SAT-Solving: Performance Analysis of Survey Propagation and DPLL

Christian Steinruecken

June 2007

Abstract

The Boolean Satisfiability Problem (**SAT**) belongs to the class of NP-complete problems, meaning that there is no known deterministic algorithm that can solve an arbitrary problem instance in less than exponential time (parametrized on the length of the input). There is great industrial demand for solving **SAT**, motivating the need for algorithms which perform well.

I present a comparison of two approaches for solving **SAT** instances: DPLL (an exact algorithm from classical computer science) and Survey Propagation (a probabilistic algorithm from statistical physics). The two algorithms were compared on randomly generated 3-**SAT** problems with varying clause to variable ratios.

1 Introduction

Boolean Satisfiability (**SAT**) is a computer science problem on the satisfiability of logical propositions with boolean variables. Given a logical proposition (e.g. $A \wedge \neg B \vee C$), we want to find an assignment of variables making this proposition true.

If such a satisfying assignment exists, the proposition is *satisfiable* (SAT), and otherwise *unsatisfiable* (UNSAT).

Various algorithms have been developed to solve satisfiability problems, and these can be roughly grouped into two kinds: *exact methods*, which are guaranteed to find the correct answer (SAT or UNSAT), and *approximate methods* which do not offer any hard guarantees on whether they will find an assignment or a refutation, and consequently have a third possible return value, UNKNOWN.

Any exact method can be turned into an approximate method by imposing resource limits (e.g. computation time, amount of memory, or number of executed instructions). All known exact **SAT** solvers exhibit exponential time complexity on certain inputs; the existence of a polynomial-time algorithm would decide the $\mathbf{P} = \mathbf{NP}$ hypothesis [3].

1.1 Historical note on SAT

SAT was the first problem to be proved NP-complete (by Stephen Cook in a landmark paper [2] that gained him the Turing Award in 1982), and ultimately caused the emergence of *complexity theory* as a new academic field within computer science.

Due to its industrial applications (e.g. in circuit design, engineering and automated planning tools) [6], there is a lot of practical interest in solving **SAT**.

1.2 Clause form

By convention, **SAT** problems are logical propositions specified in *clause form*, i.e. a product ($\wedge$) of sums ($\vee$) of literals, where

- a *literal* $x_n^{\pm}$ is either a boolean variable or its negation
- a *clause* C_m is a disjunction (logical **OR**) of literals
- the product is a conjunction (logical **AND**) of all the clauses.

Examples

A boolean proposition in clause form might look like this:

$$\{A, \neg B\} \qquad \{B, \neg C, D\} \qquad \{\neg A\}$$

This proposition is satisfiable if there is an assignment of truth values (true or false) to the four variables A, B, C, D that satisfies all three clauses. A clause is satisfied if at least one of its literals is true, e.g. $\{A, \neg B\}$ is true if either $A \mapsto$ true or $B \mapsto$ false (or both).

One solution to this problem is e.g. the assignment $\{A \mapsto \text{false}, B \mapsto \text{false}, D \mapsto \text{true}\}$. This valuation makes the formula true regardless what value is assigned to C.

Here is an example of a proposition which is unsatisfiable:

$$\{\neg G, H\} \qquad \{G\} \qquad \{\neg H\}$$

No matter what we assign to G and H, these clauses cannot be satisfied simultaneously.

Any propositional logic formula can be converted to clause form, though in certain cases this may incur an exponential blow-up in formula size.

Formalisation

In general, a **SAT** problem with N variables and M clauses is specified by two sets X (the set of variables x_n) and P (the set of clauses C_m). A literal x_n^s is then described by a variable $x_n \in X$ and a sign $s \in \{+, -\}$.

The set X^+ is the set of all *positive* literals x_n^+, and X^- the set of all *negative* literals x_n^-. Each clause $C_m \subset (X^+ \cup X^-)$ is simply a set of literals.

Example. The clauses $\{A, \neg B\}$ $\{B, \neg C, D\}$ $\{\neg A\}$ from above are specified as follows:

$$\begin{array}{ll} N = 4, & X = \{x_1, x_2, x_3, x_4\} \\ M = 3, & P = \{C_1, C_2, C_3\} \\ & \\ \text{where} & C_1 = \{x_1^+, x_2^-\}, \\ & C_2 = \{x_2^+, x_3^-, x_4^+\}, \\ & C_3 = \{x_1^-\} \end{array}$$

An *assignment* A is a set of mappings $(x_n \mapsto \mathbf{t})$ from variables $x_n \in X$ to truth values $\mathbf{t} \in \{\text{true}, \text{false}\}$. The notation $A(x_i)$ denotes the truth value assigned to variable x_i under assignment A.

