Report on a SAT competition

M. Buro / H. Kleine Büning

FB 17 – Mathematik/Informatik
Universität Paderborn

Bericht Nr. 110
Reihe Informatik
November 1992

Report on a SAT competition

Michael Buro, Hans Kleine Büning
FB 17 – Mathematik/Informatik
Universität Paderborn
Postfach 1621
D–4790 Paderborn (Germany)
E–mail: kbcsl@uni–paderborn.de

Abstract

We present the results of a SAT competition organized in 1991/92 at the University of Paderborn. Here we asked for programs solving the satisfiability problem for formulas in conjunctive normal form.

1 Introduction

Several calculi and algorithms have been developed in order to solve the NP–complete [Coo71,GaJo79] satisfiability problem for propositional formulas, i.e. to decide whether or not there is a truth value assignment satisfying the given formula.

A lot of papers have been published investigating the pros and cons of the different approaches. Some of them compare the minimal proof length of proof systems like resolution, cutting plane etc. and look for formulas which are hard to decide. Another approach is the average case analysis which has been carried out mainly for Davis–Putnam [DaPu60] algorithms. A few papers contain experimentally results discussing for which formulas one algorithm is better than another algorithm.

We were looking for the *quickest* program solving the satisfiability for formulas in conjunctive normal form. Obviously, what quickest program means, depends essentially on the set of formulas we use for the evaluation. We randomly generated a set of formulas which fulfill some properties and the quickest program has been determined by adding the time the programs needed for deciding the satisfiability of these formulas. In order to avoid the difficulties with programs written in different programming languages, we asked for programs written in the C programming language. We acknowledge gratefully the sponsorship of the competition by IBM Germany.

2 Terms of participation

The submitted programs for deciding the satisfiability of CNF formulas had to be implemented in the Kernighan/Ritchie or ANSI C programming language. The documented source code was accepted as an ASCII textfile on a 3 1/2 " MS DOS diskette. Data or object files were not allowed. The module was compiled and linked to a test program which firstly called `init_sat()`. Here, some data could be initialized. Then, `sat()` was called several times to decide respectively the satisfiability of a CNF formula. The starting address of the character string which encodes the formula in question was given to `sat()`. The return value had to be 0 if the formula was not satisfiable. Otherwise, `sat()` had to yield a value not equal to 0. The formulas had the following syntax:

formula	$\rightarrow$	clause `'\0'` | clause formula
clause	$\rightarrow$	`'('` literalseq `')'`
literalseq	$\rightarrow$	literal | literal `','` literalseq
literal	$\rightarrow$	variable | `'-'` variable

Variables were encoded as decimal numbers in the range of 1 to 1000. For example, the `'\0'` terminated C character string `"(-1,2,-3)(4,-1)(2)(103,103,2)"` is a formula. It describes the Boolean formula $(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_4 \vee \bar{x}_1) \wedge (x_2) \wedge (x_{103} \vee x_{103} \vee x_2)$ in a compressed form.

3 Test formulas

In order to determine the speed of the algorithms, we generated seven groups of formulas with 40 CNF formulas each. Groups 1..6 consisted of formulas with clauses of constant length (3..8), variables were negated with probability 1/2, there was no double occurrence of variables in a clause, and the clauses were uniformly distributed. In group 7 the clause lengths varied according to a special distribution (see below) and each literal in a clause was uniformly distributed. The number of variables and clauses was chosen empirically such that in each group formulas were satisfiable with probability $\approx 1/2$ and solvable in approximately equal times using our SAT solver which is a Davis–Putnam implementation. Due to the enormous differences of running time, we used two sets of test formulas to get a ranking of more than six participants. Here are the parameter sets for the seven times two groups:

Groups 1..6

	Clause length	3	4	5	6	7	8
hard	#Variables	215	87	55	40	32	27
	#Clauses	920	860	1,163	1,745	2,807	4,831

	Clause length	3	4	5	6	7	8
easy	#Variables	120	48	29	21	16	13
	#Clauses	510	487	621	924	1,460	2,325

Group 7

	–	110 variables and 2175 clauses
hard	–	Clause lengths 3..10 uniformly distributed with Prob. 0.99, 50..100 with Prob. 0.01
	–	55 variables and 1087 clauses
easy	–	Clause lengths 3..10 uniformly distributed with Prob. 0.99, 25..50 with Prob. 0.01

4 Evaluation and results

The closing date of the competition was April, 2nd. 1992. An international field of 36 authors took part. After a simple check, it turned out that at least 26 programs were incorrect due to a wrong syntax (e.g. C++ Code), wrong return values, illegal pointer references etc. Many programs had been developed using MS DOS which doesn't recognize the latter faults. Since we wanted to determine running times of more than ten programs, the authors of the wrong solvers got the chance to correct their code up to May, 30th. We received 25 corrected programs, such that 35 programs took part in the second round.

