Unrestricted Backtracking Algorithms for Satisfiability

I. Lynce, L. Baptista and J. Marques-Silva
Technical University of Lisbon, IST/INESC/CEL
R. Alves Redol, 9
1000-029 Lisboa, Portugal
{ines,lmtb,jpms}@sat.inesc.pt

Abstract

This paper proposes a general framework for implementing backtracking search strategies in Propositional Satisfiability (SAT) algorithms, that is referred to as unrestricted backtracking. Different organizations of unrestricted backtracking yield well-known backtracking search strategies. Moreover, this general framework allows devising new backtracking strategies. Hence, we illustrate and compare different organizations of unrestricted backtracking. For example, we propose a stochastic systematic search algorithm for SAT, that randomizes both the variable selection and the backtracking steps of the algorithm. Finally, experimental results provide empirical evidence that different organizations of unrestricted backtracking can result in competitive approaches for solving hard real-world instances of SAT.

Introduction

Backtrack search algorithms for Propositional Satisfiability (SAT) have seen significant improvements in recent years (Bayardo Jr. & Schrag 1997; Zhang 1997; Gomes, Selman, & Kautz 1998; Marques-Silva & Sakallah 1999; Prestwich 2000; Moskewicz *et al.* 2001). 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 (Bayardo Jr. & Schrag 1997; Marques-Silva & Sakallah 1999; Moskewicz *et al.* 2001), whereas recent effective strategies include restarts (Gomes, Selman, & Kautz 1998) and (very recently) randomized backtracking (Prestwich 2000).

Intrinsic to several of these improvements is randomization. Randomization has found application in different SAT algorithms, that include local search and backtrack search algorithms (McAllester, Selman, & Kautz 1997; Bayardo Jr. & Schrag 1997). 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. Search restarts are a well-known strategy for coping with hard real-world satisfiable (and often unsatisfiable) instances that is already being used by different state-of-the-art SAT solvers (Marques-Silva & Sakallah 1999; Moskewicz *et al.* 2001). Moreover, the recent work by S. Prestwich (Prestwich 2000) (though preceded by the work of others (Ginsberg & McAllester 1994; Richards & Richards 1997)) has motivated the utilization of randomly picked backtrack points in SAT algorithms.

Objectives

This paper has three main objectives. First, to propose a general form of backtracking search strategy, referred to as *unrestricted backtracking*. Second, to describe and analyze several more specific formulations of unrestricted backtracking, that include a stochastic systematic search algorithm for SAT, and search restart strategies. Third, and finally, to provide empirical evidence that specific formulations of unrestricted backtracking can lead to significant savings in the amount of search and time effort.

Organization of the Paper

The remainder of this paper is organized as follows. The next section presents definitions used throughout the paper. Afterwards, SAT Algorithms section briefly surveys SAT algorithms and the utilization of randomization in SAT. Unrestricted Backtracking section introduces unrestricted backtracking, analyzes a few implementation issues, and establishes the completeness of the proposed procedure. The next two sections relate unrestricted backtracking with stochastic systematic search and with search restart strategies. Preliminary experimental results are presented and analyzed in the Experimental Results section. Finally, we present related work and then the paper concludes.

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 if 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* (Davis & Putnam 1960). 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) (Zabih & McAllester 1988).

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. As an example, 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 (Marques-Silva & Sakallah 1999).

SAT Algorithms

Over the years a large number of algorithms have been proposed for SAT, from the original Davis-Putnam procedure (Davis & Putnam 1960), to recent backtrack search algorithms (Bayardo Jr. & Schrag 1997; Marques-Silva & Sakallah 1999; Zhang 1997; Moskewicz *et al.* 2001), to local search algorithms (Selman & Kautz 1993), 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 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 (Baptista & Marques-Silva 2000; Marques-Silva & Sakallah 1999; Moskewicz *et al.* 2001).

Backtrack Search

The vast majority of backtrack search SAT algorithms build upon the original backtrack search algorithm of Davis, Logemann and Loveland (Davis, Logemann, & Loveland 1962). Recent state-of-the-art backtrack search SAT solvers (Bayardo Jr. & Schrag 1997; Zhang 1997; Marques-Silva & Sakallah 1999; Moskewicz *et al.* 2001) 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. Clauses that are recorded due to diagnosing conflicting conditions are referred to as *conflict-induced clauses* (or simply *conflict clauses*).

