIR track, we use QUTE as a standalone solver. One of the configurations (*hybrid*) is a sequential portfolio that splits solving time between the four best performing configurations found by automated parameter configuration (see Appendix A.2). The setup for the PCNF track is a bit more complicated and involves several preprocessing steps.

3.1 Preprocessing for PCNF

It has been observed that CNF is inherently biased towards proving unsatisfiability and can thus be detrimental to proving satisfiability of QBFs [1]. In certain cases, a more symmetric circuit representation of a PCNF formula can be obtained by detecting clauses and variables introduced by Tseitin transformation [8]. We use QCIR-CONV[1.] to perform partial circuit reconstruction of this kind. When this approach works well, it results in a more compact representation of the input formula in QCIR that can be passed to QUTE. When it fails to detect a significant number of gate definitions, the output of QCIR-CONV essentially corresponds to the original PCNF and the set of initial terms generated by QUTE's QCIR interface tends to slow down propagation.

Clausal preprocessing techniques for QBF as implemented in BLOQQER and HQSPRE are very effective and can solve many instances on their own or significantly reduce their size [3, 18]. On the downside, they may remove clauses or variables that partial circuit reconstruction relies on to identify gate definitions.

Our submission combines clausal preprocessing and partial circuit reconstruction in the following way (the values of the variables mentioned below were determined by automated parameter configuration and are listed in Appendix A.1):

1. We first run HQSPRE [18] (with command-line options `--hidden` 2 `--univ_exp` 2) for `hqspre-timeout` seconds to preprocess PCNF formulas.

2. Next, we run QCIR-CONV on the preprocessed formula (or on the original formula in case HQSPRE did not terminate) for `qcir-conv-timeout` seconds.

1. http://www.cs.cmu.edu/~wklieber/qcir-conv/qcir-conv.py

3. If QCIR-conv terminates and the ratio of input variables of the CNF to input gates of the resulting QCIR instance is above `qcir-input-reduction`, Qute is run on the QCIR instance. Otherwise, we run it on the (preprocessed) PCNF instance.

HQSPre has an option for preserving gate definitions during preprocessing so as to not interfere with circuit reconstruction, but we found that our pipeline worked better with this option turned off.

3.2 Automated Parameter Configuration

Qute comes with a number of command-line parameters that can significantly affect performance. Since these parameters interact in ways that are difficult to understand, finding good settings is a challenge. The situation is exacerbated by the fact that the performance of a configuration may vary wildly depending on the instance family. We tried to deal with this challenge by using SMAC[2.] [10] to automatically configure most command-line parameters of Qute. More specifically, the following parameters were up to configuration:

- The initial limit on the number of learned clauses (terms) (`--initial-clause-DB-size` and `--initial-term-DB-size`).
- The number of constraints the maximum size of the clause (term) database is increased by every time we reach the limit (`--clause-DB-increment` and `--term-DB-increment`).
- The percentage of clauses (terms) deleted upon hitting the limit of the learned clause (term) database (`--clause-removal-ratio` and `--term-removal-ratio`).
- The decay factor for constraint activity values (`--constraint-activity-decay`) used in constraint cleaning and the increment (`--constraint-activity-inc`) activity values are bumped by whenever a constraint is used in propagation or learning. In addition to deleting a fraction of learned constraint with lowest activity scores, Qute has an option for deleting all constraints with activities below a certain threshold (`--use-activity-threshold`).[3.]
- The decision heuristic (`--decision-heuristic`) limited to variants of VSIDS or Variable-Move-To-Front (VMTF), as well as some additional parameters for VSIDS (`--var-activity-decay`, `--var-activity-inc`).
- The phase heuristic that decides which value a decision variable is assigned first (`--phase-heuristic`), and whether or not to use phase saving (`--no-phase-saving`).
- A parameter deciding whether to use dependency learning (`--dependency-learning`).
- Parameters related to an inner-outer restart scheme (`--inner-restart-distance`, `--outer-restart-distance`, `--restart-multiplier`) and a parameter that determines whether to use restarts at all (`--no--restarts`).

2. https://github.com/automl/SMAC3
3. This idea is taken from MiniSAT [5].

- A parameter (only relevant for PCNF input) that determines whether (weighted) model generation is used (`--model-generation`), as well as options for model generation (such as `--variable-weight-exponent`, `--variable-weight-scaling-factor`, `--variable-weight-universal-penalty`).

