The Effect of Restarts on the Efficiency of Clause Learning

Jinbo Huang
Logic and Computation Program
National ICT Australia
Canberra, ACT 0200 Australia
jinbo.huang@nicta.com.au

Abstract

Given the common use of restarts in today's clause learning SAT solvers, the task of choosing a good restart policy appears to have attracted remarkably little interest. Many solvers, for example, restart once every k conflicts with such a diverse choice of k as to give the impression that one policy may be just as good as another. On the other hand, results have been reported on the use of different restart policies for combinatorial search algorithms. Such results are not directly applicable to clause learning SAT solvers, as the latter are now understood as performing a form of resolution, something fundamentally different from search. In this paper we provide strong evidence that the restart policy matters, and deserves to be chosen less arbitrarily, for a clause learning SAT solver. We begin by pointing out that other things held constant, it is the combination of the restart policy and decision heuristic that completely determines the sequence of resolution steps performed by the solver, and hence its efficiency. In this spirit we implement a prototype clause learning SAT solver that facilitates restarts at arbitrary points, and conduct experiments on an extensive set of industrial benchmarks using various restart policies, including those used by well-known SAT solvers as well as a universal policy proposed in 1993 by Luby *et al*. The results indicate a substantial impact of the restart policy on the efficiency of the solver, and provide motivation for the design of better restart policies.

Introduction

Propositional satisfiability (SAT) is the problem of determining whether a propositional formula, traditionally in conjunctive normal form (CNF), has a *satisfying assignment*—an assignment of truth values to its variables making it evaluate to true. In this paper we focus on a class of algorithms for SAT that has become known as *conflict-driven clause learning*, or *clause learning* for short (Ryan 2004). These algorithms are currently the best for large SAT instances that arise from industrial applications, such as formal verification (Berre & Simon 2005).

Clause learning SAT solvers have grown out of their predecessors that implemented variants of a systematic search algorithm known as DPLL (Davis, Logemann, & Loveland 1962), which solves SAT by selecting a variable and determining, recursively, whether the formula can be satisfied by setting that variable to either value. In fact, the initial intuition for learning clauses and appending them to the CNF formula was to help prune the DPLL search, as discussed in earlier work (Marques-Silva & Sakallah 1996).

It has been shown, however, that clause learning as practiced in today's SAT solvers, particularly with unlimited restarts, corresponds to a proof system exponentially more powerful than that of DPLL (Beame, Kautz, & Sabharwal 2004). Specifically, each learning step is in fact a sequence of resolutions, and the learned clause the final resolvent thereof; conversely, a resolution proof can be simulated in polynomial time by repeatedly (i) learning each of the resolvents in the proof and (ii) restarting.[1] Clause learning can hence be as powerful as general resolution, while DPLL has been known to correspond to the exponentially weaker tree-like resolution (Beame, Kautz, & Sabharwal 2004).

Despite the dependence of this theoretical result on the assumption of unlimited restarts, remarkably little has been said in the literature on the importance of choosing a good restart policy in practice. This is in stark contrast, for example, to the sustained quest by researchers for better decision heuristics. We argue that this imbalance of attention is doing a disservice to the SAT community, because with the modern understanding of clause learning, it can be seen that the decision heuristic, *together with* the restart policy, determines the sequence of resolutions performed by, and hence the efficiency of, the solver (assuming that the learning scheme and other aspects of the solver are held constant).

In this paper we would like to take a step toward studying the practical importance of restart policies in clause learning SAT solvers, with a view to motivating further work on designing more effective policies. To this end we have created a small prototype SAT solver, called TINISAT, which implements the essentials of a modern clause learning solver and is designed to facilitate adoption of arbitrary restart policies. After choosing and fixing a reasonably effective decision heuristic, we conducted experiments using an extensive set of large industrial benchmarks, on which we ran versions of TINISAT using different restart policies including those used by well-known SAT solvers and particularly one proposed in (Luby, Sinclair, & Zuckerman 1993) based on a sequence of run lengths of the following form:

[1] This assumes a deviation from standard SAT practice—the freedom to ignore assignments made by unit propagation (Beame, Kautz, & Sabharwal 2004).

