The CQuest SAT Solver

Inês Lynce and João Marques-Silva
IST/INESC-ID, Technical University of Lisbon, Portugal
{ines,jpms}@sat.inesc-id.pt

Abstract

This paper describes the CQuest SAT solver. This solver is essentially a translation from Java to C++ of the JQuest2 SAT solver [4]. All of the Quest generation solvers [3, 4] were mainly inspired in Grasp [5] and Chaff [6]. These solvers perform backtrack search enhanced with clause recording. In addition, Chaff's lazy data structures are implemented in CQuest, in order to obtain fast unit propagation, namely on (large) recorded clauses. BerkMin [2] also inspired the CQuest SAT solver with respect to the variable branching heuristic. Experimental results have proved that CQuest is a competitive solver, specially when considering industrial benchmarks.

1 Introduction

In recent years, SAT has successfully found a large number of significant applications. SAT has also been the subject of intensive research. Improvements in SAT solvers have been characterized by a few significant paradigm shifts. First, GRASP [5] very successfully proposed using clause recording and non-chronological backtracking in SAT solvers. More recently, search restart strategies have been shown to be extremely effective for solving real-world problem instances [1]. The most recent paradigm shift was observed in Chaff [6], that proposed several significant new ideas on how to efficiently implement backtrack search SAT algorithms. A few years ago, BerkMin [2] has improved Chaff's ideas on clauses' management and branching heuristics.

2 CQuest SAT Solver

Over the years a large number of algorithms has been proposed for SAT. SAT algorithms can be characterized as being either *complete* or *incomplete*. Complete algorithms can establish unsatisfiability if given enough CPU time; incomplete algorithms cannot.

Among the different algorithms, we believe backtrack search to be the most robust approach for solving hard, structured, real-world instances of SAT. Most backtrack search SAT solvers are conceptually composed of three main stages: the decision stage, the deduction stage and the diagnosis stage. The decision stage elects the variable and value to assign at each branching step of the search process. The deduction stage identifies necessary assignments as a result of each selected variable assignment. The diagnosis stage implements the backtracking step of the algorithm.

2.1 Decision Stage

CQuest implements a mix of Chaff's [6] and BerkMin's [2] heuristic. Counters on variables are updated whenever a new clause is created (like in Chaff) or a new conflict if found (like in BerkMin). The next variable to be assigned is selected based on the values on the counters.

Chaff's heuristic was introduced due to the new lazy data structures. It selects the literal that appears most frequently over all clauses, i.e. the metrics are updated when a new clause is recorded. In addition, BerkMin's heuristic increases counters for variables on clauses involved in conflicts.

2.2 Deduction Stage

CQuest deduction stage consists in Boolean Constraint Propagation (BCP). Efficiency of BCP depends mostly on the data structures used to represent literals and clauses. CQuest implements the lazy data structures first introduced in Chaff [6]. However, small clauses (unit, binary and ternary) are implemented using dedicated data structures.

Lazy data structures are an improvement on standard implementations, where literal counters are associated with each clause, to keep track of unsatisfied, satisfied and unit clauses. Lazy implementations are characterized by each variable keeping a reduced set of clauses' references. In Chaff, for each clause there are solely two references to literals, which are said to be *watched*. Since such literals are the *last* to be assigned, unit and unsatisfied clauses are detected by examining the references in each clause. A significant feature of watched literals is that no updating takes place in the backtrack step.

2.3 Diagnosis Stage

The diagnosis stage of CQuest consists essentially in the clause recording mechanism in the presence of conflicts [5, 6], allowing the search to backtrack non-chronologically.

In CQuest, whenever a conflict is found a new clause is recorded to explain and prevent identified conflicting conditions. CQuest uses the first UIP scheme for clause recording, which is considered to be the most competitive [7]. To prevent memory exhaustion, large recorded clauses are deleted opportunistically. Recorded clauses are also used for computing the backtracking decision level, which is defined as the highest decision level of all variable assignments of the literals in each newly recorded clause.

In addition, CQuest performs search restarts whenever a limit on the number of backtracks is reached. This limit increases after each restart to guarantee completeness [1].

3 Conclusions

We have described the CQuest SAT solver that has been submitted to the SAT Competition 2004. The main characteristics of CQuest SAT solver can be summarized as follows:

- CQuest is a complete non-randomized SAT solver implemented in C++.
- CQuest performs backtrack search.
- CQuest branching heuristic is inspired on Chaff's and BerkMin's heuristics.
- CQuest data structures are inspired in Chaff's watched literals, although small clauses have dedicated data structures.
- CQuest records clauses after finding a conflict (and stops at the first UIP). Recorded clauses allow the search to backtrack non-chronologically. Large recorded clauses are discarded opportunistically.
- Search restarts are applied when a limit on the number of backtracks is reached. After each restart, this limit is increased to guarantee completeness.

References

[1] L. Baptista and J. P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *CP'00*.

[2] E. Goldberg and Y. Novikov. BerkMin: a fast and robust sat-solver. In *DATE'02*.

[3] I. Lynce and J. P. Marques-Silva. Efficient data structures for backtrack search SAT solvers. In *SAT'02*.

[4] I. Lynce and J. P. Marques-Silva. On implementing more efficient SAT data structures. In *SAT'03*.

[5] J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *ICCAD'96*.

[6] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *DAC'01*.

[7] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD'01*.