The programs were compiled on a UNIX Workstation and linked to a test program which provided the ASCII representation of the formulas to the program in question. Again, some of them failed to solve easy test formulas like "(1,2)(-1,2)(1,-2)(-1,-2)". To our surprise, many programs exceeded the estimated running time by orders of magnitudes while trying to solve the hard formulas. Therefore, a second set of much easier formulas were generated to get a ranking of more than six programs. While testing the hard formulas, three programs failed and only six had acceptable running times with respect to the fastest program. The other programs were aborted after 125,000 seconds CPU time. While solving the easy formulas, four programs stopped, due to a segmentation fault and ten entries had running times under 25,000 seconds. The following tables show the running times in detail:

Hard formulas

Participant	Time in seconds
Böhm	19,255
Stamm	23,667
Eisele	40,400
Dörre	43,015
Pretolani	76,464
Purdom	120,961

Easy formulas

Participant	Time in seconds
Böhm	268
Stamm	298
Eisele	436
Dörre	480
Durdanovic	670
Pretolani	680
Purdom	717
Dunker/Bauerfeind	1,539
Burghardt	10,949
Lippold	12,444

Altogether, 14 out of 35 programs were incorrect, even after a correction. Six programs solved the hard formulas in a reasonable amount of time and only ten programs solved the easy set of formulas. Taking into account the results of the programs for the two sets, one gets the following ranking of the top ten entries:

1	Böhm
2	Stamm
3	Eisele
4	Dörre
5	Pretolani
6	Purdom
7	Durdanovic
8	Dunker/Bauerfeind
9	Burghardt
10	Lippold

5 Program descriptions

In this section the authors of the first six entries give insight into their programs.

5.1 SAT solvers "Böhm" and "Stamm"

The SAT solver "B'öhm" is a result of a work in the group of Prof. Speckenmeyer in Düsseldorf. The programs "Böhm" and "Stamm" only differ w.r.t. the heuristics that have been used, the data structures as well as the main program are identical in both implementations.

SAT solver "Böhm" (Max Böhm)

The SAT solver "Böhm" [1] is based on the well known Davis–Putnam–Procedure, which has been enhanced by some heuristics for the selection of variables. The algorithm operates on optimized data structures allowing time efficient access operations.

Algorithm

To solve or to simplify a formula we use the following approach:

1. If F is empty, then return (input formula is satisfiable). If there exists an empty clause, then backtrack.

2. Literals in clauses of length one gets the truth value *true*. (unit–clause–rule).

[1] This research was supported by the state of Nordrhein–Westfalen in the *Forschungsverbund Paralleles Rechnen*

3. Literals appearing positively (negatively) only in the formula, are assigned *true (false)*. (pure–literal–rule)

4. A literal x is chosen using a heuristic approach. x is assigned first by the value *true*, then by the value *false*. The two simplified formulas are solved recursively.

Heuristics

The idea of the heuristic used in step 4 is based on the idea of selecting a literal for assignment occurring as often as possible in the shortest clauses of the formula. Therefore, a shortest clause is either removed or reduced in size by one. Performing this step a few times, clauses of length 1 will result often, hence the formula collapses fast.

In detail, a literal x with maximal vector $(H_1(x), H_2(x), \ldots, H_n(x))$ under the lexicographic order is chosen, where

$$H_i(x) = \alpha\, max(h_i(x), h_i(\bar{x})) + \beta\, min(h_i(x), h_i(\bar{x})),$$

and $h_i(x)$ is the number of clauses of length i, which contain x (lexicographic heuristic). Note that $H_i(x) = H_i(\bar{x})$, therefore x is chosen only if $\sum_i h_i(x) \geq \sum_i h_i(\bar{x})$. In the implementation we have chosen $\alpha = 1$ and $\beta = 2$.

Data Structures

To get time efficient access operations, we have developed an optimized data structure:

- The formula is stored as a list of clauses (ordered by clause length). Direct access to parts of the formula with constant clause length k is supported.
- A clause is represented by a list of its literals.
- For each literal x a list of clauses containing x exists.

This data structure allows to assign a literal x with the value *true* in time $O(\sum_{c \in F: x \in c} |c| + \sum_{c \in F: \bar{x} \in c} 1)$ (with the value *false* in time $O(\sum_{c \in F: x \in c} 1 + \sum_{c \in F: \bar{x} \in c} |c|)$, resp.). It is important to note that the space needed to store the resulting formula is reduced by the same size.

Using suitable pointer techniques in the removed parts of the formula, it is possible to restore the initial formula after many assignments, using reverse operations in reverse order. This needs the same amount of time (!) as previously needed by assigning the literals, only. We mention that this also results in linear space complexity in the size of the actual formula.

