Stochastic Systematic Search Algorithms for Satisfiability

I. Lynce, L. Baptista and J. Marques-Silva
{ines,lmtb,jpms}@sat.inesc.pt
Technical University of Lisbon, INESC/CEL
Lisbon, Portugal

Abstract

This paper proposes the utilization of randomized backtracking within complete backtrack search algorithms for Propositional Satisfiability (SAT). Given that state-of-the-art SAT algorithms often randomize variable selection heuristics, and given that randomization is not naturally applicable to the identification of necessary assignments, our approach provides a fully stochastic, but complete, search algorithm for SAT. In addition, different organizations of randomized backtracking are described and compared. Moreover, we relate randomized backtracking with search restart strategies, that have recently been applied to SAT algorithms, and develop a new variation of search restarts, referred to as informed restarts, where the search is restarted as a function of previously identified backtrack search conflicts; hence an informed restart may not necessarily imply the complete elimination of the search tree. Finally, experimental results provide empirical evidence that randomized backtracking can be a competitive technique in solving hard real-world instances of SAT.

1 Introduction

Backtrack search algorithms for Propositional Satisfiability (SAT) have seen significant improvements in recent years [2, 7, 12, 10, 13, 18]. These improvements result from new search pruning techniques as well as new strategies for how to organize the search. Effective search pruning techniques include, among others, clause recording and non-chronological backtracking [2, 10, 12], whereas recent effective strategies include restarts [7] and (very recently) randomized backtracking [13].

Intrinsic to several of these improvements is randomization. Randomization has found application in different SAT algorithms, that include local search and backtrack search algorithms [11, 2]. In backtrack search, current state-of-the-art SAT solvers extensively resort to randomization, most often for selecting variable assignments but (and as a result) also within search restart strategies. Moreover, the recent work by S. Prestwich [13] (though preceeded by the work of others [6, 14]) has motivated the utilization of randomly picked backtrack points in SAT algorithms.

1.1 Objectives

This paper has four main objectives. First, to propose the utilization of random backtracking in backtrack search SAT algorithms. Second, to introduce the more general form of *unrestricted backtracking* where the backtracking point can be chosen arbitraly, e.g. randomly or based on heuristics. Third, to establish the relationship between a form of restart strategy and a specific form of unrestricted backtracking. Fourth, and finally, to provide empirical evidence that random backtracking can lead to significant savings in the amount of search effort.

1.2 Related Work

The proposed randomized backtracking search strategy has much in common with Ginsberg work on dynamic backtracking [5]. Dynamic backtracking establishes a method by which backtrack points can be moved deeper in the search tree. This allows avoiding the unneeded erasing of the amount of search that has been done thus far. The target is to find a way to "erase" the value given to a variable directly as opposed to backtracking to it. In practice, one can retain the values selected for variables that are backjumped over, in some sense moving the backjump variable to the end of the partial solution in order to replace its value without modifying the values of the variables that followed it. Note that this can be done not only with the last assigned variable, but also with any other variable involved in the conflict. We simply reorder the variables that have been assigned values in the search thus far.

Interestingly, most of the implementations that later incorporated these ideas have used methods not based on backtracking. For example, later Ginsberg and McAllester combined GSAT and dynamic backtracking in an algorithm which enables arbitrary search movement [6], starting with *any complete assignment* and evolving by flipping values of variables obtained from the conflicts. Similarly, Richards and Richards implemented another algorithm, *learn*-SAT [14], that starts from a *any consistent partial labeling* to the variables. In *learn*-SAT there is no notion of important early choices, since there is no backtracking. Here, learning is incorporated by recording the causes of the conflicts. Moreover, CLS, recently proposed by Prestwich [13] involves randomizing the backtracking component by allowing backtracking to occur on *arbitrarily-chosen* variables. In CLS the search algorithm starts with an empty assignment.

1.3 Organization of the Paper

The remainder of this paper is organized as follows. Section 2 presents definitions used throughout the paper. Afterwards, Section 3 briefly surveys SAT algorithms and the utilization of randomization in SAT. Next, in Section 4 we describe randomized backtracking and its application in complete backtrack search SAT algorithms. As a natural generalization, Section 5 introduces unrestricted bactracking and establishes the relationship between unrestricted backtracking and restarts. Preliminary experimental results are presented and analyzed in Section 6, and the paper concludes in Section 7.

