Towards Provably Complete Stochastic Search Algorithms for Satisfiability

Inês Lynce, Luís Baptista and João Marques-Silva

Department of Informatics,
Technical University of Lisbon,
IST/INESC/CEL, Lisbon, Portugal
{ines,lmtb,jpms}@sat.inesc.pt

Abstract. This paper proposes a stochastic, and complete, backtrack search algorithm for Propositional Satisfiability (SAT). In recent years, randomization has become pervasive in SAT algorithms. Incomplete algorithms for SAT, for example the ones based on local search, often resort to randomization. Complete algorithms also resort to randomization. These include, state-of-the-art backtrack search SAT algorithms that often randomize variable selection heuristics. Moreover, it is plain that the introduction of randomization in other components of backtrack search SAT algorithms can potentially yield new competitive search strategies. As a result, we propose a stochastic backtrack search algorithm for SAT, that randomizes both the variable selection and the backtrack steps of the algorithm. In addition, we describe and compare different organizations of stochastic backtrack search. Finally, experimental results provide empirical evidence that the new search algorithm for SAT results in a very competitive approach for solving hard real-world instances.

1 Introduction

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

In recent years, several competitive solution strategies for SAT have been proposed and thoroughly investigated [10, 9, 11]. Advanced techniques applied to backtrack search algorithms for SAT have achieved remarkable improvements [3, 7, 9, 11, 12, 15], having been shown to be crucial for solving large instances of SAT derived from real-world applications. Current state-of-the-art SAT solvers incorporate advanced pruning techniques as well as new strategies on how to organize the search. Effective search pruning techniques include, among others, clause recording and non-chronological backtracking [3, 9, 11], whereas recent effective strategies include search restart strategies [7]. Moreover, the work of S. Prestwich [12] (inspired by the previous work of others [6, 13]) has motivated the utilization of randomly picked backtrack points in incomplete SAT algorithms. More recently, a stochastic systematic search algorithm has been proposed [8].

The remainder of this paper is organized as follows. Section 2 briefly surveys SAT algorithms and the utilization of randomization in SAT. Afterwards, Section 3 introduces a stochastic backtrack search SAT algorithm and the next section details randomized backtracking. Preliminary experimental results are presented and analyzed in Section 5. Finally, we conclude and suggest future research work in Section 6.

2 SAT Algorithms

Over the years a large number of algorithms have been proposed for SAT, from the original Davis-Putnam procedure [5], to recent backtrack search algorithms [3, 9, 11, 15], among many others.

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 and incomplete algorithms are often referred to as *systematic*, whereas incomplete algorithms are referred to as *non-systematic*.

The vast majority of backtrack search SAT algorithms build upon the original backtrack search algorithm of Davis, Logemann and Loveland [4]. A generic organization of backtrack search for SAT considers three main engines:

- The decision engine (`Decide`) which selects an elective variable assignment each time it is called.
- The deduction engine (`Deduce`) which applies Boolean Constraint Propagation, given the current variable assignments and the most recent decision assignment, for satisfying the CNF formula.
- The diagnosis engine (`Diagnose`) which identifies the causes of a given conflicting partial assignment.

Recent state-of-the-art backtrack search SAT solvers [3, 9, 11, 15] utilize sophisticated variable selection heuristics, implement fast Boolean Constraint Propagation procedures, and incorporate techniques for diagnosing conflicting conditions, thus being able to backtrack non-chronologically and record clauses that explain and prevent identified conflicting conditions. Clauses that are recorded due to diagnosing conflicting conditions are referred to as *conflict-induced clauses* (or simply *conflict clauses*).

3 Stochastic Systematic Search

In this section we describe how randomization can be used within backtrack search algorithms to yield a *stochastic systematic search* SAT algorithm.

As previously explained in Section 2, a backtrack search algorithm can be organized according to three main engines: the decision engine, the deduction engine and the diagnosis engine. Given this organization, we define a backtrack search (and so systematic) SAT algorithm to be *stochastic* provided all three engines are subject to randomization:

1. Randomization can be (and has actually been [2, 3, 11]) applied to the decision engine by randomizing the variable selection heuristic.
2. Randomization can be applied to the deduction engine by randomly picking the order in which implied variable assignments are handled during Boolean Constraint Propagation.
3. The diagnosis engine can be randomized by randomly selecting the point to backtrack to.

