DeepSAT – An Often Polynomial-Time Program for Solving the Satisfiability Problem

Will Naylor

Synopsys, Inc

ABSTRACT

DeepSAT is a program which solves the well-known satisfiability problem.

The algorithm generalizes the traditional conflict-driven clause learning by introducing new variables during the run. The superior performance of DeepSAT results from the greater power of the extended-resolution proof language (used by DeepSAT) vs the resolution proof language (used by traditional solvers).

Discussion includes benchmark data showing polynomial growth of runtime and comparison to traditional solvers, description of the algorithm, and theoretical discussion (including comments on the P ?= NP question).

1 INTRODUCTION

DeepSAT is a program which solves the well-known satisfiability problem.

DeepSAT consumes a satisfiability problem expressed as a product-of-sums[1]. For example

(A|B|C)&(!C|D|E)&(!B|D|G)&....

DeepSAT must either find assignments of TRUE or FALSE to all of the variables to make the entire Boolean expression TRUE, or else DeepSAT must prove that no such assignment exists. Note that for the entire expression to be TRUE, each clause must be TRUE.

DeepSAT attempts to construct a refutation (ie, proof that no variable assignment makes all clauses TRUE). The refutation is in the formal language called "extended resolution". Extended resolution derives a new clause from two existing clauses by a resolution step For example, C and !C appear in the clauses

(A|B|C)

and

(!C|D|E)

If C == TRUE, then !C must be FALSE, so D|E must be TRUE. Alternatively, if C == FALSE, then A|B must be TRUE. Thus

(A|B|D|E)

must be TRUE. So we derive

(A|B|D|E)

from the 2 clauses (A|B|C) and (!C|D|E). That is

(A|B|C)&(!C|D|E) --> (A|B|D|E) (equation 2)

This is an example resolution step.

[1] "|" means Boolean OR, "&" means Boolean AND, "!" means Boolean NOT. !FALSE = TRUE, !TRUE = FALSE. FALSE|FALSE = FALSE, all other combinations TRUE. TRUE&TRUE = TRUE, all other combinations FALSE.

Conflict-driven-clause-learning is a method of proving new clauses by several resolution steps (see section 3.7 for details).

A sequence a resolution steps that derives the empty clause is a refutation.

For example, consider the satisfiability problem

(A|B)&(B|C)&(A|C)&(!A|!B)&(!A|!B)&(!A|!C)

Below is a resolution refutation. It is in the high school geometry proof format, with each line consisting of <label><statement><reason>.

```
/* definition of problem */
(1)  A | B          - axiom;
(2)  B | C          - axiom;
(3)  A | C          - axiom;
(4)  !A | !B        - axiom;
(5)  !B | !C        - axiom;
(6)  !A | !C        - axiom;

/* proof */
(7)  B | !C         - resolve (1), (6);
(8)  B              - resolve (2), (7);
(9)  !B | C         - resolve (3), (4);
(10) !B             - resolve (5), (9);
(11) <CONTRADICTION> - resolve (8), (10);
```

Extended resolution also allows definition of new variables in terms of existing variables, in order to aid in constructing a resolution refutation. DeepSAT defines new variables as OR of 2 existing variables or their inverses. For example

define NEWVAR1 = B|!D

DeepSAT defines the new variable by entering 3 new clauses into its clause database [tseitin1970], which for this example are

(NEWVAR1|!B)&(NEWVAR1|D)&(!NEW_VAR|B|!D) (equation 3)

Many satisfiability problem classes require exponential length refutations in resolution [urquahart1987] [chvatal1988] [urquahart1995], whereas this author knows of no fully proved result showing that any satisfiability problem class requires an exponential length refutation in extended resolution [urquahart1995]. The length of the shortest possible refutation is important, because an algorithm which solves satisfiability problem must write a refutation, either implicitly or explicitly, in order to be correct for unsatisfiable problems. If the length of the shortest refutation is exponential, then just writing the refutation requires exponential time, without even counting the time to find the refutation, so we can have no hope of a sub-exponential runtime algorithm. Problem classes known to require exponential length refutations in resolution include pigeonhole principle, xor reordering equivalence, multiplier equivalence [urquahart1987] [urquahart1995]. But these same problem classes have polynomial-sized refutations in extended resolution.

We present a theoretical argument that the length of a refutation produced by DeepSAT is polynomial in the length of the shortest possible refutation. Since DeepSAT uses polynomial time to produce each step in the refutation, DeepSAT runs in time polynomial in the length of the shortest refutation.

Benchmark data shows that performance of DeepSAT is superior to traditional solvers on the hardest problems[2], and that growth of runtime and proof size is polynomial, as predicted by the theory.

Finally, we discuss the relationship with the P ?= NP question. In particular, (if our theoretical argument is correct), DeepSAT solves SAT in polynomial time if the shortest possible refutation is of polynomial length. Thus co-NP = NP implies that DeepSAT solves all SAT problems in polynomial time.

[2] By "hardest", we mean problems where the shortest resolution refutation is of exponential length in the problem size.

2 BENCHMARK DATA

2.1 Hard Testcases from SATRACE SAT Contest on satlive.org

Table 10 shows comparison between DeepSAT and MINISAT2 on these hard public domain testcases. (MINISAT2 is a popular modern SAT solver). The testcases were selected because they give traditional SAT solvers trouble, and they seemed to have short refutations in proof languages other than resolution.