$1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots$ The results we have obtained indicate a substantial impact of the restart policy on the efficiency of the solver. Specifically, all nontrivial restart policies we experimented with did significantly better than if restarts were disabled, and Luby's policy in particular achieved the best overall performance, on the set of benchmarks used.

The rest of the paper is organized as follows: We present the simple design of TINISAT and use it as the basis for discussing the semantics of modern clause learning, leading to an analytical explanation why restart policies are important. We then describe our experimental setup including the various restart policies we shall use and our attempts to identify a reasonable decision heuristic so that all policies can be tested on competitive ground. We then report the results obtained and make a number of important observations. Finally, we discuss related work and present our conclusions.

TINISAT: The Essentials of Clause Learning

We start by presenting the design of a simple SAT solver, TINISAT, that (i) boasts the essentials of modern clause learning technology, and (ii) facilitates the experimentation with arbitrary restart policies. It shall then provide a basis for our discussion of the importance of restart policies for clause learning algorithms. The top-level procedure of TINISAT is given in Algorithm 1, which operates on an implicit CNF formula whose satisfiability is in question.

Algorithm 1 TINISAT

```
 1: loop
 2:   if (literal = selectLiteral()) == nil then
 3:     return SATISFIABLE
 4:   if !decide(literal) then
 5:     repeat
 6:       learnClause()
 7:       if assertionLevel() == 0 then
 8:         return UNSATISFIABLE
 9:       if restartPoint() then
10:         backtrack(1)
11:       else
12:         backtrack(assertionLevel())
13:     until assertLearnedClause()
```

The following components of a modern clause learning SAT solver can be identified in Algorithm 1: decision heuristic (selectLiteral), unit propagation (decide, assertLearnedClause), clause learning (learnClause, backtrack), restarts (restartPoint, backtrack). We assume familiarity with the common terminology for DPLL and clause learning algorithms, and assume that (i) 1-UIP (Zhang *et al.* 2001) is used as the learning scheme, (ii) no clauses are ever deleted (hence completeness is not an issue), (iii) all functions have deterministic behavior,[2] and (iv) the first decision is made in decision level 2: level 1 is reserved for literals found to be implied by the CNF formula, and level 0 to signal derivation of the empty clause. The functions involved have the following semantics:

- selectLiteral uses some decision heuristic to select a free variable and then select one of its two literals, and returns it, or returns **nil** if no free variables exist.
- decide increments the decision level, sets the given literal to true, and performs unit propagation; it returns true iff no empty clause is derived.
- learnClause performs 1-UIP learning to derive an implicate of the CNF formula, and sets the assertion level (i) to 0 if the empty clause is derived, (ii) to 1 if a unit clause is derived, and otherwise (iii) to the second highest decision level among literals of the derived clause.
- assertionLevel returns the assertion level, which has been set by the last call to learnClause.
- restartPoint returns true iff the solver is to restart now according to some restart policy.
- backtrack(k) undoes all variable assignments in decision levels $> k$, and sets the decision level to k.
- assertLearnedClause adds the learned clause to the clause pool, performs unit propagation if the current decision level equals the assertion level (this is the condition under which the learned clause becomes unit), and returns true iff no empty clause is derived.

Restarts and Backtracks Unified

Note that under this design all that is needed to adopt a given restart policy is to implement restartPoint accordingly. It is also interesting to note that a normal backtrack after learning (Line 12) and a complete restart (Line 10) can both be regarded as special cases of a more general scheme (Lynce & Silva 2002) where the solver can backtrack to any level between 0 and the current decision level (exclusive). What we would like to stress here, however, is that the particular scheme used by most clause learning SAT solvers today, namely backtracking to the assertion level (except when restarting), has obscured their original characteristics (inherited from DPLL) as systematic search algorithms. In particular, these solvers do not perform branching anymore: The setting of a literal occurs on Line 4, but the setting of its negation is never explicitly tried (and possibly never tried at all even implicitly).

