Complete Restart Strategies Using a Compact Representation of the Explored Search Space

Alex S. Fukunaga
Jet Propulsion Laboratory
Califorrnia Institute of Technology
4800 Oak Grove Dr., M/S 126-347
Pasadena, CA 91109-8099
fukunaga@aig.jpl.nasa.gov

Abstract

Applying restarts to complete search algorithms for constraint satisfaction is an effective method for improving their expected performance on some difficult problems. An issue with restarts is that completeness can be lost if the algorithm is constantly interrupted in the midst of a search. We propose a general method for called path-recording, which guarantees completeness for restart strategies and requires space linear in the number of restarts and decision variables. An example application in the domain of propositional satisfiability testing is presented.

1 Introduction

A *restart strategy* is a technique by which a search/optimization algorithm is interrupted in the midst of a run, then restarted such that a different portion of the search space is explored.

Restarts have long been used in the context of local search and other incomplete search/optimization algorithms in order to encourage exploration of the search space and escape local optima. Recently, restarts have been shown to yield dramatic performance improvements when applied to complete, backtracking-based algorithms for constraint satisfaction problems (CSPs), including boolean satisfiability (SAT) [Gomes *et al.*, 1998; 2000], enabling the solution of problems which were previously unsolvable.

Gomes et al studied the variability in runtimes of algorithms for CSPs and found that the performance of backtracking algorithms exhibits a heavy-tailed distribution [Gomes *et al.*, 2000]. For many "difficult" problem instances, they observed that some runs of a randomized, complete solver terminated very quickly, while other runs on the same instance took a very long time. That is, the runtime of a given algorithm on an instance depended not only on the problem instance, but also on the particular random choices made by the algorithm. By repeatedly, rapidly restarting a search algorithm, it is possible to take advantage of any significant mass in the leftmost part of this runtime distribution, i.e., even though a problem instance might be very difficult most of the time, a series of short runs could result in a run which "got lucky" and solved the instance quickly.

RestartSearch(ProblemInstance)

```
solver-state = InitialState;
Repeat:
 solver-state=BacktrackSearch(solver-state)
 Until solution found or timeout
```

Figure 1: A restarting search strategy

Although the discovery of heavy-tailed distributions in large-scale CSPs as a motivation for restarts is relatively new, the underlying problem (costly "mistakes" made by backtracking algorithms early on in the search tree leading to long periods of fruitless search) has been well-known, and similar strategies have been previously studied. For example, Huberman, Hogg, and Lukose [Huberman *et al.*, 1997] motivate the use of algorithm portfolios (parallel runs of search algorithms applied to a single problem instance) using a similar argument. Iterative sampling [Langley, 1992] can be viewed as a degenerate backtracking algorithm that "restarts" after every leaf node expansion.

The general schema for a solver using randomized restarts is very simple, as shown in Figure 1. `BacktrackSearch` is a backtracking algorithm (e.g., chronological backtracking, conflict-directed backjumping [Stallman and Sussman, 1977]), which has been modified such that based on some *restart policy*, or *schedule*, it will interrupt the search and return control to the top-level `RestartSearch` function. In some cases, `BacktrackSearch` has been modified so that its variable/value selection strategies are randomized (c.f. [Gomes *et al.*, 1998] so that repeated calls to `BacktrackSearch` results in different search algorithm behavior. In other cases, conflict clauses learned during the previous call to `BacktrackSearch` can cause the search behavior in a subsequent call to be significantly different without any need for explicit randomization [Moskewicz *et al.*, 2001].