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 $\nu(x)$. 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 is all its clauses are satisfied, and is unsatisfied if at least one clause is unsatisfied. The SAT problem is to decide whether there exists a truth assignment to the variables such that the formula becomes satisfied.

It will often be simpler to refer to clauses as sets of literals, and to the CNF formula as a set of clauses. Hence, the notation $l \in \omega$ indicates that a literal l is one of the literals of clause ω, whereas the notation $\omega \in \varphi$ indicates that clause ω is one of the clauses of CNF formula φ.

In the following sections we shall address backtrack search algorithms for SAT. Most if not all backtrack search SAT algorithms apply extensively the *unit clause rule* [4]. If a clause is unit, then the sole free literal must be assigned value 1 for the formula to be satisfiable. The iterated application of the unit clause rule is often referred to as Boolean Constraint Propagation (BCP) [17]. For implementing some of the techniques common to some of the most competitive backtrack search algorithms for SAT, it is necessary to properly *explain* the truth assignments to the propositional variables that are implied by the clauses of the CNF formula. For example, let $x = v_x$ be a truth assignment implied by applying the unit clause rule to a unit clause clause ω. Then the explanation for this assignment is the set of assignments associated with the remaining literals of ω, which are assigned value 0.

Let $\omega = (x_1 \vee \neg x_2 \vee x_3)$ be a clause of a CNF formula φ, and assume the truth assignments $\{x_1 = 0, x_3 = 0\}$. Then, for the clause to be satisfied we must necessarily have $x_2 = 0$. We say that the implied assignment $x_2 = 0$ has the explanation $\{x_1 = 0, x_3 = 0\}$. A more formal description of explanations for implied variable assignments, as well as a description of mechanisms for their identification, can be found for example in [10].

3 SAT Algorithms

Over the years a large number of algorithms have been proposed for SAT, from the original Davis-Putnam procedure [4], to recent backtrack search algorithms [2, 10, 18, 12], to local search algorithms [15], among many others.

SAT algorithms can be characterized as *complete* and as *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*.

Among the different algorithms, we believe backtrack search to be the most robust approach for solving hard, structured, real-world instances of SAT. This belief has been amply supported by extensive experimental evidence obtained in recent years [1, 10, 12].

3.1 Backtrack Search SAT Algorithms

The vast majority of backtrack search SAT algorithms build upon the original backtrack search algorithm of Davis, Logemann and Loveland [3]. Recent state-of-the-art backtrack search SAT solvers [2, 10, 18, 12] utilize sophisticated variable selection heuristics, 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.

3.2 Applying Randomization

The utilization of different forms of randomization in SAT algorithms has seen increasing acceptance in recent years.

Randomization is essential in many local search algorithms [15]. Most local search algorithms repeatedly restart the (local) search by randomly generating complete assignments. Moreover, randomization can also be used for deciding among different (local) search strategies [11].

Randomization has also successfully been included in variable selection heuristics of backtrack search algorithms [2]. Variable selection heuristics, by being greedy in nature, are unlikely but unavoidably bound to select the wrong variable at the wrong time for the wrong instance. The utilization of randomization helps reducing the probability of seeing this happening.

Although intimately related with randomizing variable selection heuristics, randomization is also a key aspect of restart strategies [7]. Randomization ensures that different sub-trees are searched each time the search algorithm is restarted.

Finally, randomization has also been used in the backtrack step of incomplete backtrack search algorithms. In CLS [13], the backtracking variable is randomly picked each time a conflict is identified. By not always backtracking to the most recent untoggled decision variable, the CLS algorithm is able to often avoid the characteristic trashing of backtrack search algorithms. We should note, however, that the CLS algorithm is not complete and so cannot establish unsatisfiability.

Current state-of-the-art SAT solvers already incorporate some of the above forms of randomization [1, 12]. In these SAT solvers variable selection heuristics are randomized and restart strategies are utilized.