Table 10: DeepSAT and MINISAT2 on SELECTED SATRACE PROBLEMS

			solver ----------				
testcase ----------			minisat2 ----------		DeepSAT ----------		
name	# vars	# terms	conflicts	run time (sec)	conflicts	run time (sec)	# vars
**** hard cases from satlive.org:							
gripper14u.shuffled-as.sat03-397	4355	40382	88421710	10126.4	8.21E+06	9733.299	461
bevhcube6.shuffled-as.sat03-1428	576	1536	killed after 5 days		5.38E+06	1483.859	2591
marg6x6.shuffled-as.sat03-1456	156	800	killed after 5 days		3.27E+06	1006.067	1849
urqh1c6x6.shuffled-as.sat03-1469	191	1984	killed after 5 days		1.11E+07	3642.59	3110
urquhart4_25bis.shuffled-as.sat03-1554	192	512	killed after 5 days		6.25E+05	78.542	659
pyhala-braun-unsat-40-4-04.shuffled-as.sat03-1548	9638	31216	5738857	866.727	9.21E+05	2775.019	241
Urquhart-s5-b3.shuffled-as.sat03-1572	121	1116	killed after 5 days		3.22E+06	1026.674	1913
am_6_6.shuffled-as.sat03-362	2269	7809	64900221	2987.85	5.47E+06	6124.852	1049
hcb4.shuffled-as.sat03-1432	112	2048	killed after 5 days		1.32E+07	9533.689	3445
hypercube6.shuffled-as.sat03-1436	192	2048	killed after 5 days		4.26E+06	1167.125	2192

2.2 Pigeonhole principle

Satisfiability problem which asserts that P pegs must be placed in P-1 holes.

Table 11 shows DeepSAT runs on problems of size 4 to 20. For comparison, we also ran on the popular SAT solver MINISAT2. Log-log plots show runtime vs problem size. In addition, we show log-log plot of problem size vs runtime for MINISAT2, which shows up as a curve rather than a line, because of the exponential growth of runtime of MINISAT2 (because pigeonhole problem has shortest resolution refutation size which grows exponentially [urquahart1987]).

The linear shape of the DeepSAT log-log plot implies polynomial growth rate. To see this, observe that if

runtime = K*problem_size^E

then taking log of both sides yields

log(runtime) = E*log(problem_size) + log(K)

Thus, the slope of the line of the log-log plot is the exponent of the polynomial growth.

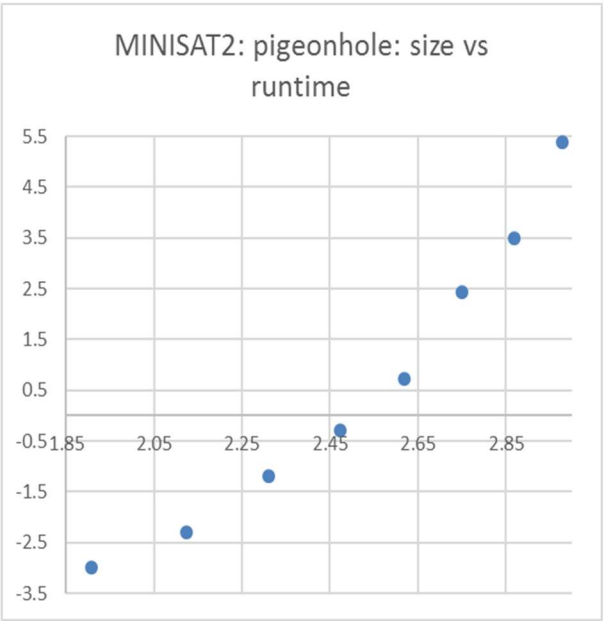
The regression fit on the log-log data shows

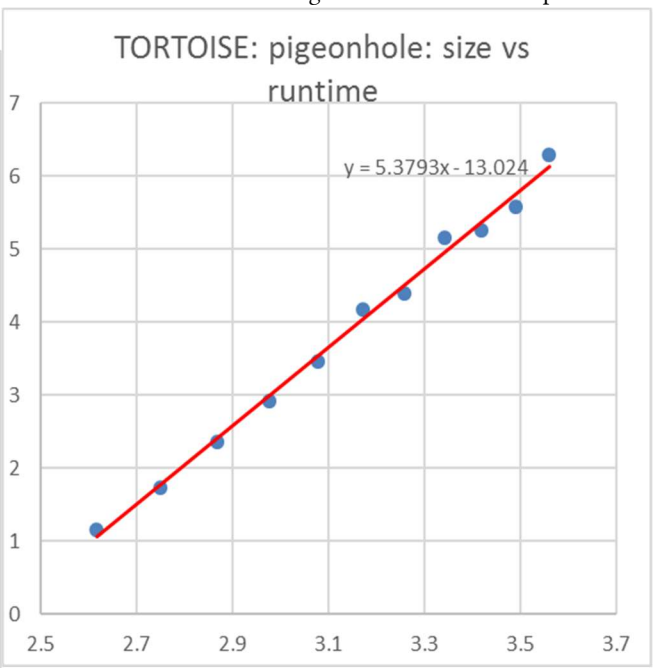
DeepSAT runtime = O(problem_size^5.4)