Address: M. Böhm, Universität Düsseldorf,
E–mail: boehm@engels1.cs.uni–duesseldorf.de

SAT solver "Stamm" (Hermann Stamm)

The program "Stamm" is based on the program "Böhm" but additionally uses the following 2–SAT procedure (executed between step 2 and step 3 of program "Böhm"):

- The implication graph G is built using the 2–SAT part $F_2 = \{c \in F : |c| = 2\}$ of the formula. This means for each clause $(x, y) \in F_2$ that G contains the edges $\bar{x} \to y$ and $\bar{y} \to x$.
- The strongly connected components (scc) of G are determined (in linear time). If a scc contains both a literal x and its complement $\bar{x}$, the actual formula F will be unsatisfiable, otherwise the literals of each scc are identified.
- The resulting (directed acyclic) graph G' is searched for paths from a literal x to its complement $\bar{x}$. If such a path exists, we have to assign *false* to x, *true* to $\bar{x}$ and all literals reachable from $\bar{x}$ have to be assigned *true*.

Tests have shown that the sizes of the search trees due to the additional 2–SAT procedure is reduced by about 50% compared with the program "Böhm". In contrast to the program "Böhm", however, caused by the 2–SAT procedure, there is no longer a linear relation between the time spent for evaluating a single vertex of the search tree and the amount of size reduction of the formula. For that reason, the overall running time of solver "Stamm" is a bit slower than of solver "Böhm" if the input consists of random formulas (formulas of a special structure are sometimes solved faster).

Address: H. Stamm, Universität Bonn,
E–Mail: hermann@holmium.informatik.uni–bonn.de

5.2 SAT solvers "Eisele" and "Dörre"

Basic version (Andreas Eisele)

The submitted program tries to find a satisfying variable assignment by means of systematic testing and backtracking. For a given formula the following steps are performed as long as possible:

- If the formula contains a unit clause the corresponding variable is instantiated. Therefore other clauses in which this variable appears in the same polarity are already satisfied and can be ignored for further processing. Occurrences of the variable in reverse polarity are removed from the containing clauses. If this leads to new unit clauses, these are treated accordingly.
- The appearance of an empty clause indicates a contradiction. All modifications done starting from the last choice–point are withdrawn and the other branch of this choice is investigated. If no choice–point remains, the formula cannot be satisfied.
- If all occurrences of a variable have the same polarity, an appropriate value can be assigned to the variable without loss of generality. Thus, all clauses with the same variable can be ignored.
- If no open clause remains, a solution has been found.

If none of the above conditions apply for a heuristically chosen variable, a choice point will be introduced and both assignments will be tried. The heuristic method tries to locate variables whose assignment simplifies maximally the remaining formula. We use a weighted number of occurrences, whereas occurrences in two–literal clauses count more than other occurrences. When guessing the value of a variable, the branch with the greatest estimated chance of success will be treated first. For satisfiable formulas, this can — in the average — reduce the fraction of the search space that has to be traversed.

In order to execute this process in an efficient way, we use data structures which facilitate the access of the following information:

- For each clause of the formula: Its original length, a list of literals, its current state (already satisfied or still open) and the number of literals which are still satisfiable.
- For each occurring variable: Lists of clauses with positive and negative occurrences, respectively, the number of positive and negative occurrences in clauses which are still open, and the same numbers restricted to occurrences in binary clauses.

Since modifications to these data structures have to be withdrawn during backtracking, all such modifications are recorded in an undo–stack.

Address: A. Eisele, Universität Karlsruhe,
E–mail: eisele@ira.uka.de

Special treatment of binary clauses (Jochen Dörre)

This modification uses a limited amount of forward checking in order to immediately recognize a situation in which for some variable only one assignment may still lead to a solution and in which this can be found out without introducing new choice points. Only when this check does not reveal such a forced variable assignment, the guessing of a (heuristically good) variable assignment continues.

The following measures are taken in order to reduce the costs of these tests.

1. To check a truth value B for a variable V we only consider the current binary clauses. The set of literals which are forced to be true if $V = B$, is calculated and checked for a pair $\{+L, -L\}$. As soon as such a pair is found, the assignment $V = B$ can be rejected.
2. This check is only performed on a variable which occurs in a clause which has become binary since the last choice point and only if that variable occurs positively as well as negatively in some binary clause. A stack of the variables satisfying this condition is maintained. The variables of this stack are treated (unless they have been bound meanwhile) just before the introduction of a choice point would be necessary.

By this optimization, formulas with exclusively binary clauses (2–SAT) can be determined without any guessing. Also in the case where binary and longer clauses are mixed up, a very frequent case in applications, a much smaller search space has to be considered leading often to a significant improvement of efficiency.

