Heuristic Backtracking Algorithms for SAT

A. Bhalla, I. Lynce, J.T. de Sousa and J. Marques-Silva
IST/INESC-ID/CEL,
Technical University of Lisbon, Portugal
{ateet,ines,jts,jpms}@sat.inesc.pt

Abstract

In recent years backtrack search SAT solvers have been the subject of dramatic improvements. These improvements allowed SAT solvers to successfully replace BDDs in many areas of formal verification, and also motivated the development of many new challenging problem instances, many of which too hard for the current generation of SAT solvers. As a result, further improvements to SAT technology are expected to have key consequences in formal verification. The objective of this paper is to propose heuristic approaches to the backtrack step of backtrack search SAT solvers, with the goal of increasing the ability of the SAT solver to search different parts of the search space. The proposed heuristics to the backtrack step are inspired by the heuristics proposed in recent years for the branching step of SAT solvers, namely VSIDS and some of its improvements. The preliminary experimental results are promising, and motivate the integration of heuristic backtracking in state-of-the-art SAT solvers.

1 Introduction

Propositional Satisfiability is a well-known NP-complete problem, with theoretical and practical significance, and with extensive applications in many fields of Computer Science and Engineering, including Artificial Intelligence and Electronic Design Automation.

Current state-of-the-art SAT solvers incorporate sophisticated pruning techniques as well as new strategies on how to organize the search. Effective search pruning techniques are based, among others, on nogood learning and dependency-directed backtracking [12] and backjumping [4], whereas recent effective strategies introduce variations on the organization of backtrack search. Examples of such strategies are weak-commitment search [13], search restarts [6] and random backtracking [7].

Advanced techniques applied to backtrack search SAT algorithms have achieved remarkable improvements [1, 9, 10], having been shown to be crucial for solving hard instances of SAT obtained from real-world applications. Moreover, and from a practical perspective, the most effective algorithms are complete, and so able to prove what local search is not capable of, i.e. unsatisfiability. Indeed, this is often the objective in a large number of significant real-world applications.

Nevertheless, it is also widely accepted that local search [11] can often have clear advantages with respect to backtrack search, since it is allowed to start the search over again whenever it gets *stuck* in a locally optimal partial solution. This advantage of local search has motivated the study of approaches for relaxing backtracking conditions (while still assuring completeness). The key idea is to *unrestrictedly* choose the point to backtrack to, in order to avoid thrashing during backtrack search. Moreover, one can think of combining different forms of relaxing the identification of the backtrack point. In this paper, we propose to use heuristic knowledge to select the backtrack point. The heuristics that we consider are inspired in the most promising branching heuristics proposed in recent years, namely the VSIDS heuristic used by Chaff [10] and Berkmin's branching heuristic [5].

The remainder of this paper is organized as follows. The next section presents a brief survey of backtrack search SAT algorithms. Then we describe heuristic backtracking and address completeness issues. Finally, we present preliminary experimental results and conclude with directions for future research work.

2 Definitions

This section introduces the notational framework used throughout the paper. Propositional variables are denoted $x_1, \ldots, x_n$, and can be assigned truth values 0 (or F) or 1 (or T). The truth value assigned to a variable x is denoted by $\nu(x)$. (When clear from context we use $x = \nu_x$, where $\nu_x \in \{0, 1\}$). A literal l is either a variable x_i or its negation $\neg x_i$. A clause ω is a disjunction of literals and a CNF formula φ is a conjunction of clauses. A clause is said to be *satisfied* if at least one of its literals assumes value 1, *unsatisfied* if all of its literals assume value 0, *unit* if all but one literal assume value 0, and *unresolved* otherwise. Literals with no assigned truth value are said to be *free literals*. A formula is said to be *satisfied* if all its clauses are satisfied, and is *unsatisfied* if at least one clause is unsatisfied. A *truth assignment* for a formula is a set of assigned variables and their corresponding truth values. The SAT problem consists of deciding whether there exists a truth assignment to the variables such that the formula becomes satisfied.

SAT algorithms can be characterized as being either *complete* or *incomplete*. Complete algorithms can establish unsatisfiability if given enough CPU time; incomplete algorithms cannot. In a search context, complete algorithms are often referred to as *systematic*, whereas incomplete algorithms are referred to as *non-systematic*.

3 Backtrack Search SAT Algorithms

Over the years a large number of algorithms have been proposed for SAT, from the original Davis-Putnam procedure [3], to recent backtrack search algorithms [1, 14, 9, 10] and to local search algorithms [11], among many others.

The vast majority of backtrack search SAT algorithms build upon the original backtrack search algorithm of Davis, Logemann and Loveland [2]. The backtrack search algorithm is implemented by a *search process* that implicitly enumerates the space of 2^n possible binary assignments to the n problem variables. Each different truth assignment defines a *search path* within the search space. A *decision level* is associated with each variable selection and assignment. The first variable selection corresponds to decision level 1, and the decision level is incremented by 1 for each new decision assignment [1]. In addition, and for each decision level, the *unit clause rule* [3] is applied. If a clause is unit, then the sole free literal must be assigned value 1 for the formula to be satisfied. In this case, the value of the literal and of the associated variable are said to be *implied*. The iterated application of the unit clause rule is often referred to as Boolean Constraint Propagation (BCP).