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 (Selman & Kautz 1993). 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 (McAllester, Selman, & Kautz 1997).

Randomization has also successfully been included in variable selection heuristics of backtrack search algorithms (Bayardo Jr. & Schrag 1997). 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 (Gomes, Selman, & Kautz 1998). Randomization ensures that different sub-trees are searched each time the search algorithm is restarted.

Current state-of-the-art SAT solvers already incorporate some of the above forms of randomization (Baptista & Marques-Silva 2000; Moskewicz *et al.* 2001). In these SAT solvers variable selection heuristics are randomized and search restart strategies are utilized.

Generalizing Backtrack Search

A few alternatives to backtrack search have been proposed (Ginsberg 1993; Ginsberg & McAllester 1994; Richards & Richards 1997), some of which can actually be viewed as generalizations of the basic backtrack search procedure (Ginsberg 1993; Richards & Richards 1997). The utilization of search restart strategies (Gomes, Selman, & Kautz 1998; Baptista & Marques-Silva 2000; Moskewicz *et al.* 2001) can also be interpreted as an attempt to generalize backtrack search.

Moreover, the utilization of randomization can and has been used in the backtrack step of (incomplete) backtrack search algorithms. In CLS (Prestwich 2000), 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, and so be very competitive for specific classes of problem instances. We should note, however, that the CLS algorithm is not complete and so cannot establish unsatisfiability.

Unrestricted Backtracking

The most simple form of backtracking is chronological backtracking, for which each backtrack step is always taken to the most recent yet untoggled decision assignment. As mentioned above, 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 in the recorded clause. Backtracking to the *most recent* decision assignment ensures completeness even when recorded clauses are eventually deleted (Marques-Silva & Sakallah 1999).

Unrestricted backtracking relaxes the condition that backtracking must be taken to the *most recent* decision assignment in a recorded clause. However, in order to relax this condition and still ensure completeness, unrestricted backtracking requires that *all* recorded clauses must be kept in the CNF formula. Since *all* recorded clauses are kept, the number of clauses grows linearly with the number of conflicts, and so in the worst-case exponentially in the number of variables[1]. Observe that with unrestricted backtracking, and for each identified conflict, the actual backtrack point can be defined using either randomization, heuristic knowledge, constant-depth backtracking, search restart, among other possible approaches.

Generic Procedure

The unrestricted 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 *unrestrictedly* deciding which decision assignment 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.

[1]However, in all our experiments with real-world problem instances, the growth is far from being exponential in the number of variables.

UNRESTRICTED_BACKTRACKING(ω_c)
```
{
   ω = Record_conflict_clause(ω_c);
   select decision assignment variable v_x in ω;
   Apply_Backtrack_Step (v_x, ω);

}
```

Figure 1: Unrestricted Backtracking

Moreover, there exists some freedom on how the backtrack step to the target decision assignment variable is performed and on when it is applied. Hence, the actual unrestricted 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 each one of the more recent decision assignments is erased.
- One can decide not to apply unrestricted 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 unrestricted backtracking is organized, the resulting algorithm still is complete.

Implementation Issues

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

Either destructive or non-destructive unrestricted backtracking can lead to potentially unstable algorithms, since there is no locality in how backtracking is performed. This instability may be a serious drawback when trying to prove unsatisfiability. As a result, we propose to only applying unrestricted 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 unrestricted backtracking can either consider the most recent recorded clause or, instead, consider a target set

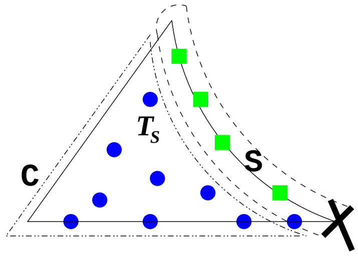

Figure 2: Identifying the tree signature of T_S

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.

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* (Baptista, Lynce, & Marques-Silva 2001) of the searched sub-tree.