Given an assignment A, the set of clauses P evaluates to the truth value $A(P) \in \{\text{true}, \text{false}\}$ as follows:

$$A(P) = \bigwedge_{1 \le m \le M} \left(\bigvee_{x_i^s \in C_m} A(x_i^s) \right)$$

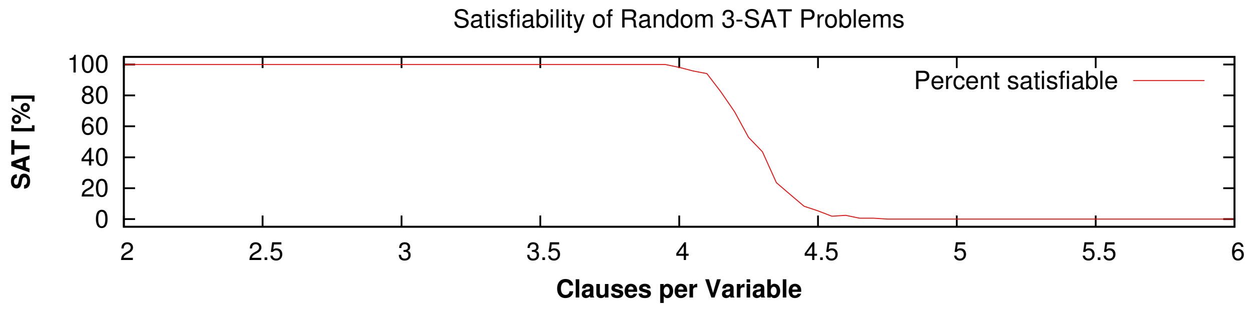
where $A(x_i^+) = A(x_i)$ and $A(x_i^-) = \neg A(x_i)$.

1.3 Structured versus random **SAT** problems

Real-life industrial **SAT** problems are typically big (involving thousands of variables and clauses) and non-random, as they are commonly generated from digital circuit netlists or automated planning tools [9]. One can imagine niche-solvers that optimise their computation by exploiting the structure found in **SAT** problems of their particular niche.

To compare the different **SAT** solving algorithms, I used randomly generated K-**SAT** problems with a given clause to variable ratio $\alpha = \frac{M}{N}$, where N is the number of variables and M the number of clauses (each having exactly K literals). The exact procedure is described in section 3.2.

If we generate a lot of problem sets for some constant N and varying α, one can observe that problems below a certain α value are easy to solve due to being *underconstrained* (therefore easily satisfiable), whereas problems above a certain α threshold are easy to solve due to being *overconstrained* (therefore easily refutable). The "middle region" of α values (between 4.0 and 4.5) is the hard region, and thus particularly interesting for testing the performance of **SAT** solving algorithms.



2 Approaches to **SAT** solving

2.1 DPLL

One of the best known exact algorithms for solving **SAT** is the *Davis-Putnam-Logemann-Loveland* (DPLL) method [5, 4].

The algorithm works by assigning values to boolean variables in turn, and backtracking when it runs into a contradiction. The algorithm terminates with SAT when an assignment is found, or with UNSAT when no backtrack alternatives are left (meaning that the input clauses are unsatisfiable).
DPLL always terminates, reporting either a solution or a contradiction.

The most popular implementation of the DPLL algorithm is zChaff, which has some additional performance tweaks from the Chaff algorithm [7] (a variant of DPLL).

2.2 GSAT and WalkSAT

GSAT and WalkSAT [8] are approximate **SAT** solvers, both belonging to a family of methods employing local search strategies to solve the problem: the algorithm chooses one variable at a time and flips it, until all clauses are satisfied. However, flipping a variable typically satisfies some clauses, but is likely to break others. The art lies in choosing the right variable. GSAT takes a greedy approach and chooses variables that maximise the number of satisfied clauses.

WalkSAT instead includes a bit of randomness, allowing it to also flip variables which are already considered in an optimal state (this gives WalkSAT a chance of escaping from some local optima). Also, it only considers variables from unsatisfied clauses, which further improves the speed of the algorithm.

Generally, both GSAT and WalkSAT can get stuck in local optima, which could prevent them from finding a global solution. The algorithms therefore restart periodically (i.e. discard and randomize the current assignments); if a satisfying assignment is found, the algorithm terminates with SAT, otherwise it keeps running until it exceeded its number of allowed iterations, after which it returns UNKNOWN.

2.3 Survey Propagation

Survey Propagation SAT solving [1] is a heuristic method introduced by A. Braunstein, M. Mezard, and R. Zecchina in 2003.

The central part of this method is an iterative message passing algorithm (SP) which computes *surveys* on a factor graph representation of the set of clauses. These surveys are used in a second algorithm (SID) to fix an assignment for one variable at a time until the surveys suggest no further improvement. At that point, a local search method (e.g. WalkSAT) is invoked on the simplified subproblem.

Note that the naming is a bit confusing: "Survey Propagation" is strictly only the name of the message passing algorithm, which forms part of (but is not identical to) the full **SAT** solving procedure, which is outlined in Figure 1.

