Curry Eating SAT Solvers (CESS)

Pravin Bhat and Indriyati Atmosukarto
Department of Computer Science and Engineering
University of Washington
Seattle, WA, 98195, USA
pro, indria@cs.washington.edu

Abstract

Boolean satisfiability (SAT) solvers have attracted much research interest, and many types of solvers have been proposed. In this paper, we study and compare two different types of solvers: a systematic solver (DPLL) and a stochastic solver (Walksat), in terms of their runtime performance in the domain of randomly generated "Hard" satisfiability problems. In addition to the implementation and empirical evaluation of the basic versions of these two SAT solvers, we present certain novel heuristics that improve their performance. Experimental results confirm modest performance enhancements derived from our intuitive heuristics.

Introduction

Satisfiability problem is one of the well-known NP-complete problems. While it is believed that the problem is not solvable in polynomial-time in the worst-case, many algorithms and heuristics have been proposed for the problem; Some of which have increased the size of general problems that can be solved in some reasonable time while others have improved performance for certain specialized problem types.

By and large, most SAT solvers come in one of two flavors: systematic solvers and stochastic solvers. A systematic solver is based on backtracking search while a stochastic solver is based on hill-climbing search. This paper presents a comparative study of two solvers, DPLL and Walksat, which are generally considered the representative solvers of the two domains. In addition to studying and comparing the performance of DPLL and Walksat, we implement several improvements to the basic versions of these algorithms.

The remainder of this paper is divided into two main sections. The section on systematic solvers begins with a quick introduction to the basic DPLL algorithm. We then present the heuristics we implemented and the motivation behind them. Empirical comparison of various DPLL heuristics are provided at the end of this section. The second section follows a similar structure summarizing our experimentation with Walksat. Finally we wrap up with a comparison of the two solvers.

Systematic Solver

Introduction to DPLL

DPLL is an algorithm for propositional satisfiability. It is the fastest known algorithm for this problem and unlike purely randomized local search methods it also happens to be complete. The first version of the algorithm was published in 1960 by Martin Davis and Hilary Putnam (Davis & Putnam 1960). However, the algorithm in its present version was formulated in (Davis, Logemann, & Loveland 1962).

DPLL takes as input a sentence in conjunctive normal form or a set of clauses. Using a recursive depth-first search with backtracking it either deems the sentence to be unsatisfiable or outputs an assignment to each literal in the sentence such that the entire sentence is satisfied.

General DPLL algorithm

The algorithm's original version uses the following heuristics: (Davis & Putnam 1960; Davis, Logemann, & Loveland 1962)

1. The algorithm performs early termination during search by detecting whether a sentence must be true or false, given a partially completed model. A clause is true if any one of the literals in the clause is assigned a value of true. Hence, a sentence can be judged to be true even before the model is complete. A clause is false if ever literal in the clause is false. A sentence is false if any one of its clauses is false, which can occur long before the model is complete. This heuristic keeps the algorithm from examining the entire search tree.

2. Given a partial model, assignment of a literal in a unit clause has the highest priority. Unit clauses are clauses in which all literals but one are assigned false by the partial model. This heuristics often leads to unit propagation thus reducing the complexity of the unsolved problem.

3. Given a partial model, a literal that appears in the remaining clauses with a consistent sign is assigned in the absence of unit clauses. Like unit propagation this heuristic prunes the search tree without compromising soundness of the solution generated.

The following is the pseudocode of the DPLL algorithm:
function DPLL($F, literal$)
returns $true$ or $false$

```
Remove clauses containing literal
Shorten clauses containing not(literal)
if F contains no clauses then
  return true
end if
if F contains empty clause then
  return false
end if
if F contains a unit or pure literal L
then
  return DPLL(F, L)
end if
choose V in F
if DPLL(F, V) then
  return true
end if
return DPLL(F, not(V))
```

DPLL Heuristics

The following heuristics were implemented to further enhance the performance of the DPLL-based solver.

Redundant clause removal was performed as a preprocessing step before a search. Redundant clauses are clauses that contain both a literal and its negated form. These clauses are tautologies and can therefore be ignored without compromising soundness.