4 Randomized Backtracking

This section describes different approaches for implementing randomized backtracking. One invariant of *all* these approaches is that *for each identified conflict a new clause is recorded and* never *deleted.* This is an essential aspect, since it guarantees the completeness of the resulting algorithm, as will be argued in section 4.3.

4.1 Generic Procedure

The generic randomized backtracking procedure is outlined in Figure 1. After a conflict (i.e. an unsatisfied clause ω_C) is identified, a conflict clause ω is created. The conflict clause is then used for *randomly* deciding which decision assignment variable is to be toggled. This contrasts with the usual non-chronological backtracking approach, in which the most recent decision assignment variable is selected as the backtrack point.

RANDOM_BACKTRACKING(ω_c)

```
{
    ω = Record_clause(ω_c);
    Randomly pick decision assignment variable v_x
      in ω;
    Toggle (v_x) with explanation ω;
}
```

Figure 1: Randomized Backtracking

Moreover, there exists some freedom on how the backtrack step to the target decision assignment variable is performed. Hence, the actual randomized backtracking procedure can be organized in several 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.
- One can decide not to apply randomized backtracking after every conflict but instead only once after every K conflicts.

In the following sub-section we address these issues. Finally, we argue that independently of how randomized backtracking is organized, the resulting algorithm still is complete.

4.2 Implementing Random Backtracking

After (randomly) selecting a backtrack point, the actual backtrack step can be organized in two different ways. The backtrack step can be *destructive*, meaning that the search tree is erased from the backtrack point down. In contrast, the backtrack step can be *non-destructive*, meaning that the search tree is not erased; only the backtrack point results in a variable assignment being toggled. The two randomized backtracking approaches differ. Destructive random backtracking is more drastic and attempts to rapidly cause the search to explore other portions of the search space.

Non-destructive random backtracking has characteristics of local search, in which the current (partial) assignment is only locally modified.

Either destructive or non-destructive random backtracking can lead to potentially unstable algorithms, since there is no locality in how backtracking is performed. This unstability may be a serious drawback when trying to prove unsatisfiability. As a result, we propose to only applying random backtracking after every K conflicts; in between non-chronological backtracking is applied. We should note that the value of K is user-defined.

In a situation where $K \neq 1$, the application of random backtracking can either consider the most recent recorded clause or, instead, consider a target set that results from the union of the recorded conflict clauses in the most recent K conflicts. The first solution is characterized by having no memory of past conflicts, whereas the second solution considers equally all identified conflicts.

4.3 Completeness Issues

Throughout the presentation of randomized backtracking the notion of clause (nogood) recording has been implicit. Each time a conflict is identified a clause is recorded that explains the conflict and prevents the same conflict from occurring again during the search. Most state-of-the-art SAT algorithms implement clause recording techniques, but of these most also eventually delete some of the (larger) recorded clauses. Clauses get deleted opportunistically whenever they are no longer *relevant* for the current search path [10].

In current state-of-the-art SAT solvers, even the ones implementing non-chronologically backtracking and clause recording with opportunistic clause deletion, the algorithms are guaranteeed to be complete, because there is always an explanation for why a solution cannot be found in the portion of the search space already searched. With randomized backtracking this is not necessarily the case; clause deletion may cause already visited portions of the search space to be searched again. A simple solution to this problem is to prevent deletion of recorded clauses. Consequently, with randomized backtracking large recorded clauses are never deleted. (We should note however that subsumption can be used to delete recorded clauses.) Since clauses are not deleted and each clause explains a conflict, no conflict is ever repeated and the algorithm is guaranteed to be complete.

The requirement to keep all recorded clauses has been previously stated by others [5, 14]. This requirement leads to a worst-case exponential growth of number of recorded clauses. However, as will be empirically shown in Section 6, this tends not to be the case (or even a problem) with real-world instances of SAT.

5 Unrestricted Backtracking and Restarts

Randomized backtracking can be interpreted as a special case of a more general form of backtracking: *unrestricted backtracking*. If instead of randomly picking the backtrack point from the identified conflict set, a heuristic or pre-defined procedure is used, then the resulting algorithm is still complete, and the backtrack point (though non-chronological) will most likely be different from the most recent decision assignment in the conflict set. Clearly, and similarly to randomized backtracking, unrestricted backtracking may also only be applied after every K conflicts, and it can either be destructive or non-destructive.