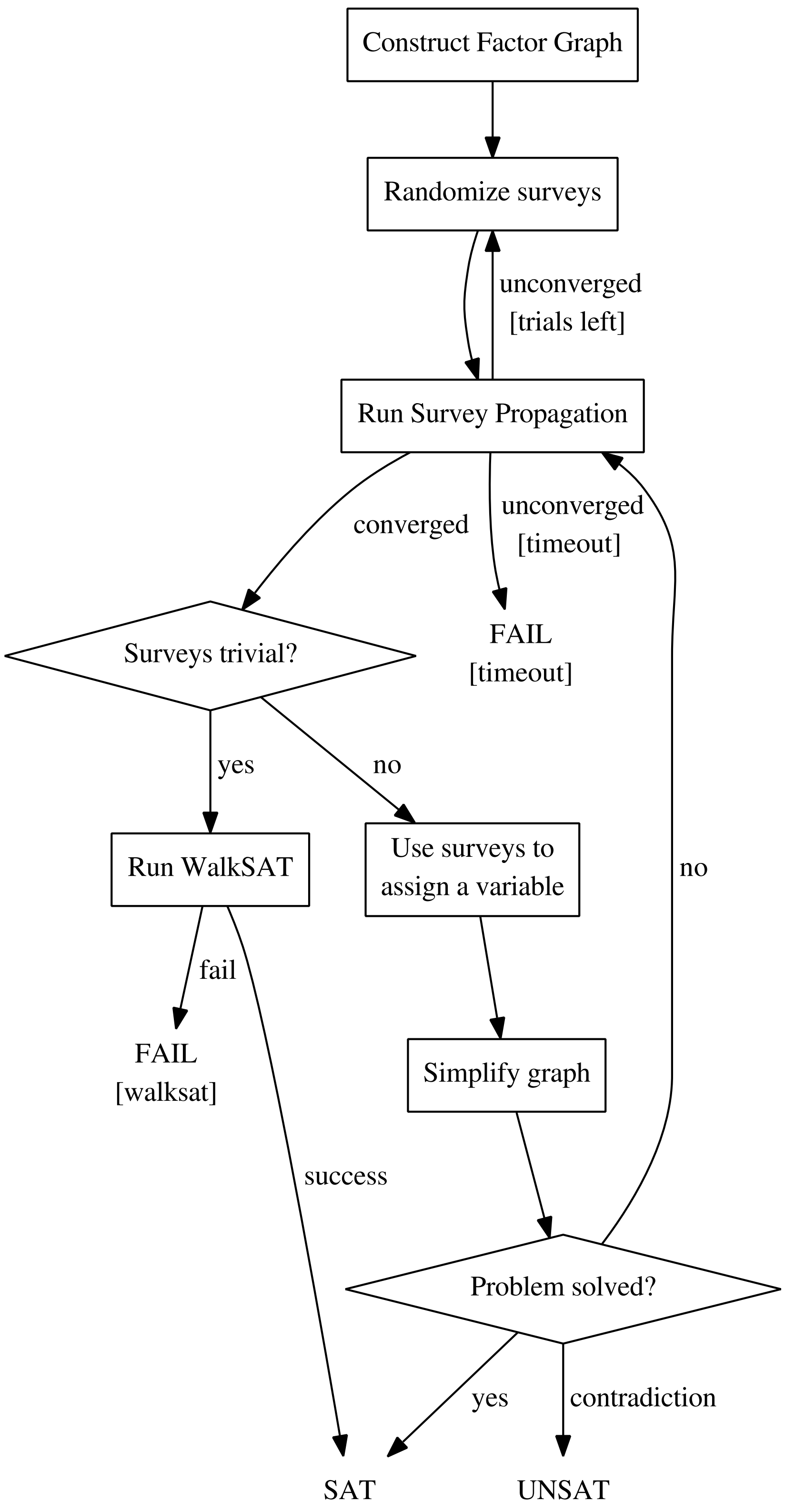


Figure 1: Survey Inspired Decimation (SID)

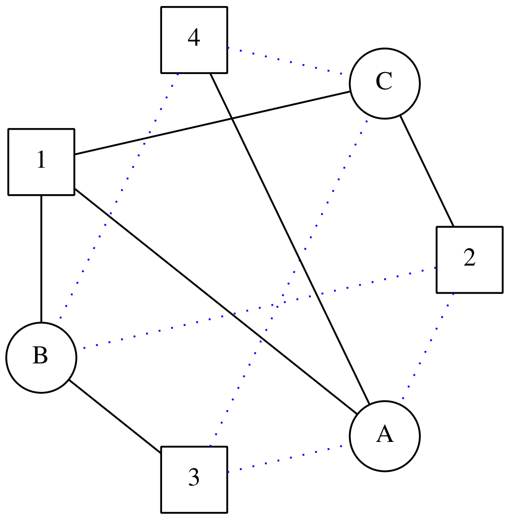
2.3.1 Constructing the factor graph

The factor graph has nodes for each variable and each clause, and edges connecting each variable node with the nodes of clauses it is contained in.