Regarding the utilization of tree signatures, observe that 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. This path consists of all current decision assignments, and all the toggled decision assignments, which have been explained by identified conflicts. Thus, the current search path characterizes a sub-tree T_S that has just been searched by the algorithm. 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. The tree signature technique is illustrated in Figure 2, where the circles capture the process of recording *all* clauses associated with conflicts, and the squares denote the clauses that are obtained from the search path that characterizes T_S.

Completeness Issues

With unrestricted 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 described in the previous section, there are effective techniques to tackle the potential exponential growth of the CNF formula.

It is important to observe that there are other approaches to ensure completeness that do not necessarily keep all recorded conflict clauses [2]:

1. One solution is to increase the value of K each time an unrestricted backtrack step is taken.

2. Another solution is to increase the relevance-based learning (Bayardo Jr. & Schrag 1997) threshold each time a unrestricted 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 unrestricted 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 (Marques-Silva & Sakallah 1999; Moskewicz *et al.* 2001), 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.

Stochastic Systematic Search

The unrestricted backtracking search strategy lends itself to very different specializations. In this section we describe how randomization can be used with unrestricted backtracking to yield a stochastic systematic search algorithm for SAT.

As shown in the previous section, any form of unrestricted backtracking that keeps *all* recorded clauses results in a complete, thus systematic, search algorithm. This algorithm can be made stochastic by randomizing the variable selection heuristic, and by randomly picking the backtracking point from the set of literals

[2] In order to describe these other approaches, we consider the generic situation in which the unrestricted backtrack step is taken after every K conflicts.

in each recorded conflict clause. Hence, unrestricted backtracking in this situation corresponds to random backtracking.

As with general unrestricted backtracking, the backtracking process can be destructive or non-destructive. Non-destructive backtracking gives the algorithm properties of local search (Prestwich 2000), whereas destructive backtracking resembles the usual backtrack step of backtrack search algorithms.

Moreover, we may only randomly pick the backtrack point after every K conflicts. In between, we may either use chronological or non-chronological backtracking, even though for most solvers non-chronological backtracking is used.

Finally, since algorithms that implement unrestricted backtracking are complete, the stochastic systematic search algorithm described above is necessarily complete.

Search Restart Strategies

Search restart strategies have been shown to be an effective technique for handling problem instances for which the run time of search algorithms are characterized by heavy-tailed distributions (Gomes, Selman, & Kautz 1998).

The generic search restart procedure consists of restarting the search algorithm after every K conflicts (or backtracks). If the value of K is fixed, the resulting algorithm is incomplete. A simple modification is to increase the backtrack cutoff value after each search restart (Baptista & Marques-Silva 2000), yielding a procedure that resembles *iterative deepening*. The resulting algorithm is complete, and this procedure has recently been incorporated into a highly optimized SAT solver (Moskewicz *et al.* 2001).

However, the increasing cutoff approach still exhibits a key drawback: paths in the search tree can still be visited more than once. Observe that, even when clause recording is used, as in (Marques-Silva & Sakallah 1999; Moskewicz *et al.* 2001), clauses are eventually deleted and so search paths may be re-visited.

Interestingly, if *all* recorded clauses are kept, then no conflict can be repeated during the search, and so no search paths can be repeated (see Figure 3). Hence, as long as the search algorithm keeps *all* recorded clauses, no search paths are ever repeated and so even a constant-cutoff restart strategy is guaranteed to be complete.

Moreover, as shown in the Completeness Issues section, even when unrestricted backtracking is only taken after every K conflicts, the resulting algorithm is complete, because all clauses recorded due to conflicts are kept.

As a result, if *all* recorded clauses are kept, then it is possible to interpret search restarts as a special case of unrestricted backtracking. Basically, the organization of the unrestricted backtracking step consists of:

- To *destructively* backtrack to the beginning of the search tree.

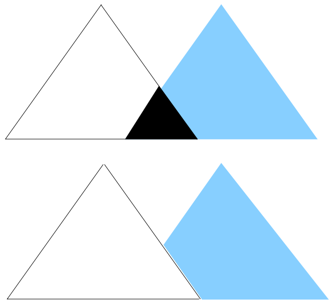

Figure 3: Eliminating repeated search paths

- To apply the unrestricted backtrack step only after every K conflicts.

As in the previous section, since the proposed algorithm is a special case of unrestricted backtracking, completeness is guaranteed.