For example, suppose assignments $\{A, \overline{B}, C, \overline{D}\}$ have been made in decision levels $2, 3, 4, 5$, respectively, before the empty clause is derived in level 5, and suppose the following clause is learned: $\overline{A} \vee D$. This clause says that the two decisions made in levels 2 and 5 alone are responsible for the conflict. Now, despite the fact that simply flipping the value of variable D in level 5 could well result in a satisfiable subproblem, Line 12 insists on taking the solver back to the assertion level, which is 2, erasing assignments $\{\overline{D}, C, \overline{B}\}$ on the way. The learned clause then gets asserted in level 2 (Line 13), and implies D (because the assignment A is present, making the clause unit), triggering a round of unit propagation. Notice that the branches $\{A, \overline{B}, \overline{C}\}$ and $\{A, B\}$, which can well contain solutions, have been skipped over without ever being explored.

[2] The learning scheme, clause deletion policy, and use of randomness are all important factors that affect the efficiency of clause learning SAT solvers, but are beyond the scope of this paper.

It should be emphasized here, as we already alluded to, that this behavior is not a peculiarity of TINISAT, but is the common practice of most current clause learning SAT solvers we have seen, including Chaff, BerkMin, MiniSat, and Siege. (The earlier solver GRASP, though, used a different backtracking scheme such that no branch was skipped over unless proven to contain no solutions.)

The Importance of Restarts

This shift of paradigm in backtracking, as we discussed, obscures the characteristics of clause learning SAT solvers as systematic search algorithms. For this reason we propose to view the modern practice of clause learning not as a version of DPLL search enhanced with resolution,[3] but as a pure resolution algorithm in its own right. In fact, what Algorithm 1 does is nothing other than the following 3-step cycle:

(1) set variables till hitting a conflict;

(2) derive a clause by resolution;

(3) unset some variables and go back to (1).

For unsatisfiable formulas, this loop terminates on deriving the empty clause in (2); for satisfiable formulas, it terminates when (1) "happens" to exhaust all variables without conflict.

In this context the importance of the restart policy becomes prominent: Together with the existing backtracking scheme, it dictates the set of assignments to undo in (3), which, together with the decision heuristic, ultimately determines the entire sequence of resolution steps performed in (2). In other words, the decision heuristic, backtracking scheme, and restart policy can all be understood as serving a single purpose, that of guiding the resolution process.

Consider for example a clause learning SAT solver that has run on a hard instance for a period of time without restarts. The solver has now accumulated a considerable number of learned clauses, which have helped update the variable and literal scores as are maintained by decision heuristics typical in clause learning SAT solvers. These new scores represent, in a way, the solver's current state of belief about the order in which future decisions should be made, having taken into account all the conflicts discovered so far. Without the freedom of restarts, however, the solver would not be able to fully execute its belief because it is bound by the decisions that have been made earlier. In particular, note that these early decisions were made *without* the benefit of the new knowledge in the form of all the conflicts discovered since. This, we believe, is the main reason why restarts can help improve the efficiency of clause learning SAT solvers even when no randomness is present (which will be the case in our experiments with TINISAT).

In this section we have provided a new understanding of modern clause learning through the design of TINISAT, a concrete and simple clause learning SAT solver, leading to an analytical argument that the *restart policy matters*. We now proceed to support this argument with an empirical study of concrete restart policies using real-world SAT benchmarks and the TINISAT solver.

[3]Recall from (Beame, Kautz, & Sabharwal 2004) that each learning step is a sequence of resolution steps the final resolvent of which is recorded as the "learned clause."

Experimental Setup

We describe in this section the restart policies we shall experiment with, the decision heuristic to be used with these policies in the experiments, and our choice of benchmarks.