We separately configured QUTE for PCNF and QCIR input. For PCNF we additionally configured the values of the three variables `hqspre-timeout`, `qcir-conv-timeout`, and `qcir-input-reduction` mentioned above. In both cases, we used the benchmark sets from QBFEval'16 with a timeout of 900 seconds and PAR10 as the target metric. PAR10 is the cumulative runtime with a penalty factor of 10 for unsolved instances (in our case, an unsolved instance would increase PAR10 by 9000). The parameter settings obtained by SMAC are listed in Appendix A.

References

[1] Carlos Ansótegui, Carla P. Gomes, and Bart Selman. The Achilles' heel of QBF. In Manuela M. Veloso and Subbarao Kambhampati, editors, *The Twentieth National Conference on Artificial Intelligence - AAAI 2005*, pages 275–281. AAAI Press / The MIT Press, 2005.

[2] Armin Biere and Florian Lonsing. Integrating dependency schemes in search-based QBF solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing - SAT 2010*, **6175** of *Lecture Notes in Computer Science*, pages 158–171. Springer Verlag, 2010.

[3] Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *International Conference on Automated Deduction - CADE 23*, **6803** of *Lecture Notes in Computer Science*, pages 101–115. Springer Verlag, 2011.

[4] Joshua Blinkhorn and Olaf Beyersdorff. Shortening QBF proofs with dependency schemes. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017*, **10491** of *Lecture Notes in Computer Science*, pages 263–280. Springer Verlag, 2017.

[5] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, **2919** of *Lecture Notes in Computer Science*, pages 502–518. Springer Verlag, 2003.

[6] Uwe Egly, Florian Lonsing, and Magdalena Widl. Long-distance resolution: Proof generation and strategy extraction in search-based QBF solving. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - LPAR 2013*, **8312** of *Lecture Notes in Computer Science*, pages 291–308. Springer Verlag, 2013.

[7] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. Clause/term resolution and learning in the evaluation of Quantified Boolean Formulas. *J. Artif. Intell. Res.*, **26**:371–416, 2006.

[8] Alexandra Goultiaeva and Fahiem Bacchus. Recovering and utilizing partial duality in QBF. In Matti Järvisalo and Allen Van Gelder, editors, *Theory and Applications of Satisfiability Testing - SAT 2013*, **7962** of *Lecture Notes in Computer Science*, pages 83–99. Springer Verlag, 2013.

[9] Alexandra Goultiaeva, Martina Seidl, and Armin Biere. Bridging the gap between dual propagation and CNF-based QBF solving. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13*, pages 811–814. EDA Consortium San Jose, CA, USA / ACM DL, 2013.

[10] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In Carlos A. Coello Coello, editor, *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers*, **6683** of *Lecture Notes in Computer Science*, pages 507–523. Springer Verlag, 2011.

[11] Charles Jordan, Will Klieber, and Martina Seidl. Non-cnf QBF solving with QCIR. In Adnan Darwiche, editor, *Beyond NP, Papers from the 2016 AAAI Workshop.*, **WS-16-05** of *AAAI Workshops*. AAAI Press, 2016.

[12] William Klieber, Samir Sapra, Sicun Gao, and Edmund M. Clarke. A non-prenex, non-clausal QBF solver with game-state learning. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing - SAT 2010*, **6175** of *Lecture Notes in Computer Science*, pages 128–142. Springer Verlag, 2010.

[13] Florian Lonsing. *Dependency Schemes and Search-Based QBF Solving: Theory and Practice*. PhD thesis, Johannes Kepler University, Linz, Austria, April 2012.

[14] Sharad Malik and Lintao Zhang. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, **52**(8):76–82, 2009.

[15] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[16] Tomás Peitl, Friedrich Slivovsky, and Stefan Szeider. Dependency learning for QBF. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017*, **10491** of *Lecture Notes in Computer Science*, pages 298–313. Springer Verlag, 2017.

[17] Marko Samer and Stefan Szeider. Backdoor sets of quantified Boolean formulas. *Journal of Automated Reasoning*, **42**(1):77–97, 2009.

[18] Ralf Wimmer, Sven Reimer, Paolo Marin, and Bernd Becker. Hqspre - an effective preprocessor for QBF and DQBF. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017*, **10205** of *Lecture Notes in Computer Science*, pages 373–390, 2017.