In chronological backtracking, the search algorithm keeps track of which decision assignments have been toggled. Given an unsatisfied clause (i.e. a *conflict* or a *dead end*) at decision level d, the algorithm checks whether at the current decision level the corresponding decision variable x has already been toggled. If not, the algorithms erases the variable assignments which are implied by the assignment on x, including the assignment on x, assigns the opposite value to x, and marks decision variable x as toggled. In contrast, if the value of x has already been toggled, the search backtracks to decision level $d - 1$.

Recent state-of-the-art SAT solvers utilize different forms of non-chronological backtracking [1, 9, 10], in which each identified conflict is analyzed, its causes identified, and a new clause created to explain and prevent the identified conflicting conditions. Created clauses are then used to compute the backtrack point as the *most recent* decision assignment from all the decision assignments represented in the recorded clause. Moreover, some of the (larger) recorded clauses are eventually deleted. Clauses can be deleted opportunistically whenever they are no longer *relevant* for the current search path [9].

4 Heuristic Backtracking

Heuristic backtracking consists of selecting the backtrack point in the search tree as a function of variables in the most recently recorded clause. Different heuristics can be envisioned. In this work we implemented different heuristics:

1. One heuristic that decides the backtrack point given the information of the most recently recorded conflict clause.

[1] Observe that all the assignments made before the first decision assignment correspond to decision level 0.

2. Another heuristic that is inspired in the VSIDS branching heuristic, used by Chaff [10].

3. Finally, one heuristic that is inspired by Berkmin's branching heuristic [5].

In all cases the backtrack point is computed as the variable with the largest heuristic metric. Completeness is ensured by marking the recorded clause as non-deletable.

We should observe that heuristic backtracking can be viewed as a special case of unrestricted backtracking [7]. While in unrestricted backtracking any form of backtrack step can be applied, in heuristic backtracking the backtrack point is computed from heuristic information, obtained from the current and past conflicts.

Next, we describe how the three different approaches of using heuristics are implemented in the heuristic backtracking algorithm.

4.1 Plain Heuristic Backtracking

Under the plain heuristic backtracking approach the backtrack point (i.e. decision level) is computed by selecting the decision level with the largest number of occurrences in a recorded clause. Afterwards, the search process backtracks to that decision level.

4.2 VSIDS-like Heuristic Backtracking

The second approach to heuristic backtracking is based on the variable state independent decaying sum (VSIDS) branching heuristic of Chaff [10]. As in Chaff, a metric is associated with each literal, which is incremented when a new clause is recorded that contains the literal; after every k decisions, the metric values are divided by a small constant. With the VSIDS-like heuristic backtracking, the assigned literal with the highest metric (of the literals in the recorded clause) is selected as the backtrack point.

4.3 BerkMin-like Heuristic Backtracking

The third approach for implementing heuristic backtracking is inspired in BerkMin's branching heuristic [5]. This heuristic is similar to the VSIDS heuristic used in Chaff, but the process of updating the metrics of the literals differs. In Berkmin's heuristic, the metrics of the literals of all clauses that are directly involved in producing the conflict, and so in creating the newly recorded clause, are updated when a clause is recorded. As in the cases of the VSIDS-like backtracking heuristic, the assigned literal with the highest metric (of the literals in the recorded clause) is selected as the backtrack point.

4.4 Relation with Unrestricted Backtracking

As mentioned above, heuristic backtracking can be viewed as a special case of unrestricted backtracking [7], the main difference being that while in unrestricted backtracking any form of backtrack step can be applied, in heuristic backtracking the backtrack point is computed from heuristic information, obtained from the current and past conflicts. As with unrestricted backtracking, a number of techniques can be used to ensure completeness. These techniques are analyzed in [7], and can be organized in two classes:

- Marking recorded clauses as non-deletable. This solution may yield an exponential growth in the number of recorded clauses.
- Increasing the number of conflicts in between applications of heuristic backtracking. This solution can be used to guarantee a polynomial growth of the number of recorded clauses.