For example, the clause set

$$\{A, B, C\}^1 \qquad \{\neg A, \neg B, C\}^2 \qquad \{\neg A, B, \neg C\}^3 \qquad \{A, \neg B, \neg C\}^4$$

is represented by the following factor graph:



The boxes are function nodes representing clauses, and the circular nodes are variables. Since literals are either negated or unnegated variables, there are two types of edges in the graph. Dotted edges indicate that the variable occurs negated in the clause it is connected to.

More formally, the factor graph can be specified exactly with an edge relation $J \in ([1..N] \times [1..M] \times \{+1, -1\})$, which contains an element $(n, m, \pm 1)$ for each edge in the graph, indicating a connection between variable x_n and clause C_m, with the type of edge (negated or unnegated).

A variable node $\textcircled{n}$ and a clause node $\boxed{m}$ are connected with an edge if and only if $x_n^+ \in C_m$ or $x_n^- \in C_m$. For connected pairs (n, m) we can define $J_m^n \in \{+1, -1\}$, indicating whether x_n appears negated or unnegated in C_m:

$$J_m^n = \begin{cases} +1 & \text{when} \quad x_n^+ \in C_m \\ -1 & \text{when} \quad x_n^- \in C_m. \end{cases}$$

2.3.2 SP Message Passing

Survey Propagation iteratively computes *surveys* $\eta_{m \to n}$ which are sent along the edges of the graph.

For each variable node n, the following values are updated repeatedly:

$$\Pi^{\mathrm{u}}_{n\to m} := \left(1 - \prod_{b\in V^{\mathrm{u}}_m(n)} (1-\eta_{b\to n})\right)\left(\prod_{b\in V^{\mathrm{s}}_m(n)} (1-\eta_{b\to n})\right) \tag{1}$$

$$\Pi^{\mathrm{s}}_{n\to m} := \left(1 - \prod_{b\in V^{\mathrm{s}}_m(n)} (1-\eta_{b\to n})\right)\left(\prod_{b\in V^{\mathrm{u}}_m(n)} (1-\eta_{b\to n})\right) \tag{2}$$

$$\Pi^{0}_{n\to m} := \prod_{b\in V(n)\setminus m} (1-\eta_{b\to n}) \tag{3}$$

$$\eta_{m\to n} := \prod_{i\in V(m)\setminus n} \left(\frac{\Pi^{\mathrm{u}}_{i\to m}}{\Pi^{\mathrm{u}}_{i\to m} + \Pi^{\mathrm{s}}_{i\to m} + \Pi^{0}_{i\to m}}\right) \tag{4}$$

Here $V(n)$ denotes the set of clause nodes neighbouring variable node n, and $V(m)$ denotes the set of variable nodes neighbouring clause nore m.

$V^{\mathrm{s}}_m(n)$ denotes the set of variable nodes $b \in V(m)$ with $J^b_m = J^n_m$ (i.e. same edge type) , and $V^{\mathrm{u}}_m(n)$ denotes the set of variable nodes $b \in V(m)$ with $J^b_m \neq J^n_m$ (opposite edge type).

This process continues until either the computation *converges* (i.e. the change between successive updates drops smaller than some constant ϵ) or until a timeout is reached.

2.3.3 Survey Inspired Decimation

SID is a procedure that simplifies the **SAT** factor graph using Survey Propagation messages as a heuristic.

1. Construct the *factor graph* of the problem from its set of clauses.
2. Repeat
   (a) Call *Survey Propagation* (SP) to compute surveys η. If SP doesn't converge, retry until a timeout is reached.
   (b) If there are non-zero surveys, ($\eta \neq 0$):
      i. Compute *biases* for each variable i, and sort them.
      ii. Find the variable with highest bias, and assign it.
      iii. Simplify the graph.
   (c) Otherwise (if all surveys are zero), call WalkSAT and exit.

   until the problem is *solved* or a *contradiction* is found.

2.4 Others

There are several other methods of solving **SAT**. For instance:

Truth Table Enumeration

The brute force way of solving **SAT**, TTE exhaustively enumerates all possible assignments of the variables and checks whether any of them satisfies the set of clauses. It runs in guaranteed exponential time $O(2^n)$ on all inputs.

Resolution

A method which finds and merges clauses with a common variable of opposite sign, thus reducing the number of clauses and variables until either an empty clause is generated (providing a refutation) no further reductions can be made (meaning that the result is satisfiable).

Ordered binary decision digrams

OBDDs convert logical formulae to a unique and compact tree representation. The uniqueness feature is its main strength, making it widely employed for model checking and logic minimisation. A formula is satisfiable if and only if its OBDD is unequal to the node representing *false*.

3 Setup

3.1 Implementations

To compare the DPLL and Survey Propagation **SAT** solvers, I downloaded reference implementations for each and modified them to include (identical) time auditing mechanisms, before compiling and linking them against the same system libraries.

I ran the resulting binaries on designated, identical 3-**SAT** problem sets, which were randomly generated using the procedure outlined below.