Next heuristic we implemented was picking a random variable when no unit clauses or pure literals were available.

The third heuristic we implemented was the MOM heuristic. The MOM variable selection strategy (Dubois *et al.* 1993; Pretolani 1993; Freeman 1994) is a heuristic that branches on a variable with Maximum number of Occurrences in Minimum size clauses. Intuitively a MOM variable assignment leads to a high reduction in problem size, which in turn precipitates the discovery of a solution or a dead-end in the current branch.

We designed a fourth novel heuristic, which in combination with MOM beats using MOM by itself, based on the principles taught in this class. We have learned that a good heuristic relaxes the problem at hand and solves the relaxed problem optimally to guide future search. It has been proven that binary propositional sentences (2SAT) can be solved in linear time (del Val 2000). Based on these two observations, a new heuristic was developed which we call the 2SAT heuristic. The 2SAT heuristic involves solving the subset of clauses in every iteration that are 2SAT clauses to proceed further. Using the solution to the 2SAT subproblem, the algorithm jumps ahead in the search tree essentially speeding up the entire search. However, the algorithm will still need to backtrack slowly as a solution to a subset of clauses is not guaranteed to be valid for the entire problem. If the 2SAT subproblem is unsatisfiable, the current branch in the search can be immediately eliminated.

To implement our proposed 2SAT heuristic we searched for current research on solving 2SAT problems in linear time since DPLL in its basic form cannot do so. In our search we came across *B*inSat (del Val 2000) which met our needs. Binsat essentially provides a valid solution to a given 2SAT problem in linear time or returns a literal in constant time that has to be assigned true for the problem to be satisfiable. In the former case we use the solution provided by BinSat to jump ahead and then backtrack slowly if no solution is found. In the later case we treat the literal returned by BinSat as a pure literal since any other assignment to the literal will clause the search to explore an unsatisfiable branch. This also allows our solver to solve purely 2SAT problems in linear time. At the time of writing this paper we have discovered that an approach quite similar to ours has been published by Zheng et al (Zheng & Stuckey 2002). While we are disappointed to realize that we have been beaten to the punch by a recent publication, we take solice in the knowledge that our approach was publication worthy.

Along the same lines we considered solving subsets of Horn clauses during every iteration as a heuristic. However testing showed that the number of Horn clauses, excluding 2SAT clauses which would be handled by the previous heuristic, at any point during the search tends to be small. We considered the possibility that the number of actual Horn clauses during an iteration might be larger given that renaming certain literals can sometimes produce new Horn clauses. Algorithms exist that given a set of clauses can find a valid renaming in linear time to convert the set into Horn clauses if possible. However no known solutions exists to pick the right subset of clauses (2^N possible combinations where N is the number of unassigned variables) from a given set that will result in the best renaming. Unable to arrive at the solution in the time available we eventually decided to drop this pursuit.

DPLL Experiment Results

The following is the specification of the system used to run our test cases: Intel Pentium 4 2.8GHz with 512MB RAM.

Six different experiment sets were conducted to analyze the empirical performance of our DPLL solver and the various heuristics implemented. Experiments were conducted on randomly generated "hard" test cases. "Hard" problems are defined as problems having a clause/variable ratio of 4.3 for 3SAT sentences and 1.0 for 2SAT sentences. Note that not all the CNF sentences used for DPLL testing are satisfiable. In addition, a time limit of 1 hour was set for each run. If the algorithm is not able to produce a solution in the given time limit it will output "Out of Time" instead of the solution.

Table 1 shows the result of the first three experiments. The first experiment results, shown in column DPLL1, quantifies the performance of the basic DPLL algorithm. DPLL1 uses redundant clause removal, unit propagation and pure literal propagation heuristics. The result shows that the algorithm runs in exponential time and fails to find a solution within the given time limit for hard problems with more than 120 variables.

Table 1 column DPLL2 shows the behavior of the DPLL algorithm when it incorporates the random variable heuristic on top of the basic heuristic set (redundant clause removal, unit propagation and pure literal). The results shown in the column is the average time of 5 runs to account for variability in run times. The result shows that the performance of the algorithm improves slightly when compared to DPLL1 but not by much.