Restart Policies

In choosing the set of restart policies for our empirical study, we have aimed to include those that are currently used by well-known SAT solvers as well as some potentially more effective ones that may have escaped the attention of the clause learning community. Specifically, we shall experiment with the following seven restart policies:

- N: a policy calling for no restarts at all.
- M: a geometric policy used in MiniSat v1.14 (Eén & Sörensson 2005) with an initial restart interval of 100 conflicts, which increases by a factor of 1.5 after each restart. We will denote it by (100, 1.5).
- Z: a fixed-interval policy used in Chaff II, also known as the 2004 version of zChaff (Moskewicz *et al.* 2001), with a restart interval of 700 conflicts, denoted (700, 1).
- B: a fixed-interval policy used in BerkMin (Goldberg & Novikov 2002) with a restart interval of 550 conflicts, denoted (550, 1).
- G: a geometric policy (32, 1.1), which we have added to improve the balance between fixed-interval and geometric policies we consider.
- S: a fixed-interval policy used in Siege (Ryan 2004) with a restart interval of 16000 conflicts, denoted (16000, 1).
- L: a class of policies proposed in (Luby, Sinclair, & Zuckerman 1993) for randomized algorithms based on the following sequence of run lengths: $1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots$ (defined below). In our experiments we take a "unit run" in this sequence to be 32 conflicts.[4] Hence the actual restart intervals are: $32, 32, 64, 32, 32, 64, 128, \ldots$ We denote this policy by (Luby's, unit=32).

The first six of these policies are straightforward, while Luby's policy can be formally defined as the sequence $t_1, t_2, t_3, \ldots$ such that:

$$t_i = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1; \\ t_{i-2^{k-1}+1}, & \text{if } 2^{k-1} \leq i < 2^k - 1. \end{cases}$$

We have chosen Luby's policy because of an interesting property it has[5]: In the context of a particular class of randomized algorithms, known as *Las Vegas* algorithms, (Luby,

[4]Use of other units is certainly possible. Given our computing resources, however, we restrict ourselves to a single representative of this class of policies. The particular choice of 32 was made, somewhat arbitrarily, after a limited number of experiments.

[5]We have not studied the theoretical relevance of this property in the context of clause learning.

Sinclair, & Zuckerman 1993) proved that this policy is *universally optimal* in the sense that (i) it achieves an expected running time that is only a logarithmic factor slower than the true optimal policy, which is determined by the specific running time distribution of the algorithm on a specific problem instance, and (ii) no other universal policy can do better by more than a constant factor.

Decision Heuristic

To maximize the reliability of our comparison of restart policies, we have taken steps to ensure that other components of the SAT solver, which will be TINISAT as we mentioned, are tuned toward their best performance. Rather than low-level optimizations, however, we focused on designing a reasonably effective decision heuristic. Based on experiments using a subset of our full benchmark suite (described below), we found that a variation of the VSIDS heuristic (Moskewicz *et al.* 2001) combined with BerkMin's practice of choosing literals from recent unsatisfied conflict clauses (Goldberg & Novikov 2002) tended to work well.

Specifically, for each literal we keep a score that is initially the number of its occurrences in the original clauses. On learning a clause, we increment the score of every literal by 1 for each of its occurrences in clauses that are involved in the resolution process.[6] The scores of all literals are halved once every 128 conflicts. When a decision is called for (Line 2 of Algorithm 1), we pick a (free) literal with the highest score from the most recently learned clause that has not been satisfied, and set it to true; if no such clause exists we pick any (free) literal with the highest score.

Benchmarks

We use the entire set of industrial benchmarks distributed by Miroslav Velev of Carnegie Mellon University at http://www.ece.cmu.edu/~mvelev/, except sss.1.0, sss.1.0a, sss-sat-1.0, vliw-sat-1.0, and vliw-sat-1.1 as they are too easy,[7] and dlx-iq-unsat-2.0 as the download appeared to be incomplete. This gives us 22 benchmark families with 251 instances totaling about 25GB in size—hence the CNF formulas have an average size of about 100MB.

Results

Our experiments consist of running TINISAT with each of the seven restart policies on the entire set of benchmarks. As a check that the experiments are not handicapped by the unsophisticated implementation of TINISAT (written in under 800 lines of C++), we have also run MiniSat v1.14 (Eén & Sörensson 2005) and Siege v4 (Ryan 2004) (given seed 123456789 for its random number generator) on the same set of benchmarks. All our experiments were conducted on a cluster of 16 AMD Athlon 64 processors running at 2GHz (comparable to a 4GHz Pentium) with 2GB of RAM under SuSE Linux 9.3 Professional. A time limit of 2 hours was imposed on all runs of the solvers, allowing us to complete all the experiments in about 80 CPU days.