3.2 Problem generator

To generate random K-**SAT** problem instances with M clauses and N variables, the following method was used:

1. For $n := 1$ to N, allocate a new variable x_n.
2. For $m := 1$ to M, generate clause C_m as follows:
    (a) Initialise $C_m := \{\}$
    (b) Repeat K times:
        i. Pick a boolean variable x_n uniformly from $X \setminus C_m$, i.e. the set of variables not yet occurring in C_m.
        ii. Choose a sign $s \in \{+, -\}$ with probability $\frac{1}{2}$.
        iii. Add the new literal x_n^s to the clause. $C_m := C_m \cup \{x_n^s\}$

3.3 Comparison

I ran the solver binaries on the precomputed problem sets, recording the computation time t and outcome $o \in \{\mathsf{SAT}, \mathsf{UNSAT}, \mathsf{UNKNOWN}\}$ for each instance, and these data to plot graphs showing the relationships between t, $\alpha = \frac{M}{N}$ and $\Pr(o = \mathsf{SAT})$ for each algorithm.

4 Results

The following graphs show medians and quartiles of the performance of zChaff, WalkSAT and Survey Propagation, computed with 170 problem samples for any given clause to variable ratio $\alpha = M/N$.

Each plots includes a graph showing the number of assignments found by the given algorithm. The step size used for α was 0.05.

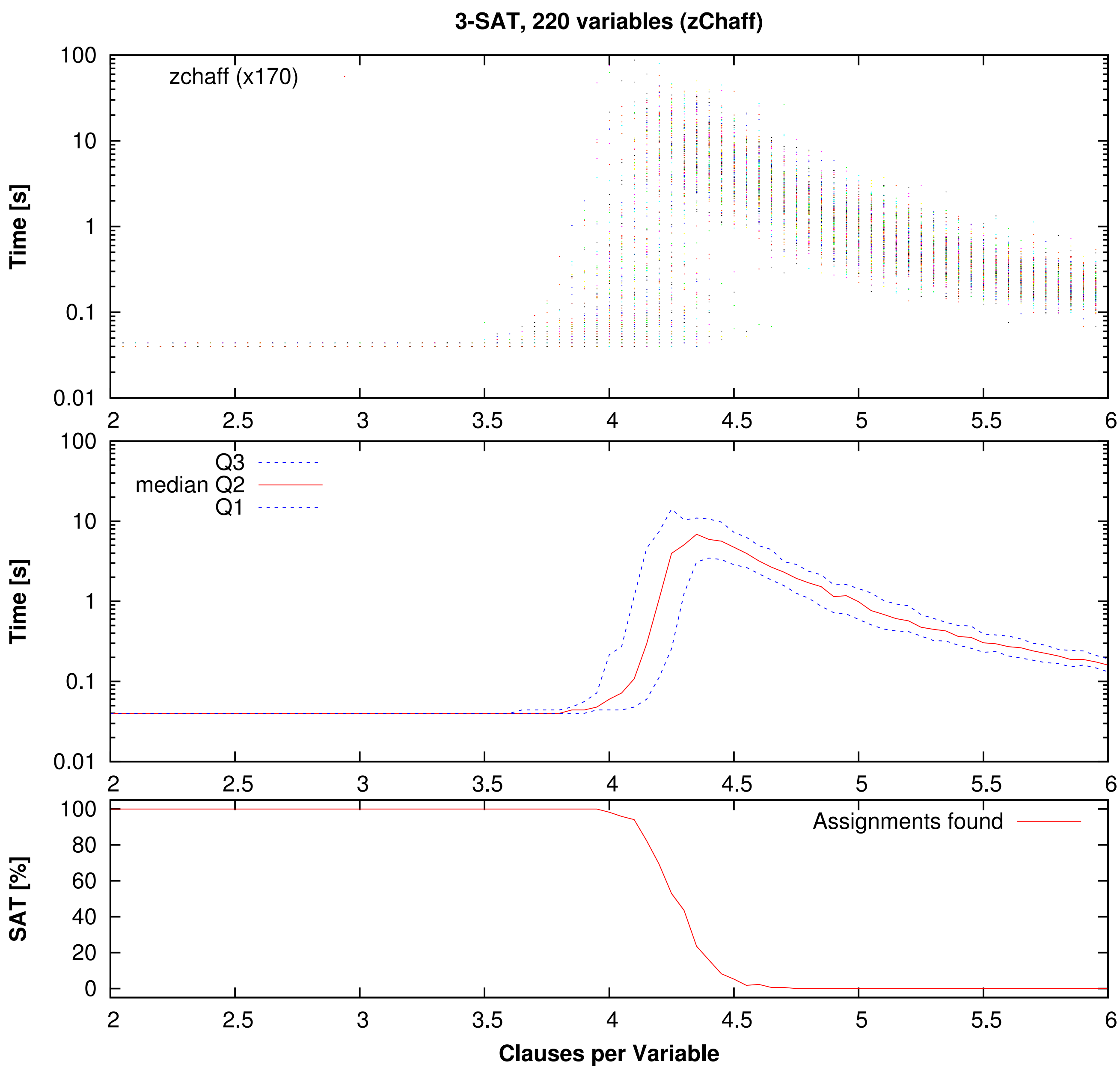


Figure 2: The graph on top shows the performance of individual **SAT** instances as measured by zChaff - each marked by a point in the graph.