There are several issues that are introduced by applying restarts. First, completeness is lost. Suppose we are given an unsatisfiable CSP problem instance. A complete, backtracking algorithm will eventually return UNSAT. However, if we take a standard backtracking algorithm and apply random restarts with some arbitrary restart frequency, we are

no longer guaranteed that we can prove unsatisfiability. Although it may be possible for the solver to prove unsatisfiability within a given time window between restarts, the *guarantee* of completeness is lost, because the solver could always end up restarting before a proof of unsatisfiability is completed.[1]

A second problem is the loss of *systematicity*, the property that nodes in the tree are visited at most once. Systematicity is at least theoretically desirable, since revisiting search states is intuitively "inefficient". Unfortunately, after a restart, there is no guarantee that a state which was visited in a previous invocation of `BacktrackSearch` in Figure 1 will not be re-examined in subsequent iterations.

One approach to guaranteeing completeness suggested in [Gomes *et al.*, 1998] is to use a restart schedule where the duration of each run in between restarts is gradually increased, so that eventually, the final uninterrupted, "restart" will result in a complete, search of the tree. A drawback of this approach is that if the underlying solver does not use a mechanism such as clause learning, then each restart essentially throws away all of the work done in the previous restarts, which can be wasteful. Note that although they proposed gradually increasing cutoffs, Gomes et al actually used a constant cutoff value between restarts [Gomes *et al.*, 1998] because the strategy of rapid random restarts maximizes the likelihood of hitting a "lucky" restart (i.e., exploiting any significant mass in the good part of the runtime distribution).

Another approach is to rely on a conflict-clause recording mechanism. Some modern SAT-solvers such as *Chaff* [Moskewicz *et al.*, 2001], *GRASP* [Marques-Silva and Sakallah, 1999], and *relsat* [Bayardo and Schrag, 1997] incorporate a learning mechanism that adds new clauses to the clause set. Usually, these solvers implement a policy which periodically eliminate some of the learned clauses based on the age and/or "relevance" of the clauses. Lynce and Marques-Silva [2002] show that if all recorded clauses are kept between restarts, or if a policy gradually caused all clauses to be kept (e.g., by gradually increasing the size-bound or relevance-bound of the retained clauses) then then completeness can be guaranteed. A problem with this second approach is that for a large enough problem instance, it will be impractical to keep all clauses for the duration of the run without running out of memory/storage. In addition, at some point, the overhead introduced by the presence of all learned clauses can slow down the search algorithm.

A third approach proposed by Baptista, Lynce and Marques-Silva [2001] for a SAT solver based on conflict-directed backjumping is to use a *search signature* that summarizes the subtree that has been explored prior to a restart. The search signature is a set of clauses added to the formula during conflict-directed backtracking representing the based on the *causes* [Marques-Silva and Sakallah, 1996] for backtracking of the variables representing the current search path. Search signatures use significantly less memory than keeping all clauses learned by conflict analysis.

In the rest of this paper, we propose *path-recording*, an alternate approach based on compactly representing the portions of the search space that have already been explored. Path-recording is similar to search signatures, but is more general, in that it is an extension to backtracking, and is totally independent of any conflict analysis mechanism. Because it depends only structure of the search tree, it can be applied to any backtracking-based algorithm. We show how path-recording can be implemented for satisfiability testing, and report some preliminary empirical results that show that path-recording can be used to improve the performance of backtracking satisfiability solvers with restarts.

[1]The same problem exists for satisfiable problems, but it is more likely to be a problem for unsatisfiable problems because in general, solving unsatisfiability requires proving that the whole tree does not contain a solution.

2 Path-Recording

We now propose a simple method which restores both completeness, and (to some extent) systematicity to randomized restart algorithms, at very little cost. The key insight is that is possible to exploit the structure of the search trees explored by backtracking algorithms in order to compactly and quickly *record* which portions of the search tree have already been explored in the form of new constraints. These constraints are added to the system in such a way as to prevent the search algorithm from re-visiting the states encoded by these constraints.

The technique will first be demonstrated using a simple example.

Suppose we have a SAT instance with 4 variables, A, B, C, D.

Assume a standard chronological backtracking algorithm (i.e., depth-first search) that searches the space of variable-value assignments. For simplicity, assume that conflicts are not detected until all variables are assigned a value, i.e., there is no constraint propagation. Furthermore, assume that variables are selected in lexicographic order, and values are assigned in an arbitrary order (first T is tried, then F).