Table 1 column DPLL3 shows the performance of the DPLL algorithm using the MOM heuristic for variable selection. From these three experiments, it can be seen that using MOM heuristic for variable selection greatly improves the running time performance of the DPLL algorithm. This result shows just how important proper variable selection heuristic can be in improving performance for larger problems.

Table 2 shows the performance of the DPLL algorithm when it includes all heuristics discussed in the previous subsection. That is, in addition to the basic heuristic set, it includes the selection of MOM variables and the 2SAT heuristic. The table also highlights the size of hard problems our solver can handle in a one hour time frame. For our solver this limit tends to be between 215-230 variables.

Similar experiments were performed on 2SAT sentences. The results of these experiments are shown in table 3. Column DPLL4 shows the results of running DPLL with the MOM variable selection on 2SAT sentences. Table 3 column DPLL5 highlights the strength of the 2SAT heuristic which is able solve "hard" 2SAT problems in linear time. The table also highlights the size of hard problems the solver can handle which happens to about 40000 variables in one hour.

Figure 1 and 2 shows a summary graph describing the runtime performance of the different DPLL heuristics on 3SAT and 2SAT problems respectively. From these figures we can see that our 2SAT heuristic greatly improves the performance of the basic DPLL solver even over the one that uses MOM.

Stochastic Solver

Introduction to Walksat

Walksat (Selman, Kautz, & Cohen 1993) is a stochastic satisfiability solver which assigns all variables a random truth assignment and then attempts to heuristically refine the assignment until the input sentence evaluates to true. There are several variants of the Walksat algorithms, and each of the variants are defined by the specific heuristics used to vary the truth assignments. All variants occasionally ignore their heuristic and make a random refinement according to some fixed probability.

Stochastic satisfiability solvers tend to perform better than systematic satisfiability solvers on randomly generated problem, however this is only true if the problems are satisfiable. In fact, Walksat is unable to determine a given problem is unsatisfiable.

General Walksat algorithm

Walksat is one of the most effective local search algorithms (hill-climbing) to be applied to satisfiability problems. On every iteration, Walksat picks an unsatisfied clause and flips one literal in the clause. There are two ways to pick a literal in a clause: (1) using "min-conflict" steps which minimizes the number of unsatisfied clauses in the new assignment, and (2) "random walk" step that picks the literal randomly.

We implemented a variant of Walksat called *W*alksat-SKC. The algorithm is as follows:

```
function Walksat-SKC
returns truth assignment

  Given a problem instance F, in CNF, and a
  noise setting N
  choose a random truth assignment, S
  while F(S) evaluates to false and
  iteration < maximum number of flips do
    randomly choose an unsatisfied clause c
    in F(S)
    if there is a literal in c whose value
    can be changed without causing any new
    clauses to become unsatisfied then
      let l be this literal
    else
      with probability N choose l in c
      randomly OR
      with probability 1 - N choose l such
      that when its value is changed, the
      smallest number of satisfied clauses in
      F(S) become unsatisfied
    end if
    change the truth assignment of l in S
  end while
```

Additional features of the algorithm includes redundant clause removal before the search begins and random restarts after 10000 iterations.

Walksat Heuristics

Certain problems are known to be sensitive to the noise setting which decides with what frequency Walksat flips the assignment of a literal on random. For such problems a good approach would be to pick this noise level automatically.

We solved this problem by implementing *A*uto-Walksat (Patterson & Kautz 2001) which is a general algorithm that can automatically tune any variant of the Walksat family of stochastic satisfiability solvers. (Patterson & Kautz 2001) claims that the success of a stochastic satisfiability solver to find a problem's solution depends on a parameter setting (called the noise) which determines the trade-off between random decisions and heuristic decisions.

In order to find the optimal noise setting, *A*uto-Walksat uses a combination of Brent's method (Brent 1993) and Golden Section search to find the noise setting that minimizes the invariant ratio, defined later in this section. The objective function value for a stochastic satisfiability solver is the value that is to be minimized during the search for the problem solution. In the case of Walksat-SKC, the objective function value is the number of unsatisfied clauses at the beginning of an iteration. The mean objective function value would then be the average of the objective function value calculated at the end of several unsuccessful iterations. Similarly, the standard deviation of the objective function is the standard deviation of the objective function value. The invariant ratio is defined as the ratio of the mean objective function value to the standard deviation of the objective function.