Address: J. Dörre, Universität Stuttgart,
E–mail: jochen@adler.ims.uni–stuttgart.de

5.3 SAT solver "Pretolani"

The B–Reduction Algorithm with Extended 2–SAT Relaxation.
(G. Gallo and D. Pretolani)

Our "B–Reduction Algorithm with Extended 2–SAT Relaxation" (BRR for short) is an enumerative method originally stated in terms of Directed Hypergraphs. In the hypergraph setting, a clause corresponds to a directed hyperarc, while a CNF formula S corresponds to a hypergraph H_S. A B–Reduction H_B of H_S corresponds to a Horn subformula S_B of S, obtained deleting all but one of the positive literals in each non–Horn clause; S is satisfiable if and only if it admits a satisfiable B–Reduction, i.e. a satisfiable subformula S_B. A B–Path P in H_B corresponds to a minimal refutation for S_B; in such a refutation, there exists one and only one last clause, i.e. one containing only negative literals. Our algorithm actually reduces a SAT problem to a (possibly exponentially large) sequence of Horn–SAT problems; we can formally describe BRR as follows:

Algorithm BRR

Step 0	Let S the input formula; set $Q = \{S\}$;
Step 1	If Q is empty, return "no"; otherwise remove a subformula S from Q;
Step 2	Apply Unit Resolution to S; if a contradiction is found, go to Step 1; if S is solved, return "yes";
Step 3	Solve an Extended 2–SAT relaxation; if a contradiction is found, go to Step 1;
Step 4	Select a B–Reduction H_B of the hypergraph H_S corresponding to S; if H_B is satisfiable, return "yes"; otherwise select a B–hyperpath $\Pi \subset H_B$;
Step 5	Let $\{n_1, \ldots, n_k\}$ the negative literals of the negative clause in Π; add to Q the subformulas $S_1, S_2, \ldots, S_k$ generated from S as follows: in formula S_i: n_i is set to false, and n_j, $1 \leq j < i$, are set to true. Go to Step 1.

The Extended 2–SAT relaxation searches a truth assignment satisfying the subset of 2–clauses (clauses of 2 literals) in S with the following restriction: a truth assignment which makes all the literals in a 3–clause false is rejected. Steps 2), 3) and 4) are formulated as path problems on hypergraphs, and implemented as hypergraph algorithms; a linear time implementation of Unit Resolution is used in Step 2).

A first version of this algorithm, in which Step 3) is omitted, is described in: G. Gallo and D. Pretolani: "A New Algorithm for the Propositional Satisfiability Problem", TR–18/92, Dipartimento di Informatica, Universita' di Pisa (to appear on Discrete Applied Mathematics).

Address: D. Pretolani, University Pisa,
E–mail: pretola@di.unipi.it

5.4 SAT solver "Purdom"

(Paul Purdom)

The following algorithm presents the basic ideas behind the entry.

(Preprocessing.) If a variable has more negative than positive occurrence, then reverse the polarity of each occurrence of the variable.

Clause Order Backtracking with Probing for CNF problems.

1. (Empty.) If the CNF problem has an empty clause, then return with no solutions.
2. (Unit clause.) If any clauses have just one variable, set the variable in the way that makes the clause be true.
3. (Pure literal.) If any variable appears only positively or only negatively, set the variable so that its clauses are true.
4. (Probe.) If there are no all–positive clauses or all–negative clauses, then stop all the recursions and report that there is a solution. Otherwise select the type (all–positive or all–negative) of clause that occurs least often.
5. (Select.) Consider variables that occur in those clauses that have the chosen type and that are no longer than any other such clauses. Among these variables, select the variable that occurs most often in clauses of the selected type.
6. (Split.) Generate two subproblems by setting the variable to *true* and to *false*.

I wish to thank Indiana University for the use of its equipment and the support of its staff. I wish to thank Professors Cynthia Brown and John Franco for insightful discussions of satisfiability algorithms.

Address: P. Purdom, Indiana University, E–mail: pwp@zebra.cs.indiana.edu

6 Conclusion

One conclusion of the competition seems to be that at the moment the quickest algorithms for deciding the satisfiability of propositional formulas in conjunctive normal form are Davis–Putnam–based programs. Beside this observation, we are surprised about the difficulties in writing correct C programs.

References

[Coo71] S. A. Cook: The Complexity of Theorem–Proving Procedures, *Proc. 3rd ACM Symp. on Theory of Computing* (1971), pp. 151–158

[DaPu60] M. Davis, H. Putnam: A Computing Procedure for Quantified Theories, *JACM* **7** (1960), pp. 201–215

[GaJo79] M. R. Garey, D. S. Johnson: Computers and Intractability, *Freeman and Company, New York* (1979)