Table 11: DeepSAT and MINISAT2 on PIGEONHOLE PROBLEM INSTANCES

			solver ----------				
testcase ----------			minisat2 ----------		DeepSAT ----------		
name	# vars	# terms	conflicts	run time (sec)	conflicts	run time (sec)	# vars
holemin.4.3	12	22	0	0	6	0.011	0
holemin.5.4	20	45	32	0	31	0.011	0
holemin.6.5	30	81	251	0.001	156	0.036	0
holemin.7.6	42	133	1206	0.005	1142	0.575	1
holemin.8.7	56	204	10495	0.063	10150	2.989	10
holemin.9.8	72	297	76178	0.52	324962	12.987	117
holemin.10.9	90	415	551941	5.32	337481	14.2	155
holemin.11.10	110	561	16769952	275.264	717183	54.82	210
holemin.12.11	132	738	1.09E+08	3092.963	1.04E+06	230.343	298
holemin.13.12	156	949	8.89E+09	241056.8	3.10E+06	838.772	675
holemin.14.13	182	1197	killed after 5 days		7.61E+06	2852.959	832
holemin.15.14	210	1485	killed after 5 days		1.38E+07	14788.07	1144
holemin.16.15	240	1816	killed after 5 days		2.80E+07	24844.51	1981
holemin.17.16	272	2193	killed after 5 days		5.59E+07	143035.4	1999
holemin.18.17	306	2619	killed after 5 days		1.30E+08	181405.9	4352
holemin.19.18	342	3097	killed after 5 days		1.99E+08	387869.5	4322
holemin.20.19	380	3630	killed after 5 days		3.72E+08	1924853	5957

EDIT NOTE: change “TORTOISE” to “DeepSAT” below





2.3 XOR reordering

Satisfiability problem which asserts that N variables exclusive-or-ed together in 2 different random orders have different results for some input.

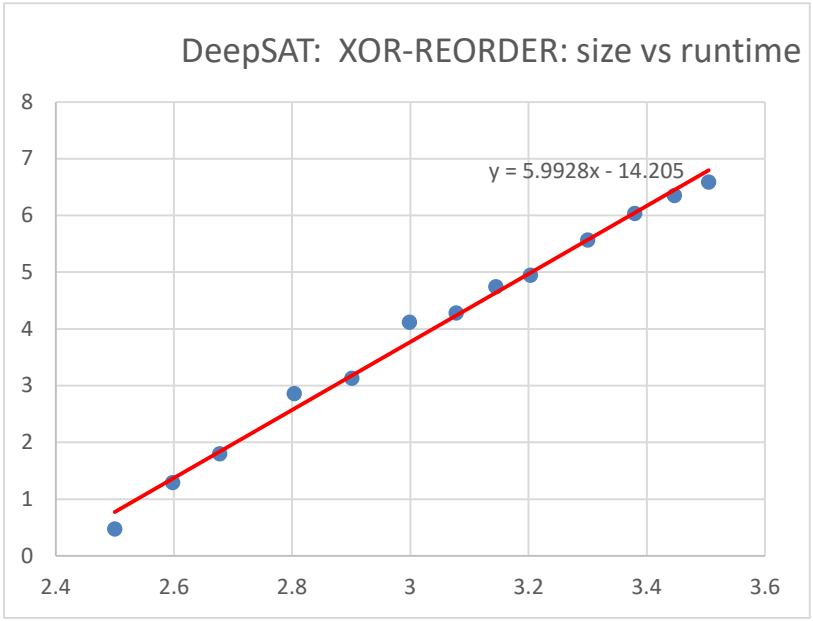
Table 12 shows DeepSAT runs on problems of size 40 to 400 variables. For comparison, we also ran on the popular SAT solver MINISAT2. The DeepSAT plot is as above.

The regression fit shows

DeepSAT runtime = O(problem_size^6.0)

Table 12: DeepSAT and MINISAT2 on XOR REORDER PROBLEM INSTANCES

			solver ----------				
testcase ----------			minisat2 ----------		DeepSAT ----------		
name	# vars	# terms	conflicts	run time (sec)	conflicts	run time (sec)	# vars
xor.40.0	121	316	17381527	75	45021	3.033	162
xor.50.0	151	396	1.01E+09	6787	193303	19.903	273
xor.60.0	181	476	1.95E+10	149245	420177	63.806	529
xor.80.0	241	636	killed after 5 days		2.76E+06	731.192	1487
xor.100.0	301	796	killed after 5 days		4.07E+06	1358.264	2048
xor.125.0	376	996	killed after 5 days		1.93E+07	13307.93	3325
xor.150.0	451	1196	killed after 5 days		2.63E+07	19201.4	4247
xor.175.0	524	1396	killed after 5 days		6.09E+07	56159.71	6218
xor.200.0	599	1596	killed after 5 days		9.03E+07	89341.69	7585
xor.250.0	749	1996	killed after 5 days		2.67E+08	374262.8	12785
xor.300.0	899	2396	killed after 5 days		5.43E+08	1101667	17984
xor.350.0	1051	2796	killed after 5 days		9.56E+08	2280038	23032
xor.400.0	1199	3196	killed after 5 days		1.36E+09	3908370	28953



2.4 Multiplier equivalence/correctness

Satisfiability problem which asserts that two different randomly generated multipliers get different results for some input. To prevent the simple enumeration refutation, the product plane is used directly as input, with no input bits feeding the product plane with bit AND logic. To make the problem hard, partial sums and carry bits at each level are scrambled by a randomly generated 3-tree of full adders.