The overall results are shown in Table 1. In the second and third columns we report for each benchmark family the number of instances and their total size (in megabytes). In the remaining columns we report the number of instances solved by each solver for each benchmark family. The total number of instances solved by each solver and the total time it spent on all instances (including the 2 hours in case it did not solve the instance) are reported in the two bottom rows.

The first observation we make from these results is that restarts definitely helped: All the six nontrivial restart policies did significantly better than the no-restarts policy. Even the least effective of them allowed TINISAT to finish 2.43 days sooner and solve 37 more instances, than the no-restarts policy.

Our second observation is that Luby's universal policy appears to outperform all the rest on this particular set of benchmarks. Given that the optimality of Luby's policy was originally proved for Las Vegas algorithms, this empirical result provides motivation for extending the theoretical study of Luby's policy to clause learning algorithms. Interestingly, Siege's policy of (16000, 1) achieved a performance close to that of Luby's, which appears consistent with the fact that the Siege solver itself exhibited a performance similar to TINISAT-L on these benchmarks (we do not have an analytical interpretation of this observation though).

To give a more concrete picture of the impact of the restart policy on the efficiency of the solver, we present detailed results in Tables 2 and 3 for two of the benchmark families where TINISAT solved all instances using every policy. For space constraints we only include three policies in each table: the no-restarts policy and the worst and best of the rest. Results on Siege are also included as a reference point.

In Table 2, we observe that Luby's policy outperformed the other two by more than a factor of two. While Siege outperformed TINISAT-L on these instances, we observe that TINISAT-L made fewer decisions and generated significantly fewer conflicts than Siege, suggesting the effectiveness of Luby's policy in guiding the solver toward shorter unsatisfiability proofs (these are all unsatisfiable instances except the last one).

In Table 3, we observe that Luby's policy again outperformed the other two by a large margin, and TINISAT-L significantly outperformed Siege, which failed to solve three of the instances. On instances they both solved, we observe that while TINISAT-L sometimes made more decisions (reflecting the overhead of restarts), it consistently generated fewer conflicts than Siege, suggesting the effectiveness of the restart policy in guiding the solver toward earlier solutions (these are all satisfiable instances).

Finally, we note that TINISAT coupled with Luby's policy (and in fact any of the other nontrivial policies) also significantly outperformed MiniSat v1.14 on these benchmarks (see Table 1). We view this as supporting evidence for our argument that the decision heuristic together with the restart policy largely determines the efficiency of a clause learning SAT solver when the learning scheme is fixed (to our knowledge MiniSat also uses 1-UIP learning).

[6] We note that this differs from similar scoring methods used by other solvers such as Chaff II and MiniSat v1.14.

[7] These five families contain a total of 357 instances, which were solved by Siege v4 in 428 seconds, TINISAT-L in 506 seconds, MiniSat v1.14 in 661 seconds, and Chaff II in 1971 seconds.

Table 1: Overall results on running TINISAT with seven restart policies: N = No restarts; M = (100, 1.5); Z = (700, 1); B = (550, 1); G = (32, 1.1); S = (16000, 1); L = (Luby's, unit=32). Cutoff was 2 hours.