The algorithm will start at the root (all variables unassigned). First, the algorithm assigns $A = T$. Then it would assign $B = T$, then $C = T$, then $D = T$. At this point, having failed to find a solution, it would backtrack on D and assign it a value of F. If that fails, it would backtrack on D and C, assigning F to C and T to D.

Using an abbreviated notation, this can be described as:

$A = T, B = T, C = T, D = T$; fail; backtrack on D.

$A = T, B = T, C = T, D = F$; fail; backtrack on D, then backtrack on C.

...and so on, until either a solution is found or we reach the state: $A = F, B = F, C = F, D = F$, at which point the search tree is exhausted and we return UNSATISFIABLE.

Figure 2 illustrates a partially completed search tree for this example.

Now, suppose that the search is interrupted after the evaluation of the state: $A = T, B = F, C = T, D = F$. That is, a restart is triggered after the 6th leaf node from the left in Figure 2 is expanded.

We can inspect Figure 2 and try to summarize the nodes that have been expanded so far. The nodes that have already been expanded are exactly the nodes to the left of the *current*

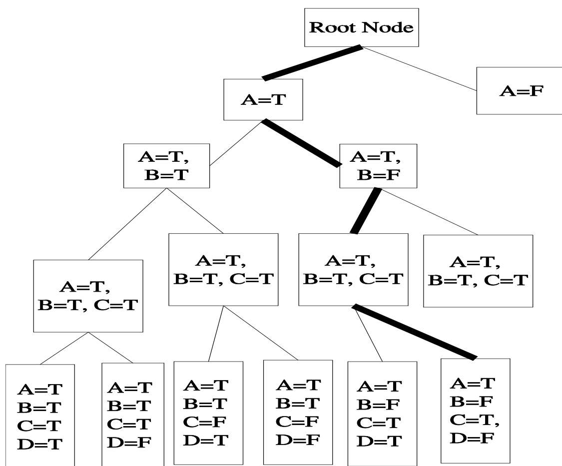

Figure 2: Partial search tree for a backtracking SAT solver

BacktrackSearch(Vars)

```
If solution-found
 Return solution
v= Select a variable from Vars
for each value a in Domain(v)
Assign(Vars=a)
DecisionStack.push_front(pair(v,a))
BacktrackSearch(Vars-v)
DecisionStack.pop_front()
Return failure (no solution)
```

Figure 3: Example backtracking algorithm with an explicitly maintained decision stack

path (the sequence of decisions marked by the dark lines). It should be clear that we have proven that any state with the partial assignment $A = T, B = T$ have been completely exhausted. Thus, we can conclude: $\neg(A \wedge B)$. Similarly, we can also conclude $\neg(A \wedge \neg B \wedge C \wedge \neg D)$. Therefore, the constraint (in conjunctive normal form) which exactly summarizes the subtree which has been explored so far (and proven not to have a solution) is $(\neg A \vee \neg B) \wedge (\neg A \vee B \vee \neg C \vee \neg D)$ The two clauses $C_1 = (\neg A \vee \neg B)$ and $C_2 = (\neg A \vee B \vee \neg C \vee \neg D))$ are added to the original formula.

Now, consider what happens after a restart. If we ever reach a partial assignment which sets $A = T, B = T$, then C_1 is violated, forcing a backtrack.

From this example, we can see intuitively that summarizing the explored portion of the search space requires us to record one clause for every right-branch taken by the current path. We now present the path recording algorithm.

Let the *decision stack* be a list $d1, d2, ..d_n$ which encodes the decisions made in the search tree.

Figure 3 shows a chronological backtracking algorithm which maintains a decision stack. The *push* and *pop* operators add and remove assignments to the decision stack.