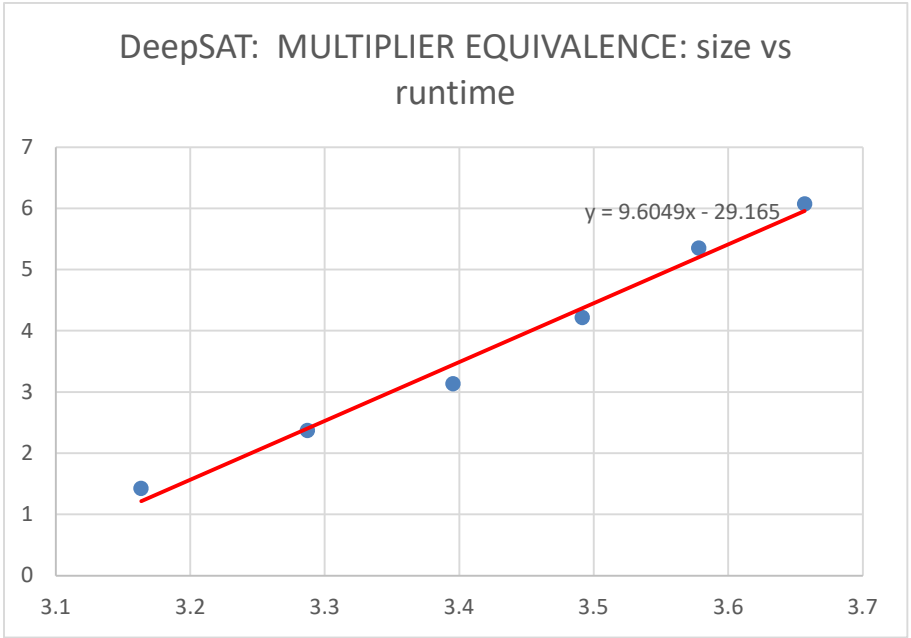
Table 13 shows DeepSAT runs on problems of size 2 to 12. For comparison, we also ran on the popular SAT solver MINISAT2. The DeepSAT plot is as above.

The regression fit shows

DeepSAT runtime = O(problem_size^9.6)

Table 13: DeepSAT and MINISAT2 on RANDOMIZED MULTIPLIER EQUIVALENCE PROBLEM INSTANCES

			solver ----------				
testcase ----------			minisat2 ----------		DeepSAT ----------		
name	# vars	# terms	conflicts	run time (sec)	conflicts	run time (sec)	# vars
rmult_noin.0.2	41	77	12	0	11	0.008	0
rmult_noin.0.3	94	217	135	0.001999	146	0.012	0
rmult_noin.0.4	169	425	3362	0.017997	788	0.042	0
rmult_noin.0.5	266	701	20048	0.124981	5705	0.172	2
rmult_noin.0.6	385	1045	132483	0.970852	30598	2.081	4
rmult_noin.0.7	526	1457	2353389	22.2006	162819	26.791	56
rmult_noin.0.8	689	1937	88293384	1174.84	862561	238.81	306
rmult_noin.0.9	874	2485	1.35E+08	2193.17	4.37E+06	1387.344	695
rmult_noin.0.10	1081	3101	4.44E+09	120118	**1.89E+07**	16646.99	1896
rmult_noin.0.11	1310	3785	killed after 10 days		8.33E+07	225829.6	6030
rmult_noin.0.12	1561	4537	killed after 10 days		2.07E+08	1199233	11969



3 DeepSAT's ALGORITHM

DeepSAT uses conflict driven clause learning in a manner similar to Chaff [chaff2001] to prove new clauses (section 3.7 below is a brief discussion of conflict-driven clause learning). Conflict-driven clause learning is an effective method to search for a resolution refutation and a satisfying variable assignment simultaneously. DeepSAT is similar to Chaff in using variable activity to choose which variable to branch, and using 2-watched-literal algorithm for implication propagation. Unlike Chaff, DeepSAT chooses variable phase randomly rather than by activity. This difference seems to be very important for generating new variables effectively.

In addition to conflict-driven clause learning, DeepSAT identifies length-1 and length-2 clauses to prove by searching cycles in length-2 and length-3 clauses. Section 3.8 below describes details.

After running conflict-driven clause learning for number of steps of order the number of literals in the clause database, DeepSAT stops to do several O(N) runtime operations. These are described below. After running these O(N) operations, DeepSAT resumes conflict-driven-clause-learning.

3.1 NEW VARIABLE HANDLING

The most important of these O(N) operations is handling of new variables. DeepSAT defines a new variable as the OR of 2 previously existing variables, for example

NEW_VARIABLE = A | !C

DeepSAT defines the new variable by entering 3 new clauses into its clause database [tseitin1970], which for this example are

(NEWVAR1|C)&(NEWVAR1|!A)&(!NEW_VAR|A|!C)

First DeepSAT examines all clauses to see if any clauses contain pairs of literals defining new variables. When found, the new variable is substituted. For example, if the clause database contains

(A | !B | !C)

DeepSAT substitutes in the new variable, yielding

(NEW_VARIABLE | !B)

Then DeepSAT considers defining new variables. If during the preceding conflict-driven-clause learning run, no length-2, literal assertions, or literal equalities were proved, then DeepSAT defines a new variable. To decide which literals to OR together to define the new variable, DeepSAT queries its clause database to find which pairs of literals are most common. This is the maximum compression heuristic. After defining the new variable, DeepSAT substitutes the new variable into all possible clauses.

The goal of new variable definition is to keep the clauses compressed so that conflict-driven-learning proves at least one length-2, literal assertion, or literal equality during the conflict-driven-clause-learning pass. If defining 1 new variable is insufficient to keep DeepSAT proving these, DeepSAT defines 2 new variables in the next pass. If this is still insufficient, DeepSAT defines 4 next time, etc.

3.2 EQUALITY COLLAPSING

Equal literals confuse DeepSAT's maximum compression heuristic by distorting the literal pair count used to choose the literals for the new variables. So it is important to find and remove equal literals. DeepSAT uses an O(N) runtime depth-first search algorithm to find all equalities implied by length-2 clause loops, and then substitute out equal literals from all clauses.