5 Experimental Results

This section presents the experimental results of applying heuristic backtracking to different classes of problem instances. In addition, we compare heuristic backtracking with other forms of backtracking relaxations, namely search restarts and random backtracking [7]. Our goal here has been to test the feasibility of heuristic backtracking algorithm using three different heuristics: a default heuristic, the VSIDS heuristic and the Berkmin's heuristic. Experimental evaluation of the different algorithms has been done using the JQUEST SAT framework [8], a Java framework for prototyping SAT algorithms. All the experiments were run on the same P4/1.7GHz/1GByte of RAM/Linux machine. The CPU time limit for each instance was set to 2000 sec, except for instances from Beijing family, for which the maximum run time allowed was 5000 sec. In all cases where the algorithm was unable to solve an instance, the reason was memory exhaustion. The total run times for solving different class of benchmarks are shown in Table 1 and results for some specific instances are shown in Table 2. In both tables, *Time* denotes the CPU time and X denotes the number of aborted instances. In addition, each column indicates a different form of backtracking relaxation:

- RST indicates that the search restart strategy is applied with a cutoff value of 100 backtracks and is kept fixed. All recorded clauses are kept to ensure completeness.
- RB indicates that random backtracking is applied at each backtrack step.
- HB(P) indicates that heuristic backtracking is applied at each backtrack step. All recorded clauses are kept.
- HB(C) indicates that the VSIDS-like heuristic backtracking is applied at each step. All recorded clauses are kept.

Table 1: Performance of different algorithms on different classes of benchmark

Benchmarks	#I	RST		RB		HB(P)		HB(C)		HB(B)	
		Time	X	Time	X	Time	X	Time	X	Time	X
bmc-galileo	2	1885.93	0	3052.19	1	1575.97	0	1570.48	0	1553.83	0
bmc-ibm	11	3486.17	1	5781.31	1	4326.73	1	4340.53	1	4318.04	1
Hole	5	317.35	0	2318.71	1	245.69	0	244.27	0	240.27	0
Hanoi	2	2208.09	1	3560.72	1	2113.51	1	2113.02	1	2111.94	1
BMC-barrel	8	4764.22	2	7498.39	3	4505.44	2	4504.73	2	4505.00	2
BMC-queueinvar	10	96.92	0	776.87	0	78.30	0	76.32	0	76.52	0
Beijing	16	1190.09	2	5751.29	3	4539.51	2	4513.11	2	4520.41	2
Blocksworld	7	937.45	0	1312.18	0	324.07	0	325.73	0	320.41	0
Logistics	4	11.9	0	31.09	0	12.73	0	12.27	0	12.13	0
par16	10	972.39	0	1968.87	0	256.89	0	251.34	0	250.40	0
ii16	10	39.19	0	102.10	0	120.22	0	119.64	0	118.62	0
ucsc-ssa	102	29.89	0	37.59	0	29.41	0	29.56	0	29.48	0
ucsc-bf	223	78.10	0	106.57	0	85.05	0	78.93	0	79.78	0

- HB(B) indicates that the Berkmin-like heuristic backtracking is applied at each step. All recorded clauses are kept.

As can be concluded from the experimental results, heuristic backtracking can yield significant savings in CPU time, and also allow for a smaller number of instances to be aborted. This is true for several of the classes of problem instances analyzed.

6 Conclusions and Future Work

This paper proposes the utilization of heuristic backtracking in backtrack search SAT solvers. The most well-known branching heuristics used in state-of-the-art SAT solvers were adapted to the backtrack step of SAT solvers. The experimental results illustrate the practicality of heuristic backtracking. Besides the preliminary experimental results, a more comprehensive experimental evaluation is required. In addition, future work entails deriving conditions for selecting among search restarts and heuristic backtracking.

References

[1] R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence*, pages 203–208, July 1997.

Table 2: Performance of different algorithms on some specific instances

Instances	RST	RB	HB(P)	HB(C)	HB(B)
	Time	Time	Time	Time	Time
Beijing2bitadd_10	942.70	> 5000	4083.75	4061.04	4070.14
Beijing3bitadd_31	> 5000	> 5000	> 5000	> 5000	> 5000
Beijing3bitadd_32	> 5000	> 5000	> 5000	> 5000	> 5000
BMC-longmult7	263.55	> 2000	165.46	165.30	163.89
BMC-longmult8	> 2000	> 2000	1234.53	1223.3	1227.76
crypto-cnf-r3-b1-k1.1	16.61	127.31	46.20	45.58	45.91
crypto-cnf-r3-b1-k1.2	123.89	556.65	109.24	108.28	108.31

[2] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 5:394–397, July 1962.

[3] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, July 1960.

[4] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1979.

[5] E. Goldberg and Y. Novikov. BerkMin: a fast and robust sat-solver. In *Proceedings of the Design and Test in Europe Conference*, pages 142–149, March 2002.

[6] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the National Conference on Artificial Intelligence*, pages 431–437, July 1998.

[7] I. Lynce and J. P. Marques-Silva. Complete unrestricted backtracking algorithms for satisfiability. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, pages 214–221, May 2002.

[8] I. Lynce and J. P. Marques-Silva. Efficient data structures for backtrack search SAT solvers. In *Proceedings of the International Symposium on Theory and Applications of Satisfiability Testing*, pages 308–315, May 2002.

[9] J. P. Marques-Silva and K. A. Sakallah. GRASP-A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[10] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, June 2001.

[11] B. Selman and H. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 290–295, August 1993.

[12] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, October 1977.

[13] M. Yokoo. Weak-commitment search for solving satisfaction problems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 313–318, 1994.

[14] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997.