Based on the observation in (McAllester, Selman, & Kautz 1997), it has been been found that the optimal perfor-

Table 1: DPLL on 3SAT where OT means ”Out of Time”

Data	Sat/Unsat	n	l	Time(sec)		
				DPLL1	DPLL2	DPLL3
CNF7	Unsat	50	215	2	1.2	0
CNF6	Sat	90	387	62	53.4	0
CNF10	Sat	100	430	10	17.8	2
CNF21	Sat	110	473	380	778	3
CNF22	Unsat	120	516	1280	2966.8	17
CNF23	Unsat	130	559	OT	OT	94
CNF24	Sat	140	602	OT	3294.4	0
CNF25	Unsat	150	645	OT	OT	108
CNF26	Sat	160	688	OT	OT	10
CNF27	Unsat	170	731	OT	OT	1037
CNF28	Sat	180	774	OT	OT	8
CNF29	Unsat	190	817	OT	OT	1815
CNF11	Sat	200	860	OT	OT	4

Table 2: DPLL -hBest -t1 on 3SAT where OT means ”Out of Time”.

Data	Sat/Unsat	n	l	Time(sec)
CNF7	Unsat	50	215	0
CNF6	Sat	90	387	0
CNF10	Sat	100	430	2
CNF21	Sat	110	473	3
CNF22	Unsat	120	516	16
CNF23	Unsat	130	559	88
CNF24	Sat	140	602	1
CNF25	Unsat	150	645	86
CNF26	Sat	160	688	9
CNF27	Unsat	170	731	842
CNF28	Sat	180	774	5
CNF29	Unsat	190	817	1386
CNF11	Sat	200	860	3
CNF61	Sat	205	882	108
CNF62	Sat	210	903	2595
CNF63	Sat	215	925	1017
CNF64	Unknown	220	946	OT
CNF65	Sat	225	968	1061
CNF66	Unknown	230	989	OT
CNF67	Unknown	235	1011	OT
CNF68	Unknown	240	1032	OT
CNF69	Unknown	245	1054	OT
CNF30	Unknown	250	1075	OT
CNF31	Unknown	300	1290	OT
CNF32	Unknown	350	1505	OT
CNF33	Unknown	400	1720	OT

Table 3: DPLL on 2SAT where OT means ”Out of Time”.

Data	Sat/Unsat	n	l	Time(sec)	
				DPLL4	DPLL5
CNF9	Sat	12000	12000	131	209
CNF52	Sat	15000	15000	212	240
CNF53	Sat	18000	18000	308	305
CNF50	Sat	20000	20000	384	374
CNF70	Sat	21000	21000	412	204
CNF71	Sat	22000	22000	813	826
CNF72	Sat	23000	23000	906	244
CNF73	Sat	24000	24000	994	260
CNF74	Unsat	25000	25000	1701	76
CNF75	Unsat	26000	26000	OT	148
CNF76	Sat	27000	27000	OT	333
CNF77	Unsat	28000	28000	OT	148
CNF78	Sat	29000	29000	OT	401
CNF51	Sat	30000	30000	OT	436
CNF54	Unsat	40000	40000	OT	418
CNF55	Unknown	50000	50000	OT	OT
CNF56	Unknown	60000	60000	OT	OT
CNF57	Sat	70000	70000	OT	2303
CNF58	Unknown	80000	80000	OT	OT
CNF59	Unknown	90000	90000	OT	OT
CNF60	Unknown	100000	100000	OT	OT

mance of most stochastic satisfiability solvers occurs when the noise level is about ten percent above the setting that minimizes the invariant ratio. Auto-Walksat uses this observation to find the optimal noise setting. Our implementation of Auto-Walksat only uses the Golden Section search to find the minimum invariant ratio.

Walksat Experiment Results

The experiments were conducted in a manner similar to the DPLL experiments described above. However to test Walksat only ”hard” problems that are satisfiable were chosen. In addition, a time limit of 15 minutes was used for each.