For example, the length-2 clause loop

(A | B) & (!B | C) & (!C | !A)

implies

A == !B == !C

3.3 CLAUSE SHORTENING

Clauses that are longer than they need to be confuse DeepSAT's maximum compression heuristic for new variable generation, because these clauses introduce literal pairs into the literal pair count that should not be in the count. The traditional conflict-driven-clause-learning method often generates clauses that have unneeded literals. Also, as the clause database grows and its implicativity strengthens, many clauses contain literals which are redundant with respect to the entire clause database, and thus these literals can be removed and thus the clauses shortened. The sections below describe how DeepSAT handles these two issues.

3.3.1 CONFLICT-DRIVEN-CLAUSE-LEARNING CLAUSE SHORTENING

When conflict-driven-clause-learning generates a new clause, DeepSAT follows the implication DAG that leads to each literal of the new clause. If for a literal, the implication DAG leading to that literal has a cut composed entirely of other literals of the new clause, then the literal is deleted from the new clause.

3.3.2 GENERAL CLAUSE SHORTENING

Periodically, DeepSAT randomly selects clauses in its clause database to be shortened. The clause to be shortened is temporarily removed from the clause database. DeepSAT then sets variables to falsify literals of the clause. The variables are selected in random order. If implication propagation (BCP) forces any literal of the clause to be TRUE or FALSE, then the clause is shortened. Details are shown in the examples below.

For example, suppose the database contains the clause

(A | B | C | D | E)

Suppose DeepSAT sets

A = FALSE
B = FALSE
C = FALSE

and the implication propagation through the clause database yields

D = TRUE

then literal E can be deleted from the clause.

Then, if we backtrace through the implication propagation DAG, and we discover that

B = FALSE

is not necessary to derive

D = TRUE

then we delete literal B from the clause as well.

On the other hand, if setting

A = FALSE
B = FALSE
C = FALSE

yields

D = FALSE

by implication propagation, then we delete D from the clause.

Finally, if setting

A = FALSE
B = FALSE
C = FALSE

in that order yields a conflict, then we delete variables D and E from the clause.

Then, if we backtrace through the implication propagation DAG, and we discover that

B = FALSE

is not necessary to derive the conflict, we delete literal B from the clause as well.

If a conflict is found during implication propagation, then a conflict clause is added to the clause database.

3.4 CLAUSE DELETION

The presence of redundant clauses bloats memory usage and increases runtime, but most importantly, distorts the literal pair count, which confuses the max compression heuristic for generating new variables. So it is important to delete redundant clauses as rapidly as possible. Of course DeepSAT never deletes clauses from the original problem definition, nor clauses needed to define new variables.

DeepSAT deletes clauses in two different ways:
1) DeepSAT attempts to prove length-2, 3, and 4 clauses redundant, and deletes the clause if it is redundant.
2) DeepSAT keeps activity counts on length-4 and above clauses, and deletes the clauses if their activity count is too small.

Details are below.

3.4.1 CLAUSE DELETION of LENGTH-2, LENGTH-3, and LENGTH-4 CLAUSES

Periodically, DeepSAT tests all length-4 clauses and smaller for redundancy. DeepSAT starts with longest clauses first, and proceeds in random order for clauses of the same length.

The following example shows how DeepSAT tests for clause redundancy.

Assume the clause

(A | B | C)

is in DeepSAT's clause database.

DeepSAT temporarily removes the clause from its database, then sets variables from the clause to test implicativity of the clause database minus the tested clause.

If setting

A = FALSE
B = FALSE

yields

C = TRUE

and

A = FALSE and C = FALSE implies B = TRUE

and

B = FALSE and C = FALSE implies A = TRUE

then the clause is redundant and DeepSAT deletes it.

DeepSAT does similar tests for length-2 and length-4 clauses.

While doing these tests for clause deletion, DeepSAT records information which sometimes implies equalities or proved literal values. For example, if during testing described above, DeepSAT notices that

A = FALSE implies B = FALSE

and

A = TRUE implies B = TRUE

then it follows that A = B.

On the other hand, if

A = FALSE implies B = FALSE

and

A = TRUE implies B = FALSE

then it follows that B = FALSE.

3.4.2 CLAUSE DELETION of LENGTH-4 and LONGER CLAUSES

DeepSAT deletes enough length-4 and longer clauses to keep the runtime spent doing implication propagation through length-4 and longer clauses no more than the runtime spent doing implication propagation through length-3 and smaller clauses. The metric DeepSAT uses to decide which clauses to delete is

value_of_clause = 1/(time_since_last_progation_through_clause*clause_length)

DeepSAT deletes clauses with the smallest value_of_clause first.

3.5 RESTARTS

A "restart" is an interruption of conflict-driven clause learning where all variable settings get backtracked, and then variable setting resumes from scratch.

DeepSAT has very frequent restarts. DeepSAT measures the runtime from the last restart until the first conflict. DeepSAT does another restart when the runtime exceeds 10X the runtime until the first conflict.

The activity count used to select variables to branch is retained during restarts. After runtime of approximately 500 times number of literals, the activity count is reset to 0. (Runtime here is number of clause visits during implication propagation). These resets are to prevent DeepSAT from getting stuck on one part of the problem.

3.6 COMMENTS on LENGTH-2 CLAUSES

Length-2 clauses play a special role in DeepSAT. See the theory section below (section 4).

Random selection of variable phase during branching, random ordering of variable branching during clause shortening give a high probability of proving most of the length-2 clauses that can be proved directly by resolution from other length-2 and length-3 clauses of high or moderate activity, with no longer clauses proven as intermediate steps.

As discussed above, DeepSAT defines new variables only if it is unable to prove any length-1, length-2, or length-3 clauses during a pass of conflict-driven-clause-learning. It is interesting that when DeepSAT runs on problems that traditional solvers can solve, DeepSAT usually does not need to introduce any new variables to solve the problem. If DeepSAT does not need to introduce any new variables, then DeepSAT provably runs in polynomial time, because it must terminate after proving O(N^2) length-2 or shorter clauses.