Benchmark Family	Number of Instances	Size (MB)	Number of Instances Solved								
			N	M	Z	B	G	S	L	MiniSat	Siege
dlx-iq-unsat-1.0	32	4543	10	32	32	32	32	32	32	23	17
engine-unsat-1.0	10	62	7	7	8	7	7	7	7	7	7
fvp-sat.3.0	20	414	1	16	9	9	18	17	14	11	20
fvp-unsat.1.0	4	4	4	4	4	4	4	4	4	4	4
fvp-unsat.2.0	22	74	22	22	22	22	22	22	22	20	22
fvp-unsat.3.0	6	243	0	0	0	0	0	1	0	0	5
liveness-sat-1.0	10	1264	6	5	8	9	6	4	7	7	6
liveness-unsat-1.0	12	1134	4	4	4	4	4	4	4	4	4
liveness-unsat-2.0	9	537	3	3	3	3	3	3	3	3	3
npe-1.0	6	695	2	4	3	4	3	4	3	3	3
pipe-ooo-unsat-1.0	15	1452	6	7	7	7	7	7	7	3	8
pipe-ooo-unsat-1.1	14	775	7	8	8	8	8	8	8	3	9
pipe-sat-1.0	10	1662	9	6	9	10	5	9	10	6	10
pipe-sat-1.1	10	984	9	7	10	10	10	10	10	8	10
pipe-unsat-1.0	13	989	8	8	9	8	8	8	9	4	12
pipe-unsat-1.1	14	760	9	9	11	11	11	10	11	4	14
vliw-sat-2.0	9	1611	5	8	5	6	8	9	9	6	9
vliw-sat-2.1	10	3680	4	2	1	2	5	6	5	2	2
vliw-sat-4.0	10	3076	10	10	10	10	10	10	10	10	7
vliw-unsat-2.0	9	695	0	1	1	1	2	2	2	0	7
vliw-unsat-3.0	2	124	0	0	1	0	0	0	2	0	2
vliw-unsat-4.0	4	405	1	1	1	1	1	1	1	0	1
Total	251	25180	127	164	166	168	174	178	180	128	182
Total Time on All Instances (days)			11.28	8.85	8.49	8.56	8.00	7.80	7.68	11.45	7.45

Table 2: Detailed results for fvp-unsat-2.0, all unsatisfiable except 7pipe_bug. Abbreviations: Dec. (decisions), Con. (conflicts), Res. (restarts). Times are in seconds. Three policies included: Policy N (no restarts) and the worst and best of the rest.

Benchmark	Number of		TINISAT-N				TINISAT-M				TINISAT-L				Siege		
	Vars	Clauses	Dec.	Con.	Res.	Time (s)	Dec.	Con.	Res.	Time (s)	Dec.	Con.	Res.	Time (s)	Dec.	Con.	Time (s)
2pipe	892	6695	3014	1170	0	0.04	3174	1287	4	0.05	2938	1066	16	0.04	4155	1741	0.06
2pipe_1_ooo	834	7026	2509	1089	0	0.04	3147	1387	5	0.06	2766	1210	19	0.05	4541	2423	0.10
2pipe_2_ooo	925	8213	2895	1371	0	0.06	3085	1356	5	0.06	3133	1412	22	0.06	4552	2269	0.09
3pipe	2468	27533	19265	6566	0	0.54	22157	8404	9	0.74	20805	7599	86	0.72	25197	8718	0.53
3pipe_1_ooo	2223	26561	11702	5455	0	0.45	12629	5245	8	0.44	15806	6257	65	0.59	16073	7620	0.49
3pipe_2_ooo	2400	29981	16688	7435	0	0.70	23334	9943	9	1.04	23500	10164	117	1.14	23858	9465	0.69
3pipe_3_ooo	2577	33270	22712	10182	0	1.09	27407	12487	10	1.37	23423	9038	100	1.05	26745	11997	1.01
4pipe	5237	80213	69238	24060	0	5.46	95579	37319	12	8.49	80798	25041	250	6.17	98531	31725	4.03
4pipe_1_ooo	4647	74554	44644	18764	0	3.72	55785	23445	11	5.35	56859	22797	220	5.06	59881	27800	3.51
4pipe_2_ooo	4941	82207	54429	22013	0	5.14	71958	29928	12	7.36	74383	26652	253	6.70	70755	30660	4.15
4pipe_3_ooo	5233	89473	65850	22820	0	5.74	88936	34633	12	9.50	76519	23385	227	6.22	82797	33116	4.93
4pipe_4_ooo	5525	96480	73743	30316	0	8.33	169434	81833	14	35.62	88729	29030	254	8.17	90161	36656	5.53
5pipe	9471	195452	101530	18176	0	6.03	110712	14569	10	4.27	131655	14135	126	4.67	139115	15618	2.58
5pipe_1_ooo	8441	187545	94343	30709	0	11.49	117321	37638	12	18.16	104141	31560	254	12.43	146601	58115	13.55
5pipe_2_ooo	8851	201796	97195	33725	0	13.58	140816	48862	13	29.24	123345	33414	268	15.02	145892	55019	13.60
5pipe_3_ooo	9267	215440	123283	33839	0	17.53	149626	48628	13	28.17	120410	30881	254	13.96	133506	47582	12.06
5pipe_4_ooo	9764	221405	216620	78339	0	52.23	265416	87930	15	56.29	236519	64407	509	35.57	254647	90128	23.14
5pipe_5_ooo	10113	240892	116861	37552	0	17.99	141425	42796	13	24.41	121361	30269	254	15.50	150600	47137	12.47
6pipe	15800	394739	577706	208817	0	273.60	571806	184524	16	192.17	453483	88171	640	73.44	518038	91997	28.77
6pipe_6_ooo	17064	545612	496762	124712	0	181.56	463974	125891	15	153.58	404756	85444	636	81.42	485418	146967	63.59
7pipe	23910	751118	1188862	388233	0	764.32	1168826	318819	18	497.61	876368	130092	998	171.05	997777	131085	59.19
7pipe_bug	24065	731850	166382	17509	0	13.89	542511	143273	16	119.26	144616	14526	131	10.27	282794	12309	4.21
Total			3566233	1122852	0	1383.53	4249058	1300197	252	1193.24	3186313	686550	5699	469.3	3761634	900147	258.25