Interestingly, search restart strategies [7] can be related with unrestricted backtracking. Let us suppose an algorithm for which unrestricted backtracking is applied after every K conflicts, and for which the backtracking step is destructive. Clearly, after K conflicts a set of dependencies has been constructed that contains *all* decision assignments that were deemed responsible for *all* identified K conflicts. Let C denote this set of decision assignments, and let D be the current set of all decision assignments.

We can conceptually reorganize the search tree in such a way that the decision assignments in $D-C$ precede the ones in C. Clearly, with this rearrangement, there is no point in backtracking to any of the decision assignments in $D-C$, since none of these decision assignments contributed to the identified K conflicts. As a result, if our purpose is to restart the search from a point not related with previously identified conflicts (i.e. a new search path unrelated with previous conflicts), then it certainly suffices to backtrack to the most recent decision assignment in set $D - C$, keep *all* other decision assignments in set $D - C$, and erase the ones in C. We refer to this form of restarting the search as *informed restarts*, since only the decision assignments that are associated with identified conflicts are discarded. Consequently, informed restarts are defined as the application of destructive unrestricted backtracking, after every K conflicts, when the search tree is logically organized in sets $D - C$ and C and the backtrack point is picked to be the most recent decision assignment in set $D - C$.

Inst	ucsc-bf		ucsc-ssa	
Solver	Time	Nodes	Time	Nodes
Satz	1775	601130	1124	262158
Relsat	38	79230	33	79436
Grasp	58	33397	13	14233
Chaff	6	21809	2	14284

Table 1: Results for the UCSC instances

Inst	sss1.0a			sss1.0_sat			sss2.0		
Solver	Time	Nodes	X	Time	Nodes	X	Time	Nodes	X
Satz	1800	—	9	7826	—	39	15115	—	75
Relsat	1477	—	7	8000	—	40	12889	—	48
Grasp	1603	257126	8	2242	562178	11	13298	3602026	65
Chaff	20	73834	0	27	107020	0	84	228349	0

Table 3: Results for the SSS instances

Inst	ucsc-bf		ucsc-ssa	
Quest 0.5	Time	Nodes	Time	Nodes
Rst100	61	38915	13	13962
RB1	87	27169	27	14746
RB10	63	36625	15	13919
Rst100+RB1	88	27698	27	15048
Rst100+RB10	64	37415	15	14213

Table 2: Results for the UCSC instances

Inst	sss1.0a			sss1.0_sat			sss2.0		
Quest 0.5	Time	Nodes	X	Time	Nodes	X	Time	Nodes	X
Rst100	178	42545	0	352	95445	0	906	296637	1
RB1	128	12889	0	343	30568	0	609	39696	0
RB10	117	25432	0	346	79563	0	632	122170	1
Rst100+RB1	81	11578	0	146	23263	0	392	38337	0
Rst100+RB10	43	15930	0	97	37482	0	292	90569	0

Table 4: Results for the SSS instances

6 Experimental Results

This section presents and analyzes preliminary experimental results of the techniques proposed in this paper. A new SAT solver, Quest0.5, was implemented. Quest0.5 is built on top of the GRASP SAT solver [10], but incorporates restarts as well as random backtracking. In order to compare the proposed techniques we also include results for a few state-of-the-art SAT solvers, namely satz [9], relsat [2], GRASP [10] and Chaff [12]. For all the experimental results presented in this section a PIII @866MHz Linux machine with 512MByte of RAM was used. The CPU time limit for each instance was set to 200 seconds.

As mentioned above, Quest0.5 implements both restarts and random backtracking. With respect to random backtracking, the results presented in this section assume bactracking is non-destructive, and that random backtracking is applied after every *M backtracks*. For Quest0.5 we chose to use the number of backtracks instead of the number of conflicts to decide when to apply random backtracking. This results from how the search algorithm in the original Grasp code is organized. Moreover, for the experimental results shown below, RB1 indicates that random backtracking is taken on every backtrack step, RB10 indicates that random backtracking is taken every 10 backtracks and Rst100 indicates that a search restart is taken every 100 backtracks.