Introducing new variables more often than described above led to DeepSAT getting stuck on some SAT problems. The difficulty seems to be that DeepSAT proves many clauses relating the new variable definitions to each other, but not enough regarding the underlying SAT problem. Waiting until DeepSAT has proved all length-1, length-2 clauses possible before introducing new variables forces DeepSAT to prove length-2 and shorter clauses regarding the underlying SAT problem if it is possible to do so.

3.7 CONFLICT-DRIVEN CLAUSE LEARNING

Conflict-driven clause learning is the main algorithm used by modern satisfiability solvers such as CHAFF [chaff2001] and MINISAT2. Conflict-driven-clause learning is an effective method to search for a resolution refutation and a satisfying variable assignment simultaneously.

Conflict-driven clause learning works by repeatedly setting values to variables and then propagating the implications of these settings. DeepSAT chooses the variable with the highest activity count to branch first (see details on activity count below). DeepSAT decides whether to set the variable TRUE or FALSE randomly. Then the implications of the variable setting are propagated (called "binary-constraint-propagation" or "BCP" in the literature). After all implications are propagated, one of the following happens:

1) All clauses are satisfied (that is, all clauses are TRUE). DeepSAT terminates and outputs the satisfying variable assignment it has found.
2) Implication propagation leads to an inconsistency in implied setting for a variable (called a "conflict" in the literature). Analysis of the implication DAG implies a new clause, which is added to the clause database, and will prevent this conflict from ever occurring again (the clause is called a "conflict clause" in the literature). Then DeepSAT backtracks the variable settings until the conflict clause is no longer falsified and the conflict clause no longer implies variable setting.

   And then DeepSAT resumes setting variables.
3) Neither of the above. In this case, DeepSAT continues assigning values to variables.

As conflict clauses accumulate, fewer and fewer variables need to be set to lead to a conflict. If a satisfying assignment is not eventually found, then we reach the condition that setting a single variable alone leads to a conflict, and then setting the same variable to the opposite value also leads to a conflict. When this happens, DeepSAT terminates and reports that the problem is unsatisfiable.

Below are simple examples to show how conflict-driven clause learning works.

3.7.1 EXAMPLE of CONDITION #1 ABOVE

Consider the following three clauses:

(A | B | C) & (!C | D | E) & (A | !E | F)

Assume we first branch

B = FALSE

Nothing happens, so we keep branching. Assume we next branch

D = FALSE

Again, nothing happens, so we keep branching. Assume we next branch

A = FALSE

Since A = FALSE and B = FALSE, clause (A | B | C) implies

C = TRUE

Since C = TRUE, !C = FALSE, and we previously branched D = FALSE, so clause (!C | D | E) implies

E = TRUE

Since E = TRUE, !E = FALSE, and we previously branched A = FALSE, so clause (A | !E | F) implies

F = TRUE

Now all clauses are true, so DeepSAT terminates and reports the satisfying variable assignment

A = FALSE
B = FALSE
C = TRUE
D = FALSE
E = TRUE
F = FALSE

3.7.2 EXAMPLE of CONDITION #2 ABOVE

Again consider these three clauses:

(A | B | C) & (!C | D | E) & (A | !E | F)

Now we consider a different variable setting pattern. Assume previously we have branched

B = FALSE
D = FALSE
F = FALSE

Now assume we branch

A = FALSE

Since A = FALSE and B = FALSE, clause (A | B | C) implies

C = TRUE

Since C = TRUE, !C = FALSE, and we previously branched D = FALSE, so clause (!C | D | E) implies

E = TRUE

But since A = FALSE and F = FALSE, clause (A | !E | F) implies !E = TRUE, which implies

E = FALSE

We now have an implication conflict, because one implication path proves E = FALSE, and another proves E = TRUE. Backtracing from the conflict, we discover that once

B = FALSE, D = FALSE, F = FALSE

Setting

A = FALSE

inevitably leads to the conflict by implication propagation. Thus if

B = FALSE, D = FALSE, F = FALSE

we must have

A = TRUE

The clause which directly enforces this by implication propagation is

(A | B | D | F)

so we add this new clause to the clause database, which ensures that this conflict can never occur again.

To add this clause to the database but not have it be false, we must backtrack setting A. To have the clause not immediately cause an implication, we must also backtrack setting F.

All variables involved in the conflict have their activity count incremented. Variables with the highest activity count get branched first. Older activity counts less than recent activity.

Note that clause (A | B | D | F) can be derived from the other clauses by resolution.

3.8 LOOP RESOLUTION

DeepSAT searches for cycles in the clause structure of length-2 and length-3 clauses which imply length-1 and length-2 clauses. If a length-1 clause gets proved, its implications are propagated by BCP. If a length-2 clause get proved, DeepSAT checks the new clause for redundancy against length-2 and length-3 clauses already in the database. The redundancy check is the same as described in section 3.4.1 for deletion of length-2, length-3, and length-4 clauses.

3.8.1 CLAUSE LOOP IMPLYING LENGTH-1 CLAUSE

A cycle of length-2 clauses with inverted literals between pairs of clauses adjacent in the cycle except one pair which is same phase implies a length-1 clauses.

For example

(A | B) & (!B | C) & (!C | A)

implies

A = TRUE

3.8.2 CLAUSE LOOP IMPLYING LENGTH-2 CLAUSE

A cycle of clauses as above but with one of the clauses being length-3 implies a length-2 clause.