For the deduction engine, randomization only affects the order in which assignments are implied, and hence can only affect which conflicting clause is identified first, and so it is not clear whether randomization of the deduction engine can play a significant role. As a result, we chose to randomize the two other engines of the backtrack search SAT algorithm.

Since the randomization of the decision engine is simply obtained by randomizing the variable selection heuristic [2, 3, 11], in the next section we focus on the randomization of the diagnosis engine.

4 Randomized Backtracking

State-of-the-art SAT solvers currently utilize different forms of non-chronological backtracking, for which each identified conflict is analyzed, its causes identified, and a new clause created and added to the CNF formula. 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.

The diagnosis engine of a non-chronological backtrack search algorithm can be randomized by randomly selecting the point to backtrack to. The conflict clause is then used for *randomly* deciding which decision assignment is to be toggled. This form of backtracking is referred to as *random backtracking*.

In SAT solvers implementing non-chronological backtracking and clause recording, even with opportunistic clause deletion, the algorithms are guaranteed to be complete, because there is always an implicit explanation for why a solution cannot be found in the portion of the search space already searched. However, in order to relax this backtracking condition and still ensure completeness, randomized backtracking requires that *all* recorded clauses must be kept in the CNF formula.

Moreover, there exists some freedom on *how* the backtrack step to the target decision assignment variable is performed and on *when* it is applied. For example, one can decide not to apply randomized backtracking after every conflict but instead only once after every K conflicts.

4.1 Completeness Issues

With randomized backtracking, clause deletion may cause already visited portions of the search space to be visited again. A simple solution to this problem is to prevent deletion of recorded clauses, i.e. no recorded conflict clauses are ever deleted. If no conflict clauses are deleted, then conflicts cannot be repeated, and the backtrack search algorithm is necessarily complete. The main drawback of keeping all recorded clauses is that the growth of the CNF formula is linear in the number of explored nodes, and so exponential in the number of variables. However, as will be described in Section 4.2, there are effective techniques to tackle the potential exponential growth of the CNF formula. Moreover, experimental data from Section 5 clearly indicate that the growth of the CNF formula is not exponential in practice.

It is important to observe that there are other approaches to ensure completeness that do not necessarily keep all recorded conflict clauses:

1. One solution is to increase the value of K each time a randomized backtrack step is taken.
2. Another solution is to increase the relevance-based learning [3] threshold each time a randomized backtrack step is taken (i.e. after K conflicts).
3. One final solution is to increase the size of recorded conflict clauses each time a randomized backtrack step is taken.

Observe that all of these alternative approaches guarantee that the search algorithm is eventually provided with enough space and/or time to either identify a solution or prove unsatisfiability. However, all strategies exhibit a key drawback: *paths in the search tree can be visited more than once*. Moreover, even when recording of conflict clauses is used, as in [9, 11], clauses can eventually be deleted and so search paths may be re-visited.

We should note that, as stated earlier in this section, if *all* recorded clauses are kept, then no conflict can be repeated during the search, and so no search paths can be repeated. Hence, as long as the search algorithm keeps *all* recorded conflict clauses, no search paths are ever repeated.

4.2 Implementation Issues

After (randomly) selecting a backtrack point, the actual backtrack step can be organized in two different ways:

- One can *non-destructively* toggle the target decision assignment, meaning that all other decision assignments are unaffected.
- One can *destructively* toggle the target decision assignment, meaning that all of the more recent decision assignments are erased.

The two randomized backtracking approaches differ significantly. Destructive randomized backtracking is more drastic and attempts to rapidly cause the search to explore other portions of the search space. Non-destructive randomized backtracking has characteristics of local search, in which the current (partial) assignment is only locally modified.

Another significant implementation issue is memory growth. Despite the growth of the number of clauses being linear in the number of searched nodes, for some problem instances a large number of backtracks will be required. However, there are effective techniques to tackle the potential exponential growth of the CNF formula. Next we describe two of these techniques:

1. The first technique for tackling CNF formula growth is to opportunistically apply subsumption to recorded conflict clauses. This technique is guaranteed to effectively reduce the number of clauses that are kept in between randomized backtracks.
2. Alternatively, a second technique consists of *just* keeping recorded conflict clauses that explain why each sub-tree, searched in between randomized backtracks, does not contain a solution. This process is referred to as identifying the *tree signature* [1] of the searched sub-tree.