When a restart takes place, the solver invokes the

ProcessPath(path_remaining,path-above)

```
Decision = Pop_from_back(path_remaining)
If Decision = right_branch
 C = NewClause()
 Add_Literal(C,Decision)
 For each Decision in Path_Above
  Add_Literal(C,Opposite(Decision))
 Add_Clause_To_Clause_Database(C)
ProcessPath(path_remaining,
     path-above+Decision)
```

Figure 4: A Path-Recording algorithm for SAT

`ProcessPath` algorithm (Figure 4). The call to `ProcessPath(CurrentPath,NULL)` adds the necessary clauses required to record the portion of the search space explored in this current invocation of the backtracking algorithm.

The number of clauses learned by path recording after each restart is linear in V, the number of decision variables. More precisely, it is no more than the number of right-branches in the current path. Thus, after R restarts, there will be at most RV new clauses. In addition, there is a small, $O(V)$ space requirement for maintaining the current path.

The call to `ProcessPath(CurrentPath)` clearly takes $O(length(CurrentPath))$ time, which is $O(V)$, linear in the number of decision variables. Since ProcessPath only needs to be called just before a restart, the overhead introduced into the overall solver will be negligible, assuming that restarts are relatively infrequent.

The other overhead is incurred when adding/removing decisions to the current path (Figure 3). However, both of these operations can be done in constant time per decision if the current path is implemented as a stack. Profiling the SAT solver used for our experiments in Section 3 showed that this overhead is negligible.

The above model and examples assumed a very simple depth-first backtracking algorithm with static (lexicogoraphic) variable and value orderings. However, it is easy to see that even with more sophisticated variable/value ordering, path-recording still retains completeness. Also, the usual extensions to backtracking such as constraint propagation and conflict-directed backjumping do not affect its correctness.

Also, in our examples we assumed that the restart took place immediately after some leaf node was opened. Path-recording is correct even if the restart takes place in an interior node of the search tree, because the nodes which are eliminated from future consideration correspond to the leaf nodes to the left of wherever the current path is.

The above algorithm for propositional SAT can be generalized in a straightforward manner for arbitrary trees. Figure 5 shows part of a search tree for a problem on a tree with branch factor three (e.g., a constraint satisfaction problem where variables are assigned one of 3 possible values). Given the current path marked by the dark line, the constraints to be added in this case are:

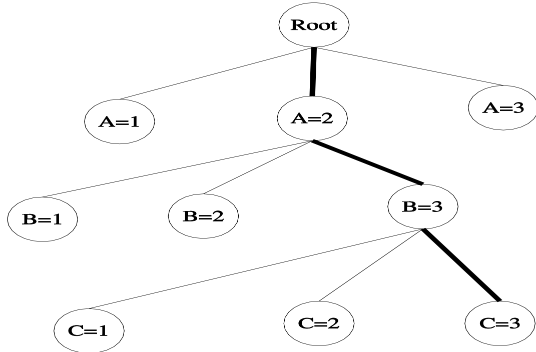

Figure 5: Partial search tree for a backtracking algorithm applied to a 3-valued search problem

$\neg(A = 1)$,
$\neg((A = 2 \wedge B = 1) \vee (A = 2 \wedge B = 2))$,
$\neg((A = 2 \wedge B = 3 \wedge C = 1) \vee (A = 2 \wedge B = 3 \wedge C = 2))$

3 Empirical Study: SAT

We now empirically study path-recording applied to the domain of SAT. Backtracking algorithms based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [Davis and Putnam, 1960; Davis *et al.*, 1962] represent the current state of the art in complete, SAT solvers. Gomes et al extended two systematic DPLL solvers, *satz* and *relsat*, with random restarts and found significantly improved performance compared to the complete, deterministic original versions on some problem instances. Random restarts are now commonly used in modern DPLL-based solvers such as *Chaff* [Moskewicz *et al.*, 2001].