For example

(A | B) & (!B | C | D) & (!C | A)

implies

(A | D)

3.8.3 SEARCH ALGORITHM

First loops implying length-1 clauses are searched. When no more length-1 loops remain, then loops implying length-2 clauses are searched. When no loops implying length-1 or length-2 clauses remain, the algorithm reports the search is complete.

DeepSAT keeps length-2 and length-3 clauses in a priority queue. Clauses get popped off the queue to be expanded. Expanding a clause means that DeepSAT finds all loops which can prove clauses using the popped clause. Expanding searches for either length-1 clause loops, or length-2 or shorter clause loops. The priority queue stores clauses sorted by type of search. Clauses due for length-1 search pop first. After clause is expanded for length-1, the clause if re-inserted into the queue for length-2 search. After clause is expanded for length-2, clause is removed from the queue.

When the queue becomes empty, the algorithm reports that the search is complete.

Initially, all clauses go into the queue labelled for the smallest loop that can be searched from the clauses. Length-2 clause can expand for length-1 loops; length-3 clause can expand for length-2 loops. When a new clause is proved, either by loop resolution, or by other means, the clause is inserted into the queue in the same manner.

4 THEORETICAL ANALYSIS: WHY DeepSAT PRODUCES POLYNOMIAL-LENGTH REFUTATIONS

First in section 4.1 we give a brief overview of the mathematics of DeepSAT. Then in section 4.2 we prove that DeepSAT's loop resolution inference engine (see section 3.8) produces a refutation whose length is worst case polynomial in the length of the shortest possible refutation, if DeepSAT is given a set of new variable definitions that are carefully chosen. Then we argue that DeepSAT's heuristic for generating new variable definitions is sufficiently good to produce polynomial length refutations, and in section 5 we give empirical evidence which supports the theory.

4.1 OVERVIEW of DeepSAT MATHEMATICS

The loop resolution inference engine of DeepSAT uses resolution to derive all possible length-2 and smaller clauses from length-3 and smaller clauses already in DeepSAT's database. When all possible derivations are completed, DeepSAT defines new variables in terms of existing variables, using a heuristic designed to achieve maximum average compression of clause length. Then DeepSAT substitutes the new variables into all clauses to reduce the clause lengths, generating new length-3 and smaller clauses.

The process repeats until either a refutation is found or a satisfying variable assignment is found.

4.2 WHY DeepSAT PRODUCES POLYNOMIAL-LENGTH REFUTATIONS

First in section 4.2.1 we prove that any extended resolution proof can be re-written to use only length-3 clauses and smaller, by defining new variables and substituting into the clauses of the proof to shorten the clauses. Then in section 4.2.2 we prove that DeepSAT's length-2 loop resolution inference engine can generate a refutation of polynomial length if given the new variables defined for the re-written proof. Then we prove that very much lower-quality set of new variable definitions is sufficient to enable DeepSAT to find a polynomial-length refutation. Finally, we claim that it is easy to generate such a lower-quality set of new variable definitions.

4.2.1 COMPRESSING CLAUSES IN THE SHORTEST POSSIBLE REFUTATION

Let

proof

be an extended resolution refutation of a satisfiability problem.

Define

proof_length(proof)

as the number of literals in "proof".

Define

shortest_proof

as the proof with the smallest possible "proof_length".

Define

shortest_proof_length

as proof_length(shortest_proof).

THEOREM 1:
It is always possible to transform "shortest_proof" into another refutation which has these properties:
1) The new refutation uses only length-3 and smaller clauses
2) The new refutation proves only length-2 and smaller clauses using loop resolution
3) The new refutation requires the definition of no more than shortest_proof_length new variables.

PROOF:
See appendix 9 for rigorous proof of THEOREM 1. The proof is similar to the reduction of general satisfiability to 3-SAT given in [Garey&Johnson].
END_OF_PROOF

The transformation of shortest_proof is accomplished by defining new variables which, when substituted into the clauses of shortest_proof, shorten all of the clauses of shortest_proof into length-2 and smaller clauses. A few new steps in the transformed proof are needed, as detailed in appendix 9.

4.2.2 DeepSAT's INFERENCE ENGINE GENERATES POLYNOMIAL-LENGTH PROOFS

In this section, we argue that DeepSAT always produces refutations with

proof_length <= O(shortest_proof_length^(2*P)) (equation 4)

where P >= 1, and the number of new variables DeepSAT defines is

number_of_new_variables <= O(shortest_proof_length^P) (equation 5)

First we prove that DeepSAT can find a polynomial-length refutation if given a short "hint" consisting of a special set of new variable definitions. Then we prove that very much lower-quality set of new variable definitions is sufficient to enable DeepSAT to find a polynomial-length refutation. Finally, we claim that it is easy to generate such a lower-quality set of new variable definitions.

Define

num_vars

as the number of variables in DeepSAT's clause database.

LEMMA 2:
If a resolution refutation using only length-3 and smaller clauses (and proving by loop resolution only length-2 and smaller clauses) is possible, DeepSAT's loop resolution inference engine (length-2 engine) will
derive a refutation with

proof_length <= 2*(4*num_vars^2) + 1*(num_vars)

(equation 6)

PROOF:
Only 4*num_vars^2 length-2
clauses are possible; only num_vars length-1 clauses are possible,
so the proof_length bound follows.

DeepSAT derives all possible implications provable by resolution involving
length-3 and smaller clauses, and by assumption, a refutation is possible,
so DeepSAT will derive a refutation.
END_OF_PROOF

Usually DeepSAT finds refutations with proof_length much smaller than O(num_vars^2).