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 (Velev & Bryant 1999). 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 average 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 (Marques-Silva & Sakallah 1999), but incorporates restarts as well as *random backtracking*. Random backtracking is a specialization of unrestricted backtracking. The backtracking step is applied non-destructively after every K *backtracks* [3]. Furthermore, in what concerns implementation issues (see the correspondent section), 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.

[3]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 (Marques-Silva & Sakallah 1999).

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

Moreover, for the experimental results shown below, the following 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 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.
- **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. (This is confirmed by evaluating either the CPU Time, the number of nodes or the number of aborted instances.) 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.

Related Work

In this section we describe related work, namely how randomization has been integrated in the backtrack step of different SAT solvers.

The introduction of randomization in the backtrack step is related with Ginsberg's work on dynamic backtracking (Ginsberg 1993). 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 directly "erase" the value assigned to a variable as opposed to backtracking to it, 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 currently follow it. Interestingly, most of the implementations that later incorporated dynamic backtracking (Ginsberg & McAllester 1994; Richards & Richards 1997) have used methods not based on backtracking.

More recently, CLS, proposed by Prestwich (Prestwich 2000), involves randomizing the backtracking component by allowing backtracking to *arbitrarily-chosen* variables. In CLS (Prestwich 2000), 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, and so can be very competitive for specific classes of problem instances. We should note, however, that the CLS algorithm is not complete and so it cannot establish *unsatisfiability*.

Moreover, search restarts have been proposed and shown effective for real-world instances of SAT by the work of Gomes, Selman and Kautz (Gomes, Selman, & Kautz 1998). The search is repeatedly restarted whenever a cutoff value is reached. The algorithm proposed in (Gomes, Selman, & Kautz 1998) is not complete, since the restart cutoff point is kept constant. In (Baptista & Marques-Silva 2000), search restarts were used jointly with learning to solving very hard real-world instances of SAT. This later algorithm is complete, since the backtrack cutoff value increases after each restart (see the Completeness Issues section). In these two approaches, some forms of search redundancy exist, that may unnecessarily increase the run times. Nonetheless, as we argued in the mentioned section, there are effective techniques to avoid repeating search paths.

Conclusions

In this paper we explore some ideas related with the introduction of randomization in solving the satisfiability problem. We introduce the notion of unrestricted backtracking, which enables *unrestrictedly* selecting the point to backtrack to, as well as new techniques for guaranteeing the completeness of SAT algorithms that characterize unrestricted backtracking. Moreover, we relate unrestricted backtracking with randomized backtracking, that randomly chooses the point to backtrack to, and we show that search restart strategies can be viewed as a specific case of unrestricted backtracking. Preliminary experimental results clearly indicate that significant savings in search effort can be obtained by using different forms of unrestricted backtracking. For future work, other variations of unrestricted backtracking can be considered. In addition, a more extended experimental evaluation and categorization of the proposed techniques should be carried out.

References

Baptista, L., and Marques-Silva, J. P. 2000. Using randomization and learning to solve hard real-world instances of satisfiability. In *International Conference on Principles and Practice of Constraint Programming*, 489–494.

Baptista, L.; Lynce, I.; and Marques-Silva, J. 2001. Complete search restart strategies for satisfiability. In *IJCAI Workshop on Stochastic Search Algorithms*.

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

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

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

Ginsberg, M., and McAllester, D. 1994. GSAT and dynamic backtracking. In *Proceedings of the International Conference on Principles of Knowledge and Reasoning*, 226–237.

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

Gomes, C. P.; Selman, B.; and Kautz, H. 1998. Boosting combinatorial search through randomization. In *Proceedings of the National Conference on Artificial Intelligence*.

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

McAllester, D.; Selman, B.; and Kautz, H. 1997. Evidence of invariants in local search. In *Proceedings of the National Conference on Artificial Intelligence*, 321–326.

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference.*

Prestwich, S. 2000. A hybrid search architecture applied to hard random 3-sat and low-autocorrelation binary sequences. In *Proceedings of International Conference on Principles and Practice of Constraint Programming*, 337–352.

Richards, E. T., and Richards, B. 1997. 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.*

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

Velev, M. N., and Bryant, R. E. 1999. 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, 37–53.

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

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