Regarding the utilization of tree signatures, observe that it is always possible to characterize a tree signature for a given sub-tree T_S that has just been searched by the algorithm. Each time, after a conflict is identified and a randomized backtrack step is to be taken, the algorithm defines a path in the search tree. Clearly, the explanation for the current conflict, as well as the explanations for all of the conflicts in the search path, provide a *sufficient* explanation of why sub-tree T_S, that has just been searched, does not contain a solution to the problem instance.

4.3 Randomized Backtracking and Search Restart Strategies

It is interesting to observe that randomized backtracking strategies can be interpreted as a generalization of search restart strategies. The latter always start the search process from the root of the search tree, whereas the former randomly select the point in the search tree from which the search is to be restarted (assuming destructive backtracking is used). Moreover, observe that both approaches impose the same requirements in terms of completeness, and that the alternative techniques for completeness described in Section 4.1 for random backtracking also apply to search restart strategies.

It is also interesting to observe that the two strategies can be used together. In this case, each strategy S_{rb} (for randomized backtracking) or S_{rst} (for restarts) is applied after every K_{rb} or K_{rst} conflicts, respectively. In general we assume $K_{rb} < K_{rst}$, since S_{rst} causes the search to explore new portions of the search space that differ more drastically from those explored by S_{rb}.

5 Experimental Results

This section presents and analyzes experimental results that evaluate the effectiveness of the techniques proposed in this paper in solving hard real-world problem instances. Recent examples of such instances are the superscalar processor verification problem instances developed by M. Velev and R. Bryant [14]. We consider four sets of instances: *sss1.0a* with 9 satisfiable instances, *sss1.0* with 40 selected satisfiable instances, *sss2.0* with 100 satisfiable instances, and *sss-sat-1.0* with 100 satisfiable instances. For all the experimental results presented in this section a PIII @ 866MHz Linux machine with 512 MByte of RAM was used. The CPU time limit for each instance was set to 200 seconds, except for the *sss-sat-1.0* instances for which it was set to 1000 seconds. Since randomization was used, the number of runs was set to 10 (due to the large number of problem instances being solved). Moreover, the results shown correspond to the median values for all the runs.

In order to analyze the different techniques, a new SAT solver — Quest0.5 — has been implemented. Quest0.5 is built on top of the GRASP SAT solver [9], but incorporates restarts as well as *random backtracking*. Random backtracking is applied non-destructively after every K *backtracks* [1]. Furthermore, in what concerns implementation issues (see section 4.2), the backtracking point is selected from the union of the recorded conflict clauses in the most recent K conflicts and the tree signature of each sub-tree is kept in between randomized backtracks.

Moreover, for the experimental results, a few configurations were selected:

- **Rst1000+inc100** indicates that restarts are applied after every 1000 backtracks (i.e. the initial cutoff value is 1000), and the increment to the cutoff value after each restart is 100 backtracks. (Observe that this increment is necessary to ensure completeness.)
- **Rst1000+ts** configuration also applies restarts after every 1000 backtracks and keeps the clauses that define the tree signature when the search is restarted. Moreover, the cutoff value used is 1000, being kept fixed, since completeness is guaranteed.
- **RB1** indicates that random backtracking is taken at every backtrack step;
- **RB10** applies random backtracking after every 10 backtracks;
- **Rst1000+RB1** means that random backtracking is taken at every backtrack and that restarts are applied after every 1000 backtracks. (The identification of the tree signature is used for both randomized backtracking and for search restarts.)
- **Rst1000+RB10** means that random backtracking is taken after every 10 backtracks and also that restarts are applied after every 1000 backtracks. (The identification of the tree signature is used for both randomized backtracking and for search restarts.)

The results for Quest0.5 on the SSS instances are shown in Table 1. In this table, *Time* denotes the CPU time, *Nodes* the number of decision nodes, and X the average number of aborted problem instances. As can be observed, the results for Quest0.5 reveal interesting trends:

- Random backtracking taken at every backtrack step allows significant reductions in the number of decision nodes.
- The elimination of repeated search paths in restarts, when based on identifying the tree signatures and when compared with the use of an increasing cutoff value, helps reducing the total number of nodes and CPU time.
- The best results are always obtained when random backtracking is used, independently of being or not used together with restarts.