Table 4 compares the performance of Walksat and Auto-Walksat on 3SAT problems. Table 5 compares the performance of Walksat and Auto-Walksat on 2SAT problems. The results also highlights the maximum number of variables the solver can handle which is more than 225 variables for 3SAT and more than 6000 variables for 2SAT in a 30 minute time frame.

Figure 3 and 4 shows a summary graph describing the running time performance of the Walksat and the Auto-Walksat heuristics on 3SAT and 2SAT problems respectively.

From the various experiment results, it can be seen that Auto-Walksat does not outperform the basic Walksat solver. Auto-Walksat has a near constant overhead of estimating the right noise setting which accounts for some of the performance difference. However we were surprised to see cases where the auto-tuning process actually hurt our performance. Our guess is that Auto-Walksat improves performance for noise-sensitive problems which generally tend to be non-random real world problems. To confirm that our implementation actually works we tested our solver on the same test set used by (Patterson & Kautz 2001). While these test cases were beyond the limit of our Auto-Walksat solver we made sure that our solver picked the optimal noise setting. Table 6 shows the noise setting picked up by our implementation of Auto-Walksat and those reported in the original Auto-Walksat paper for comparison.

Discussion

This section provides an evaluation of the DPLL and Walksat solvers in terms of runtime performance and maximum number of variables that can be solved in a reasonable time frame.

Table 7 compares basic DPLL and our heuristic laden DPLL to basic Walksat on hard 3SAT problems. The basic Walksat algorithm beats both these solvers on large problem sets. This result confirms the well known fact that Walksat outperforms most variants of DPLL in finding solutions to a randomly generated ”hard” satisfiability problem. However unlike DPLL, Walksat remains incomplete in the sense that it cannot distinguish between unsatisfiable problems and exceptionally hard to solve problems.

On the other hand our 2SAT heuristic allows our DPLL solver to solve 2SAT problems in linear time. For this problem set our DPLL solver outperforms Walksat by an order of magnitude in the number of variables that be solved in a reasonable time frame. The basic Walksat can solve up to 6000 variable 2SAT problems in reasonable time while our version of the DPLL solver can handle upwards of 40000 variables.

Table 4: Walksat on 3SAT where OT means ”Out of Time”.

Data	n	l	Time(sec)	
			Walksat	Auto-Walksat
CNF6	90	387	17	22
CNF10	100	430	1	23
CNF21	110	473	21	47
CNF24	140	602	12	18
CNF26	160	688	47	11
CNF28	180	774	289	OT
CNF11	200	860	67	OT
CNF61	205	882	334	OT
CNF62	210	903	122	633
CNF63	215	925	14	OT
CNF65	225	968	36	195
CNF69	245	1054	22	OT

Table 5: Walksat on 2SAT where OT means ”Out of Time”.

Data	n	l	Time(sec)	
			Walksat	Auto-Walksat
CNF80	100	100	0	0
CNF81	150	150	0	1
CNF82	200	200	0	2
CNF83	250	250	0	2
CNF84	300	300	0	2
CNF85	350	350	1	3
CNF86	400	400	0	4
CNF87	450	450	1	3
CNF88	500	500	0	1
CNF89	600	600	1	6
CNF90	700	700	8	15
CNF100	1000	1000	2	11
CNF101	2000	2000	51	73
CNF102	3000	3000	195	OT
CNF103	4000	4000	111	OT
CNF104	5000	5000	302	OT
CNF105	6000	6000	552	OT
CNF106	7000	7000	OT	OT
CNF107	8000	8000	OT	OT
CNF108	9000	9000	OT	OT
CNF109	10000	10000	OT	OT

Table 6: Optimal noise setting for DIMACS benchmark problems where c is the clause length for each satisfiability problem..

Data	n	l	c	Optimal noise calculated	Original optimal noise (Patterson & Kautz 2001)
f600	600	2550	3	42	50
g125	2250	70163	18	18	20
ssa7552-158	13088	3034	5	35	40

Table 7: Performance comparison between basic DPLL, Walksat, DPLL using all heuristics and Auto-Walksat on 3SAT where OT means "Out of Time".