[19] Lintao Zhang and Sharad Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified Boolean formula evaluation. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, **2470** of *Lecture Notes in Computer Science*, pages 200–215. Springer Verlag, 2002.

Appendix A. Parameter Settings for Each Configuration

Below, we list the parameter settings for each configuration. Whenever a parameter is not mentioned explicitly, the default was used.

A.1 Prenex CNF Track

- (*default*) This configuration used the following parameters for preprocessing:

`hqspre-timeout`	400
`qcir-conv-timeout`	90
`qcir-input-reduction`	2

  Default parameters were used for Qute with the exception of `--model-generation` weighted.

- (*opt500*) This is the best configuration found by SMAC:

`hqspre-timeout`	400
`qcir-conv-timeout`	30
`qcir-input-reduction`	1.356
`--initial-clause-DB-size`	4000
`--initial-term-DB-size`	4000
`--term-DB-increment`	500
`--term-removal-ratio`	0.706
`--clause-DB-increment`	1500
`--clause-removal-ratio`	0.187
`--constraint-activity-decay`	0.95
`--constraint-activity-inc`	-9.07
`--decision-heuristic`	0
`--dependency-learning`	off
`--inner-restart-distance`	200
`--outer-restart-distance`	800
`--restart-multiplier`	4.38
`--model-generation`	weighted
`--variable-weight-exponent`	1.692
`--variable-weight-scaling-factor`	0.721
`--variable-weight-universal-penalty`	0.526
`--phase-heuristic`	false

- (*random*) This configuration was obtained by manually tweaking *opt500* and increasing `hqspre-timeout` to 450 seconds.

A.2 Prenex non-CNF Track

- (*opt617*)

`--term-DB-increment`	1500
`--term-removal-ratio`	0.467
`--clause-DB-increment`	1500
`--clause-removal-ratio`	0.652
`--constraint-activity-decay`	0.99
`--constraint-activity-inc`	5.38
`--decision-heuristic`	2
`--dependency-learning`	all
`--no-restarts`	true
`--no-phase-saving`	true
`--phase-heuristic`	watcher
`--var-activity-decay`	0.827
`--var-activity-inc`	6.61

- (*opt993*)

`--initial-clause-DB-size`	4000
`--initial-term-DB-size`	4000
`--term-DB-increment`	500
`--term-removal-ratio`	0.806
`--clause-DB-increment`	1500
`--clause-removal-ratio`	0.188
`--constraint-activity-decay`	0.872
`--constraint-activity-inc`	0.443
`--decision-heuristic`	1
`--dependency-learning`	outer
`--no-restarts`	true
`--phase-heuristic`	qtype
`--var-activity-decay`	0.786
`--var-activity-inc`	7.746
`--use-activity-threshold`	true

- (*hybrid*) This submission implements a sequential portfolio that runs each of four configurations for a limited amount of time. Two of these are the configurations *opt617* and *opt993* described above. The remaining two are the configurations *seq1* and *seq2* listed below. The configuration *hybrid* first runs *seq1* for 23 seconds, then *opt617* for 119 seconds, afterwards *seq2* for 385 seconds, and finally *opt993* for the time remaining until timeout.

- (*seq1*)

`--initial-clause-DB-size`	4000
`--initial-term-DB-size`	4000
`--term-DB-increment`	1500
`--term-removal-ratio`	0.745
`--clause-DB-increment`	500
`--clause-removal-ratio`	0.550
`--constraint-activity-decay`	0.929
`--constraint-activity-inc`	5.053
`--decision-heuristic`	4
`--dependency-learning`	outer
`--restart--multiplier`	3.280
`--inner-restart-distance`	400
`--outer-restart-distance`	800
`--phase-heuristic`	qtype
`--no-phase-saving`	true
`--var-activity-decay`	0.465
`--var-activity-inc`	2.080

- (*seq2*)

`--term-DB-increment`	500
`--term-removal-ratio`	0.147
`--clause-DB-increment`	250
`--clause-removal-ratio`	0.766
`--constraint-activity-decay`	0.878
`--constraint-activity-inc`	4.361
`--decision-heuristic`	4
`--dependency-learning`	all
`--restart--multiplier`	4.164
`--inner-restart-distance`	400
`--outer-restart-distance`	100
`--phase-heuristic`	watcher
`--no-phase-saving`	true
`--var-activity-decay`	0.766
`--var-activity-inc`	3.548
`--use-activity-threshold`	true