The middle graph shows quartiles and median of the same data.

Finally, the bottom graph shows what proportion of problems the algorithm could satisfy for a given value of α. Since Chaff / DPLL is an exact algorithm and was not time bounded in this run, the data coincide with actual satisfiability (i.e. all counted negative samples were proved UNSAT rather than left UNKNOWN.

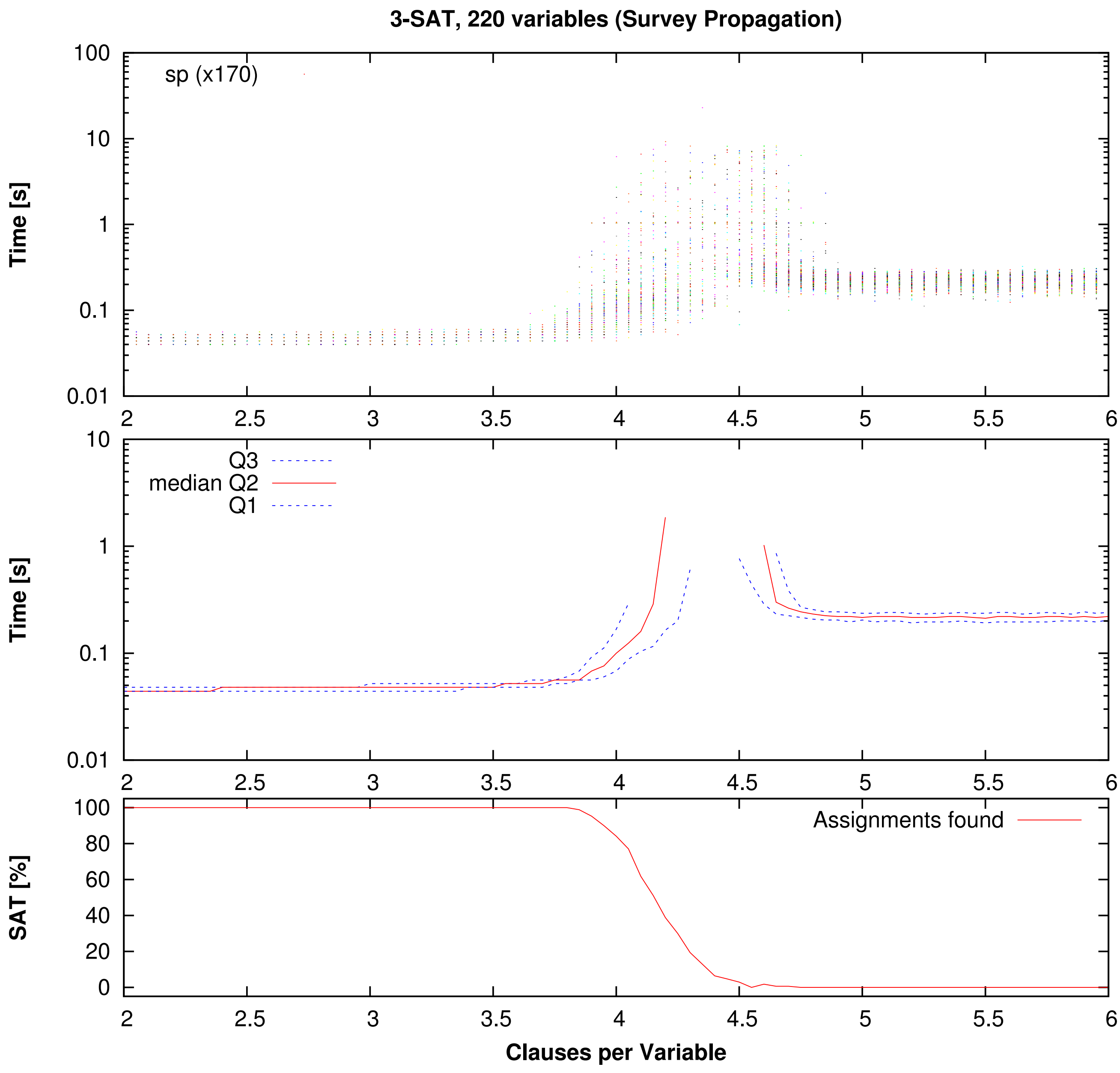


Figure 3: Survey Propagation's performance on the exactly same problem set.

The bottom graph shows what proportion of problems Survey Propagation could find assignments for, and note that this may be less than the number of assignments actually existing.