Data	n	l	Time(sec)			
			DPLL 1	Walksat	DPLL -hBest	Auto-Walksat
CNF6	90	387	62	17	0	22
CNF10	100	430	10	1	2	23
CNF21	110	473	380	21	3	47
CNF24	140	602	OT	12	1	18
CNF26	160	688	OT	47	9	11
CNF28	180	774	OT	289	5	OT
CNF11	200	860	OT	67	3	OT
CNF61	205	882	OT	334	108	OT
CNF62	210	903	OT	122	2595	633
CNF63	215	925	OT	14	1017	OT
CNF65	225	968	OT	36	1061	195
CNF69	245	1054	OT	22	OT	OT

References

Brent, R. P. 1993. Algorithms for minimization without derivatives.

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

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

del Val, A. 2000. On 2-SAT and renamable horn. In *AAAI: 17th National Conference on Artificial Intelligence*. AAAI / MIT Press.

Dubois, O.; Andre, P.; Boufkhad, Y.; and Carlier, J. 1993. Sat versus unsat. In *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*.

Freeman, J. W. 1994. Improvements to propositional satisfiability search algorithms.

McAllester, D. A.; Selman, B.; and Kautz, H. A. 1997. Evidence for invariants in local search. In *Proc. AAAI/IAAI*, 321–326.

Patterson, D. J., and Kautz, H. 2001. Auto-walksat: A self-tuning implementation of walksat. In *Proceedings of SAT2001: Workshop on Theory and Application of Satisfiability Testing*, volume 9.

Pretolani, D. 1993. Solving satisfiability problems: An algorithm implementation challenge? In *Workshop notes, 2nd DIMACS Challenge*.

Selman, B.; Kautz, H. A.; and Cohen, B. 1993. Local search strategies for satisfiability testing. In Trick, M., and Johnson, D. S., eds., *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*.

Zheng, L., and Stuckey, P. J. 2002. Improving sat using 2sat. In *Proceedings of the twenty-fifth Australasian conference on Computer science*, 331–340. Australian Computer Society, Inc.