Table 3: Detailed results for vliw-sat-4.0, all satisfiable. Abbreviations: Dec. (decisions), Con. (conflicts), Res. (restarts). Times are in seconds. Three policies included: Policy N (no restarts) and the worst and best of the rest. Cutoff was 2 hours.

Benchmark	Number of		TINISAT-N				TINISAT-S				TINISAT-L				Siege		
	Vars	Clauses	Dec.	Con.	Res.	Time (s)	Dec.	Con.	Res.	Time (s)	Dec.	Con.	Res.	Time (s)	Dec.	Con.	Time (s)
bug1	521188	13378641	9881446	604996	0	2199.98	8521585	49471	3	242.00	6064544	22613	220	236.31	4138878	26364	154.52
bug2	521158	13378532	7680490	267252	0	712.04	6189255	17318	1	118.74	6274425	11959	125	174.57	3185190	16616	102.06
bug3	521046	13376161	4447142	2098	0	57.72	4447142	2098	0	57.75	3327664	2781	36	76.90	4460466	24509	153.75
bug4	520721	13348117	7670838	563100	0	2045.31	6246054	43329	2	229.49	5710694	16378	156	187.37	4092308	19896	132.98
bug5	520770	13380350	7754522	86586	0	582.81	7459833	29041	1	190.47	4620905	7980	92	133.09	–	–	–
bug6	521192	13378781	8808865	362411	0	1246.53	7811661	44263	2	219.33	5449328	13250	126	160.90	4138295	37005	199.82
bug7	521147	13378010	7893893	595572	0	1603.23	6125501	19414	1	115.87	4070162	8026	92	120.43	4008315	15148	103.39
bug8	521179	13378617	7793145	340079	0	1557.75	6166611	38239	2	216.89	4564989	10454	122	151.55	–	–	–
bug9	521187	13378624	6045105	40574	0	289.25	6871992	46122	2	284.25	3547174	5713	62	97.72	4475278	30692	190.22
bug10	521182	13378625	6935993	44236	0	252.06	8242265	56847	3	354.18	4681271	10683	123	150.14	–	–	–
Total			74911439	2906904	0	10546.68	68081899	346142	17	2028.97	48311156	109837	1154	1488.98			11836.74

Related Work

While in this work we have focused on restart policies for clause learning, in previous work researchers have studied the use of restarts in combinatorial search algorithms. Fixed-interval as well as dynamic restart policies, for example, were studied in (Gomes, Selman, & Kautz 1998; Kautz *et al.* 2002) using a randomized version of Satz (Li & Anbulagan 1997), a SAT solver based on pure DPLL search (without clause learning). Another example is found in (Walsh 1999), where a geometric policy was found to be particularly effective for search problems that exhibit a "small world" topology.