[1] For Quest0.5 we chose to use the number of backtracks instead of the number of conflicts. original GRASP code is organized [9].

Table 1. Results for the SSS instances

Inst	sss1.0a			sss1.0			sss2.0			sss-sat-1.0		
Quest 0.5	Time	Nodes	X	Time	Nodes	X	Time	Nodes	X	Time	Nodes	X
Rst1000+inc100	208	59511	0	508	188798	0	1412	494049	0	50512	8963643	39
Rst1000+ts	161	52850	0	345	143735	0	1111	420717	0	47334	7692906	28
RB1	79	11623	0	231	29677	0	313	31718	0	10307	371277	1
RB10	204	43609	0	278	81882	0	464	118150	0	6807	971446	1
Rst1000+RB1	79	11623	0	221	28635	0	313	31718	0	10330	396551	2
Rst1000+RB10	**84**	**24538**	**0**	**147**	**56119**	**0**	**343**	**98515**	**0**	**7747**	**1141575**	**0**
GRASP	1603	257126	8	2242	562178	11	13298	3602026	65	83030	12587264	82

– **Rst1000+RB10** is the only configuration able to solve all the instances in the allowed CPU time for *all* runs.

The experimental results reveal additional interesting patterns. When compared with the results for GRASP, Quest 0.5 yields dramatic improvements. Furthermore, even though the utilization of restarts reduces the amount of search, it is also clear that more significant reductions can be achieved with randomized backtracking. In addition, the integrated utilization of search restarts and randomized backtracking allows obtaining the best results, thus motivating the utilization of multiple search strategies in backtrack search SAT algorithms.

6 Conclusions and Future Work

This paper proposes and analyzes the application of randomization in the different components of backtrack search SAT algorithms. A new, stochastic but complete, backtrack search algorithm for SAT is proposed.

In conclusion, the contributions of this paper can be summarized as follows:

1. A new backtrack search SAT algorithm is proposed, that randomizes the variable selection and the backtrack steps.
2. The proposed SAT algorithm is shown to be complete, and different approaches for ensuring completeness are described.
3. Randomized backtracking is shown to be a generalization of search restart strategies, and their joint utilization is proposed.
4. Experimental results clearly indicate that significant savings in search effort can be obtained for different organizations of the proposed algorithm.

In the near future, we expect to consider other variations of this new algorithm. We can envision establishing a generic framework for implementing backtracking strategies, allowing the implementation of different hybrids, all guaranteed to be complete and so capable of proving unsatisfiability.

References

1. L. Baptista, I. Lynce, and J. Marques-Silva. Complete search restart strategies for satisfiability. In *IJCAI Workshop on Stochastic Search Algorithms*, August 2001.
2. L. Baptista and J. P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *International Conference on Principles and Practice of Constraint Programming*, pages 489–494, September 2000.
3. 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, 1997.
4. 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.
5. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.
6. M. Ginsberg and D. McAllester. GSAT and dynamic backtracking. In *Proceedings of the International Conference on Principles of Knowledge and Reasoning*, pages 226–237, 1994.
7. C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the National Conference on Artificial Intelligence*, July 1998.
8. I. Lynce, L. Baptista, and J. Marques-Silva. Stochastic systematic search algorithms for satisfiability. In *LICS Workshop on Theory and Applications of Satisfiability Testing*, June 2001.
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. D. McAllester, B. Selman, and H. Kautz. Evidence of invariants in local search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 321–326, August 1997.
11. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, 2001.
12. S. Prestwich. A hybrid search architecture applied to hard random 3-sat and low-autocorrelation binary sequences. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 337–352, September 2000.
13. E. T. Richards and B. Richards. Restart-repair and learning: An empirical study of single solution 3-sat problems. In *CP Workshop on the Theory and Practice of Dynamic Constraint Satisfaction*, 1997.
14. M. N. Velev and R. E. Bryant. Superscalar processor verification using efficient reductions from the logic of equality with uninterpreted functions to propositional logic. In *Proceedings of Correct Hardware Design and Verification Methods*, LNCS 1703, pages 37–53, September 1999.
15. H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997.