Figure 1: Running time performance of DPLL on 3SAT

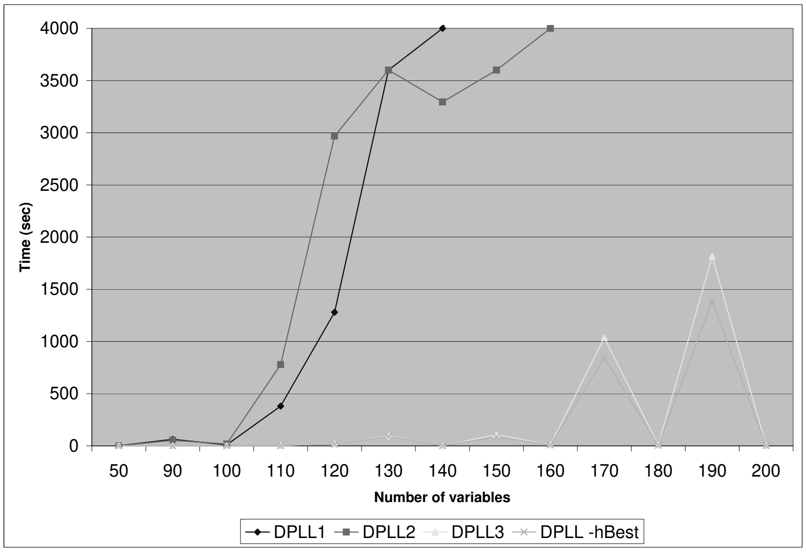


Figure 2: Running time performance of DPLL on 2SAT

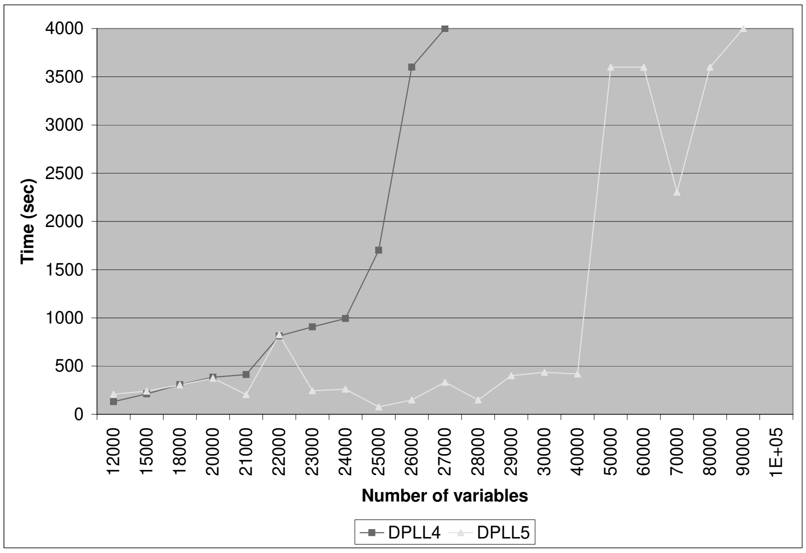


Figure 3: Running time performance of Walksat on 3SAT

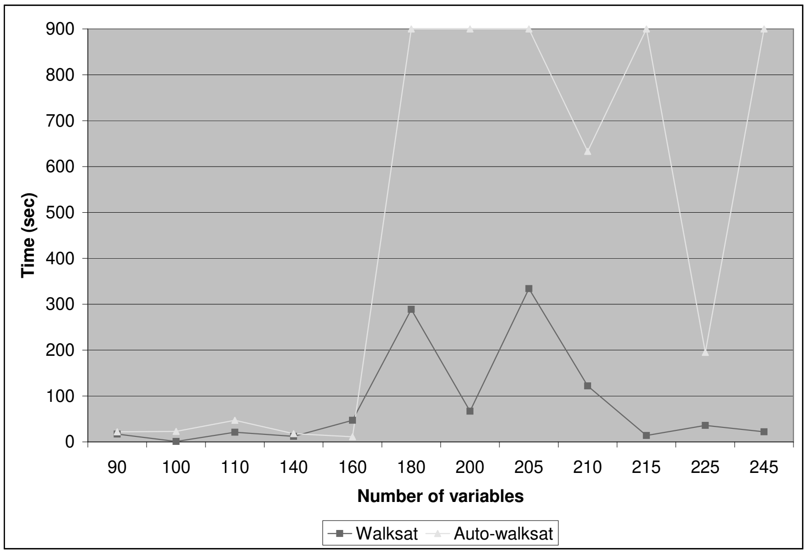
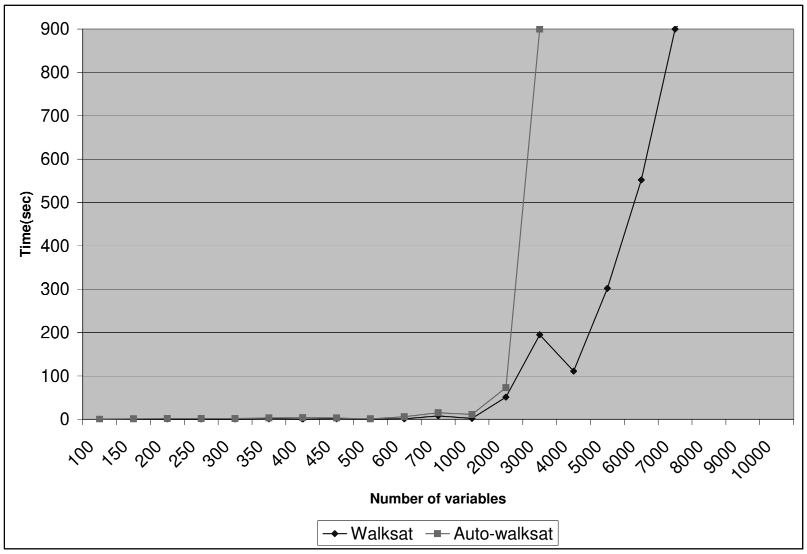


Figure 4: Running time performance of Walksat on 2SAT



Appendix

Division of work:
Pravin: Implementation in C# for DPLL and Walksat, research on heuristics, report.
Indri: Implementation in java for random satisfiability problem generator, generation of test cases, research on heuristics, running experiments, report