We implemented a systematic, DPLL based SAT-solver. It uses conflict-directed backjumping, 1-UIP clause learning, and the VSIDS variable selection heuristic as implemented in *zChaff* [Zhang *et al.*, 2001]. The current implementation is a prototype, lacking several features found in state-of-the-art solver implementations. It uses a counter-based scheme for unit propagation [Crawford and Auton, 1993] rather than a watched literal scheme, and we have made no attempt to optimize data structures for cached memory accesses; both of these was found to have a significant impact on performance [Moskewicz *et al.*, 2001; Zhang and Malik, 2003]. Finally, our solver is implemented in Common Lisp with only moderate hand-coded optimizations (e.g., type declarations, user-level memory management), which imposes some performance penalties for the sake of enabling rapid prototyping. Nevertheless, the overall performance of our solver seems to be competitive with reasonably efficient solvers such as *relsat*. All of our experiments were performed on a 2.78GHz Intel Pentium 4, 512K L2 cache, and 1GB memory.

As a further optimization which exploits path-recording, a simple subsumed-clause elimination scheme was implemented. A clause c is subsumed by another c_1 if the literals in c are a strict subset of the literals in c_1. For example, $(a \vee b \vee \neg c)$ is subsumed by $(a \vee \neg c)$. Note that if c_1 subsumes c, then c is redundant and can be removed from the database without affecting the search algorithm behavior. Removing subsumed clauses can speed up the search (by removing the overhead associated with maintaining the subsumed clauses in the clause database). During a restart, when the path clauses are added to the database, all previously existing clauses are compared to the path clauses, and any subsumed clauses which are found are removed from the database.

When restarting, the solver uses the algorithm described in Section 2 and adds the derived path clauses to the clause database.

We tested the following configurations:

- Forget-everything
- Path-Recording (PR)
- PR+1UIP+Forget-Conflicts
- PR+1UIP
- 1UIP

These configurations are composed of different combinations of three clause learning and forgetting strategies, as shown in Tables 1 and 2. *Path recording* indicates whether path recording was turned on. *1-UIP* indicates the use of the 1-UIP conflict-based learning algorithm. *Forget conflicts* indicates that even when 1-UIP was activated, all conflict clauses were deleted during each restart. Thus, in our experiment, conflict clauses are either not learned at all, learned and never forgotten, or deleted after each restart. We have not yet experimented with various policies for periodically removing clause databases,such as relevance-based, age-based, and length-based policies (c.f. [Bayardo and Schrag, 1997; Zhang *et al.*, 2001]).

Two sets of test formula were used: The first set was a set of 150 variable, 615-clause *unsatisfiable* formula (100 instances) from the uuf150 benchmark set at the `satlib` (www.satlib.org) database (hard, random unsatisfiable instances from the phase transition region [Mitchell *et al.*, 1992]). The second set was the Superscalar Suite 1.0a benchmark set (9 instances, all satisfiable) by Miroslav Velev, available at http://www.ece.cmu.edu/ mvelev.

Three *restart policies* were used, all of them using a $increment$ parameter:

- *constant* - restarts every $increment$ backtracks.
- *doubling* - The nth restart occurs after $interval \times 2^n$ backtracks. E.g., if $interval$=100, restarts occur at 100, 200, 400, 800, 1600...backtracks.
- *linear* - The first restart occurs at $OriginalInterval = increment$ backtracks. The next restart occurs $n \times increment$ backtracks after the current restart. E.g., if $OriginalInterval$=100, then restarts occur at 100, 300, 600, 1000, 1500... backtracks.

Tables 1 and 2 and summarizes the results. The "clauses at end" column indicates the numbe of clauses in the clause database at the end of each run, including all clauses learned by 1-UIP and path-recording. We observe that:

configuration name	path recording	1-UIP	forget conflicts	restart policy	restart increment	backtracks	assignments	clauses at end	runtime
Forget-Everything	n	n	n/a	constant	1000	no successes			
Path-Recording (PR)	y	n	n/a	constant	1000	18510	610789	809	4.96
Forget-Everything	n	n	n/a	constant	100	no successes			
Path-Recording (PR)	y	n	n/a	constant	100	20024	656354	1917	7.90
PR+1UIP+Forget-Conflicts	y	y	y	constant	100	12951	431668	1458	5.52
1UIP	n	y	n	constant	100	8203	276376	8766	10.50
PR+1UIP	y	y	n	constant	100	8201	278132	1950	8.98
Forget-Everything	n	n	n	linear	100	97724	3235352	645	25.41
Path-Recording (PR)	y	n	n	linear	100	17384	575339	798	4.74
PR+1UIP+Forget-Conflicts	y	y	y	linear	100	9365	318631	1398	3.79
1UIP	n	y	n	linear	100	7861	267469	8493	7.15
PR+1UIP	y	y	n	linear	100	7922	270536	3415	6.52
Forget-Everything	n	n	n	doubling	100	29040	961636	645	7.50
Path-Recording (PR)	y	n	n	doubling	100	17313	571734	709	4.64
PR+1UIP+Forget-Conflicts	y	y	y	doubling	100	8703	297254	3106	4.70
1UIP	n	y	n	doubling	100	7952	270652	8590	7.94
PR+1UIP	y	y	n	doubling	100	8027	273136	5033	6.60

Table 1: Comparison of restart strategies on 150 variable, 645 clause unsatisfiable formulas (100 instances from the uuf150 dataset at `satlib`). Mean ove100 instances. (Timeout after 100000 backtracks)

- The utility of path-recording depends on the class of problem instance, as well algorithmic features (e.g., restart strategy, whether clause-learning was enabled).
- When 1-UIP is not used, path-recording significantly improves performance over not learning anything at all.
- When 1-UIP is used, path-recording does not significantly reduce the number of backtracks, but eliminating conflict clauses subsumed by the recorded path results in a smaller clause database, resulting in somewhat faster runtime than 1-UIP alone.

For the constant restart schedule, PR+1UIP+Forget-Conflicts outperformed pure path-recording. However, in Table 1, it is worth noting that path-recording by itself is sufficient to allow all of the instances to be solved, even though *none* of the Forget-Everything runs succeeded in proving unsatisfiability of the formulas within the backtrack bound (100,000 total backtracks) bound, for an $increment$ value of either 100 or 1000. This demonstrates a case where the risk of applying a rapid restart strategy is mitigated by restoring completeness using path-recording.

4 Discussion

We described path-recording, an algorithm for generating constraints summarizing the subtrees explored by a backtracking algorithm. Path-recording can be implemented as an extension to most backtracking algorithms for constraint satisfaction and optimization. Preliminary experiments indicate that it can help the performance of a restarting, SAT-solver. For any particular domain, there will be tradeoffs between 1) the benefits of completeness and limited systematicity offered by the technique, and 2) the overhead associated with generating and representing these new constraints.

Path-recording is quite similar to the search signature technique proposed by Baptista et al [2001]. The main idea of remembering sufficient constraints to summarize the subtree that has just been searched is the same. The main difference is that path-recording generalizes this key idea and presents and implements it in a framework completely independent of conflict-directed backtracking. As a result, it can be applied to many other domains to which systematic tree-search algorithms can be applied.

In addition to its application to restarts, the path-recording technique can be used to share/broadcast the exhausted regions of the search space among processes in an algorithm portfolio or distributed problem-solving system. For example, if two backtracking-based solvers are executed in parallel on the same problem instance, they can succinctly communicate to each other which regions of the search space have already been exhausted, without having to explicly list every single state that has been explored.