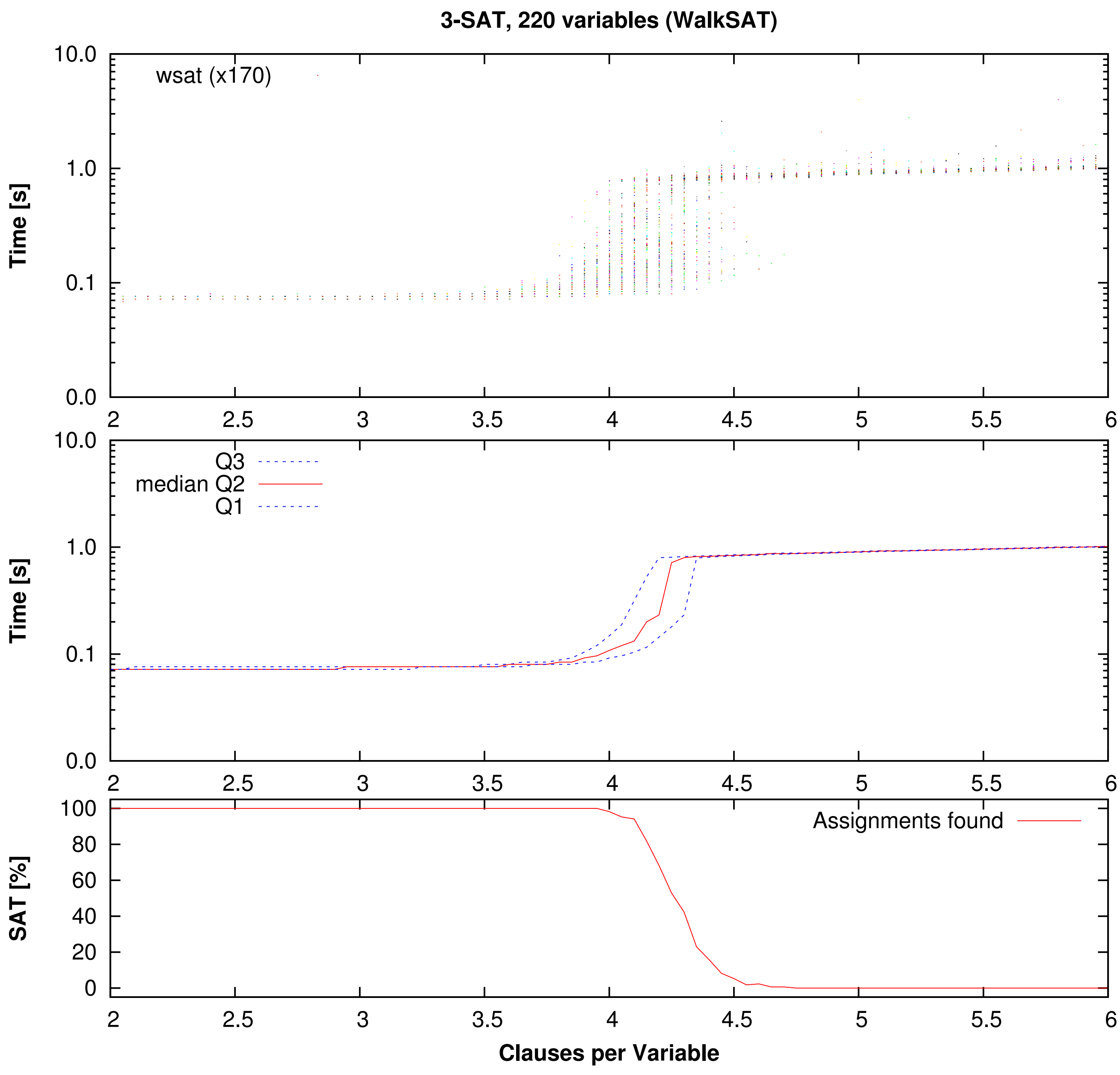
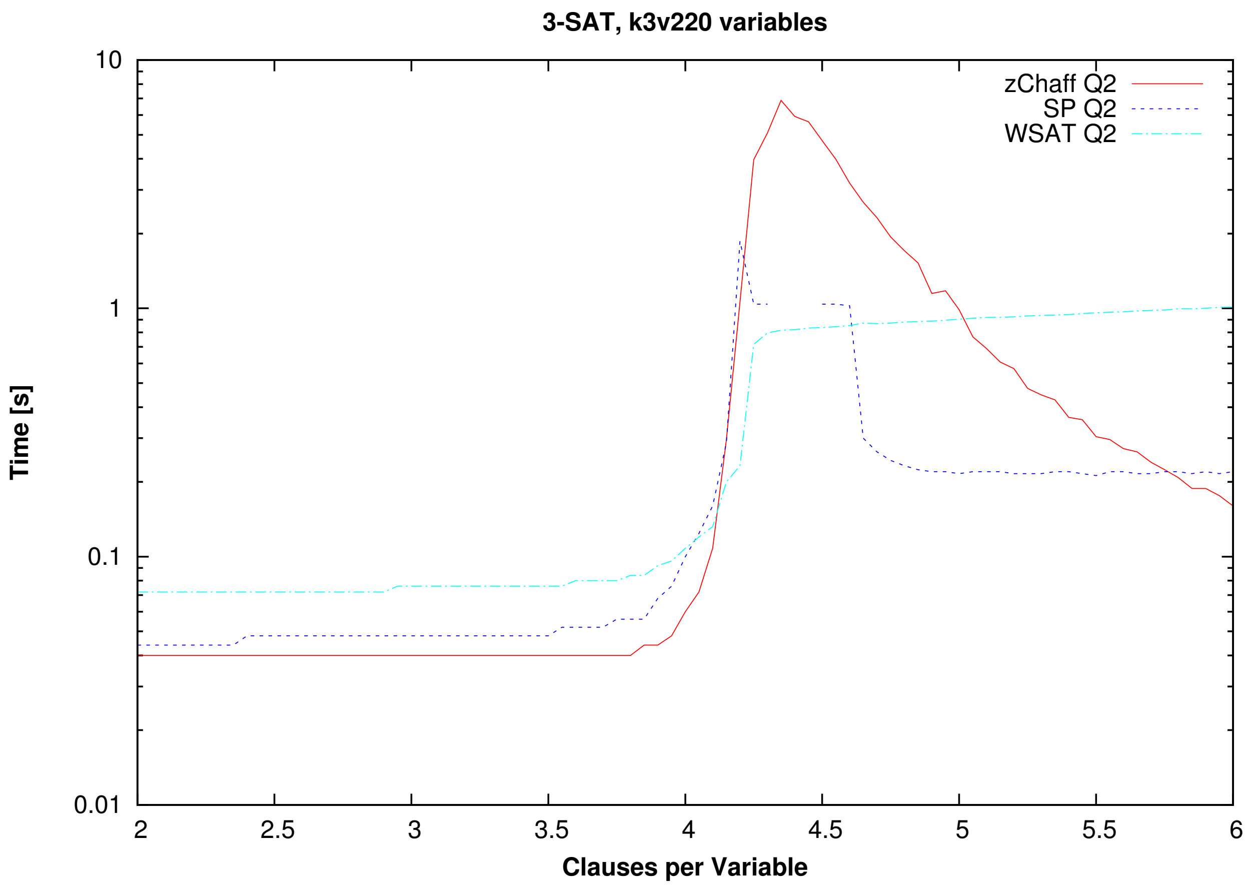