We start by including results for the UCSC problem instances, available as part of the DIMACS suite [8]. These instances are organized in two sets, the *bf* and the *ssa* problems instances. The number of *bf* instances is 223, of which 17 are satisfiable and 206 are unsatisfiable. The number of *ssa* instances is 102, of which 80 are satisfiable and 22 are unsatisfiable.

The results are shown in Tables 1 and 2. In each table, *Time* denotes the CPU time and *Nodes* the number of decision nodes. As can be observed, the utilization of restarts or randomized backtracking is not particularly useful, but also does not negatively affect the results when compared to Grasp. Observe that Chaff, a highly optimized SAT solver, performs extremely well in these problems instances, even though the amount of search is similar to both Grasp and Quest0.5. Moreover, satz performs significantly worse than the other algorithms and aborts some instances (7 for the bf set, and 6 for the ssa set).

The main objective of the techniques proposed in this paper is to be effective in solving hard real-world problem instances. Recent examples of such instances are the superscalar processor verification problem instances developed by M. Velev [16]. We consider three sets of instances, the first (sss1.0a) with 9 satisfiable instances, the second (sss1.0.sat) with 40 satisfiable instances, and the third (sss2.0) with 100 satisfiable instances. Tables 3 and 4 include results for these problem instances. Table 3 includes the results for satz, relsat, Grasp and Chaff, and Table 4 includes the results for the new SAT solver Quest0.5. Besides the *Time* and *Nodes* defined earlier, X denotes the number of aborted problem instances. Besides Chaff and Quest0.5, the other three algoritms are clearly inadequate for solving the SSS problem instances, in all cases always aborting on a large number of problem instances. In terms of CPU times Chaff is faster than Quest0.5 again due to its highly optimized implementation. However, and in contrast, all organizations of Quest0.5 in general require significantly less decision nodes than Chaff. This holds true either when using restarts, random backtracking or both.

The results for Quest0.5 reveal interesting trends. Clearly, random backtracking allows significant reductions in the number of decision nodes. The best results are always obtained when random backtracking is used together with restarts. Due to how random backtracking is implemented in Quest0.5 [1], the best configuration in terms of the amount of search is not the best configuration in terms of CPU time.

7 Conclusions

In this paper we explore some ideas related with the introduction of randomization in solving the satisfiability problem. We explain randomized backtracking, which randomly chooses the point to backtrack to, and introduce the notion of unrestricted backtracking. Moreover, we relate unrestricted backtracking with restart strategies. Preliminary experimental results clearly indicate that significant savings in search effort can be obtained by using random backtracking.

Other variations to the randomized backtracking can be considered. In randomized backtracking the algorithm randomly chooses the point to backtrack to based on the variables involved in the diagnosed conflict. Even keeping randomization associated with this choice, other aspects can be taken into account. A natural evolution is to consider heuristic approaches on how to select the backtrack point. In addition, a more extended experimental evaluation and categorization of the proposed techniques should be carried out.

References

[1] L. Baptista and J. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, 2000.

[2] 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.

[3] 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.

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

[5] M. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[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.

[1] In order to implement non-destructive random backtracking, the search tree needs to be reconstructed after each random backtrack. Hence if random backtracking is taken at each backtrack point, then a lot of tree reconstruction takes place, and consequently the CPU times are necessarily worse than when the random backtrack step is taken after every 10 backtracks.

[8] D. S. Johnson and M. A. Trick, editors. *Second DIMACS Implementation Challenge.* American Mathematical Society, 1993. (DIMACS SAT instances available from URL ftp://Dimacs.Rutgers.EDU/pub/-challenge/sat/benchmarks/cnf).

[9] C. M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of International Conference on Principles and Practice of Constraint Programming*, 1997.

[10] J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[11] 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, 1997.

[12] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering a (super?) efficient SAT solver. In *Proceedings of the Design Automation Conference*, 2001. Accepted for publication.

[13] 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, 2000.

[14] E. T. Richards and B. Richards. Non-systematic search and no-good learning. *Journal of Automated Reasoning*, 24(4):483–533, 2000.

[15] 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, 1993.

[16] 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.

[17] R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the National Conference on Artificial Intelligence*, pages 155–160, 1988.

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