Although we have focused on constraint satisaction, path-recording can be applied to other tree-search domains. For example, in branch-and-bound algorithms for optimization (e.g., traveling salesperson), path-tracking can be used to enable restarts without losing completeness and systematicity. This will allow short, rapid "probes" of the search space in order to try to quickly obtain good upper bounds that can be used to increase pruning and decrease the overall search effort. However, in some of these domains, the overhead incurred by adding the constrain-checking infrastructure necessary to enable path-recording may overwhelm the potential reduction in search effort, while in CSP domains such as SAT, there is very little additional overhead incurred by path-recording because efficient representation of constraints is already a necessary part of state-of-the-art solvers. Another promising application seems to be branch-and-bound algorithms for integer programming, which, like CSPs, readily supports the insertion of new constraints derived by path-recording.

configuration name	path recording	1-UIP	forget conflicts	restart policy	restart increment	backtracks	assignments	clauses at end	runtime
Forget-Everything	n	n	n/a	constant	1000	6919	1137654	36982	10.8
Path-Recording (PR)	y	n	n/a	constant	1000	6634	1082914	37099	11.2
PR+1UIP+Forget-Conflicts	y	y	y	constant	1000	3654	579406	37417	8.4
1UIP	n	y	n	constant	1000	2597	433470	39576	6.3
PR+1UIP	y	y	n	constant	1000	2625	437258	39457	6.7
Forget-Everything	n	n	n/a	doubling	1000	18482	2368821	36982	25.2
Path-Recording (PR)	y	n	n/a	doubling	1000	13862	1879868	37063	19.6
PR+1UIP+Forget-Conflicts	y	y	y	doubling	1000	2123	352206	37517	5.7
1UIP	n	y	n	doubling	1000	1319	223317	38299	3.0
PR+1UIP	y	y	n	doubling	1000	1346	224185	38244	3.2
Forget-Everything	n	n	n/a	linear	1000	10706	1660774	36982	16.3
Path-Recording (PR)	y	n	n/a	linear	1000	7667	1150884	37110	11.9
PR+1UIP+Forget-Conflicts	y	y	y	linear	1000	2312	393093	37242	5.3
1UIP	n	y	n	linear	1000	1349	236247	38327	3.4
PR+1UIP	y	y	n	linear	1000	1345	234301	38241	3.3

Table 2: Comparison of restart strategies on sss1.0a dataset (9 instances). Mean of 10 randomized runs on the batch of 9 instances (e.g., expected runtime to solve all 9 instances sequentially)

Acknowledgments

This work was performed by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. Thanks to Nathan Sturtevant and Russell Knight for helpful discussions related to this work.

References

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

[Bayardo and Schrag, 1997] Roberto J. Jr. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

[Crawford and Auton, 1993] James M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In Richard Fikes and Wendy Lehnert, editors, *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 21–27, Menlo Park, California, 1993. AAAI Press.

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

[Davis *et al.*, 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[Gomes *et al.*, 1998] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 431–437, Madison, Wisconsin, 1998.

[Gomes *et al.*, 2000] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.

[Huberman *et al.*, 1997] B.A. Huberman, R.M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, 1997.

[Langley, 1992] P. Langley. Systematic and nonsystematic search strategies. In *Proc. First International Conf on Artificial Intelligence Planning Systems*, pages 145–152, 1992.

[Lynce and Marques-Silva, 2002] I. Lynce and J.P. Marques-Silva. Complete unrestricted backtracking algorithms for satisfiability. In *Proc. Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, May 2002.

[Marques-Silva and Sakallah, 1996] J.P. Marques-Silva and K.A. Sakallah. GRASP - a new search algorithm for satisfiability. In *Proc. IEEE/ACM International Conerence on Copmputer-Aided Design*, November 1996.

[Marques-Silva and Sakallah, 1999] 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.

[Mitchell *et al.*, 1992] David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions for SAT problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, Menlo Park, California, 1992. AAAI Press.

[Moskewicz *et al.*, 2001] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. "chaff: Engineering an efficient sat solver". In *Proc. 38th Design Automation Conference (DAC2001)*, Las Vegas, 2001.

[Stallman and Sussman, 1977] 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, 1977.

[Zhang and Malik, 2003] L. Zhang and S. Malik. Cache performance of sat solvers: A case study for efficient implementation of algorithms. In *Proc. Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, Portofino, Italy, May 2003.

[Zhang *et al.*, 2001] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *Proc. ICCAD*, pages 279–285, 2001.