Figure 4: WalkSAT's performance on the exactly same problem set.



This graph shows the medians of all three algorithms (Chaff, SP and WSAT) in the same coordinate system. Note that Survey Propagation's line is interrupted where the majority of samples terminated with UNKNOWN.

5 Conclusions

I compared the DPLL / Chaff and Survey Propagation algorithms using relatively small **SAT** problems (involving no more than 300 variables); the problems were small enough to allow each instance to be solved exactly (by finding an assignment or refutation), and big enough to expose interesting differences between the algorithms.

In their original paper on Survey Propagation [1], the authors warn that, on small problem instances, their algorithm often fails to converge. (They define a "small" problem to be one which has approximately $N \approx 1000$ variables).

My experiments confirm that there are convergence problems in the region where the clause to variabe ratio α lies between 4.1 and 4.6. Only Chaff succeeds in this interval, though at the expense of reaching its computation-time peak at $\alpha \approx 4.25$.

Most notably, however, SP is significantly better than both DPLL and WalkSAT on problems where $\alpha \in [4.7, 5.5]$.

It is important to bear in mind that the performance properties of these algorithms on small problems may well be different from their performance on bigger problems. Also, this study gives little clue as to which algorithm wins on industrial problems, which are likely to be structurally different from uniformly generated random **SAT** instances.

5.1 Further work

As already indicated above, it would be desirable to run these comparisons with much bigger problem sets, and compare the performance as the number of variables increases.

It would also be interesting to test the algorithms on actual industrial problems, or generate **SAT** problem sets that are deliberately less random, or incorporate different kinds of structural constraints.

Also, it might be useful to benchmark Survey Propagation in more detail, e.g. audit the time spent in SID and WalkSAT, and maybe collect and plot statistics about the types of FAIL states SID returns.

Finally, using something other than WalkSAT as local search method in the decimation process might give some more insights on how it impacts overall performance and much credit should be attributed to the local search method compared to the time it takes for SP to converge.

References

[1] Alfredo Braunstein, Marc Mézard, and Riccardo Zecchina. Survey Propagation: An algorithm for satisfiability. *Random Structures and Algorithms*, 27(2):201–226, 2005.

[2] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on theory of computing*, pages 151–158. ACM, 1971.

[3] Stephen A. Cook. The P versus NP problem. In *The Millenium Prize Problem.* Clay Mathematical Institute, April 2000.

[4] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[5] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[6] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.

[7] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM Press, 2001.

[8] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In Michael Trick and David Stifler Johnson, editors, *Proceedings of the Second DIMACS Challenge on Cliques, Coloring, and Satisfiability*, pages 521–532, 1993.

[9] Ji-Ae Shin and Ernest Davis. Processes and continuous change in a SAT-based planner. *Artificial Intelligence*, 166(1-2):194–253, 2005.