These previous studies of restart policies were based on the observation that for many combinatorial search problems, repeated runs of a randomized search algorithm on the same problem instance (or, similarly, runs of a deterministic search algorithm on random instances of a problem) exhibit a "heavy-tailed" distribution (Gomes, Selman, & Crato 1997), meaning that any run has a nonnegligible probability of taking exponentially longer than all previous runs that have been seen. Intuitively, a restart policy can help combat this unpredictability of search performance by cutting off runs before they take too long in the hope that one of the future runs will succeed quickly.

Interestingly, part of this opportunistic thinking appears to remain valid in the context of clause learning, which as we discussed performs resolution instead of search. After all, one cannot always predict the performance of a clause learning solver on a particular problem instance, either, given its known performance on similar instances (for a quick reminder of this fact, see the performance of Siege in Table 3). We believe, therefore, that the applicability of these previous results to clause learning SAT solvers will be an interesting topic for future work.

Conclusions

Through the design of a simple clause learning SAT solver, we have established a new formulation of modern clause learning as a resolution process guided, among other things, by the decision heuristic and restart policy. This leads to an analytical argument for the importance of the restart policy in clause learning, which we studied empirically by comparing the performance of various policies on a large number of challenging industrial benchmarks. Our results indicate a substantial impact of the restart policy on the efficiency of the clause learning solver. In particular, Luby's universal policy exhibits the best overall performance on the set of benchmarks used. We view our work as a step toward a better understanding of the role played by restarts in clause learning, and hope that it will motivate further work on the design of more effective restart policies, including dynamic ones which we have not considered in this work.

Acknowledgments

Thanks to the reviewers for commenting on an earlier version of this paper. National ICT Australia is funded by the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

References

Beame, P.; Kautz, H. A.; and Sabharwal, A. 2004. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research* 22:319–351.

Berre, D. L., and Simon, L. 2005. The Annual SAT Competitions. http://www.satcompetition.org/.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem proving. *Journal of the ACM* (5)7:394–397.

Eén, N., and Sörensson, N. 2005. MiniSat—A SAT solver with conflict-clause minimization. In *SAT 2005 Competition*.

Goldberg, E., and Novikov, Y. 2002. BerkMin: A fast and robust SAT-solver. In *DATE*, 142–149.

Gomes, C. P.; Selman, B.; and Crato, N. 1997. Heavy-tailed distributions in combinatorial search. In *CP*, 121–135.

Gomes, C. P.; Selman, B.; and Kautz, H. A. 1998. Boosting combinatorial search through randomization. In *AAAI*, 431–437.

Kautz, H. A.; Horvitz, E.; Ruan, Y.; Gomes, C. P.; and Selman, B. 2002. Dynamic restart policies. In *AAAI*, 674–681.

Li, C. M., and Anbulagan. 1997. Heuristics based on unit propagation for satisfiability problems. In *IJCAI*.

Luby, M.; Sinclair, A.; and Zuckerman, D. 1993. Optimal speedup of Las Vegas algorithms. *Information Processing Letters* 47(4):173–180.

Lynce, I., and Silva, J. P. M. 2002. Complete unrestricted backtracking algorithms for satisfiability. In *SAT*.

Marques-Silva, J., and Sakallah, K. 1996. GRASP—A new search algorithm for satisfiability. In *ICCAD*, 220–227.

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *DAC*, 530–535.

Ryan, L. 2004. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, School of Computing Science.

Walsh, T. 1999. Search in a small world. In *IJCAI*, 1172–1177.

Zhang, L., and Malik, S. 2002. The quest for efficient Boolean satisfiability solvers. In *CADE*, Lecture Notes in Computer Science, 295–313.

Zhang, L.; Madigan, C.; Moskewicz, M.; and Malik, S. 2001. Efficient conflict driven learning in a Boolean satisfiability solver. In *ICCAD*.