LEMMA 3:
If DeepSAT's loop resolution inference engine (length-2 engine) is given:
1) new variable definitions from shortest_proof
AND
2) new variable definitions from THEOREM 1
AND
3) original satisfiability problem, but with all clauses except variable definition clauses shortened
to length-2 and smaller, as detailed in THEOREM 1
then DeepSAT's main inference engine will always derive a refutation
with

proof_length <= O(shortest_proof_length^2) (equation 7)

PROOF:
From THEOREM 1, we know that a refutation using only length-3 and smaller
clauses is possible.

After new variable definitions from THEOREM 1

num_vars = O(shortest_proof_length) (equation 8)

Substituting into result from LEMMA 2 gives us

proof_length <= O(shortest_proof_length^2) (equation 9)

END_OF_PROOF

Usually we see much slower growth rates, as discussed earlier.

Thus we see that DeepSAT can find a polynomial-length refutation if given a short "hint" consisting of the correct new variable definitions.

We now prove that very much lower-quality set of new variable definitions is sufficient to enable DeepSAT to find a polynomial-length refutation.

DEFINITION 4:
We say a set S of new variable definitions "compress and contain"
an extended resolution refutation R if all of the new variable
definitions in the extended resolution refutation R are contained
in S and the new variable definitions in S are sufficient to
compress R to a refutation which proves only length-2 and smaller
clauses.

LEMMA 5:

Let K be a number => 1 and let P be a number >= 1.

Let DeepSAT's loop resolution inference engine (length-2 engine) be given

num_new_variables = K*shortest_proof_length^P (equation 10)

new variable definitions. If ANY subset of
the new variable definitions is sufficient to compress and contain
ANY extended resolution refutation, then DeepSAT's
length-2 loop resolution inference engine will always derive a refutation
with

proof_length <= O(shortest_proof_length^(2*P)) (equation 11)

PROOF:
By assumption, we know that a refutation proving only length-2 and smaller
clauses is possible.

After new variable definitions

num_vars = O(shortest_proof_length^P) (equation 12)

Substituting into result from LEMMA 2 gives us

proof_length <= O(shortest_proof_length^(2*P)) (equation 13)

END_OF_PROOF

CONJECTURE 7:
DeepSAT's heuristic for generating new variable definitions
is sufficiently good to always satisfy LEMMA 5 for some constant
K and some power P.

ARGUMENT:
LEMMA 5 is actually quite a weak condition. Most of the new variables
defined can be useless, and ANY subset can compress and contain
ANY extended resolution proof, even one very much longer than the shortest
possible. It is likely that there are many many extended resolution
proofs that are polynomially longer than the shortest possible. So having
a large set of new variables defined according to a good heuristic, it
seems likely that some subset will compress and contain at least one
of the many extended resolution proofs.

The strongest argument is that our empirical data seems consistent
with CONJECTURE 7, and we have so far been unable to find any counterexample.
END_OF_ARGUMENT

CONJECTURE 8:
Extended resolution can simulate any refutation proof in any proof language
regarding the satisfiability problem, with at most polynomial growth in
proof length.

See [urquahart1995] for discussion of CONJECTURE 8.

In the abstract and elsewhere in this paper, we make assertions about "all proofs" and "any proof in any language". If CONJECTURE 8 is false, we need to weaken the statements to "proofs in languages that extended resolution can polynomially simulate".

If CONJECTURE 8 is true, then any statements about polynomial behavior of extended resolution refutations applies to ALL refutations in ALL languages.

LEMMA 9:
DeepSAT produces refutations with

proof_length <= O(num_new_variables^R) (equation 14)

for some R >= 1.

PROOF:
The result follows from equation 10, equation 11, and
the details of the DeepSAT algorithm from section 3.

END_OF_PROOF

5 FURTHER ANALYSIS of EMPIRICAL EVIDENCE of SECTION 2

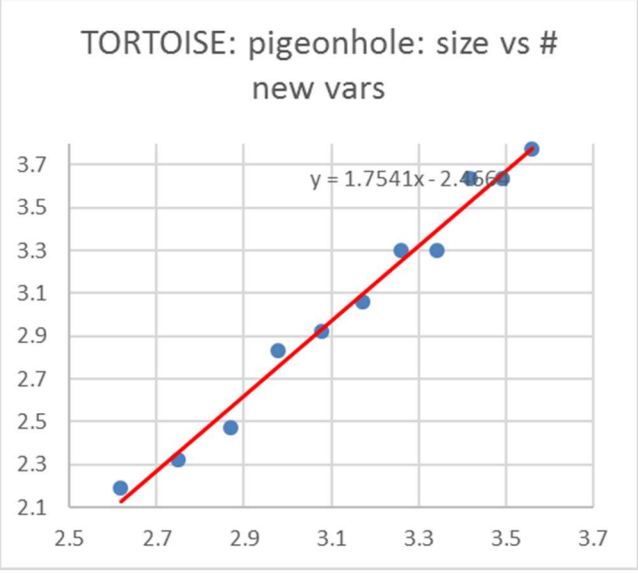
From LEMMA 9, we see that the runtime of DeepSAT is polynomial in the number of new variables introduced. CONJECTURE 7 says that number of new variables is polynomial in the shortest proof length. Below we discuss empirical support for CONJECTURE 7 and related matters.

5.1 Pigeonhole principle

Satisfiability problem which asserts that P pegs must be placed in P-1 holes.

Linear regression results of log-log data from table 11 and section 2.2 give these results:

EDIT NOTE: change “TORTOISE” to “DeepSAT” below



N = number of literals in the satisfiability problem.

Analysis of the data follows:
1) shortest refutation known to author [cook1976]: O(N^(4/3))
2) from plot: num_new_variables = O(N^1.75)

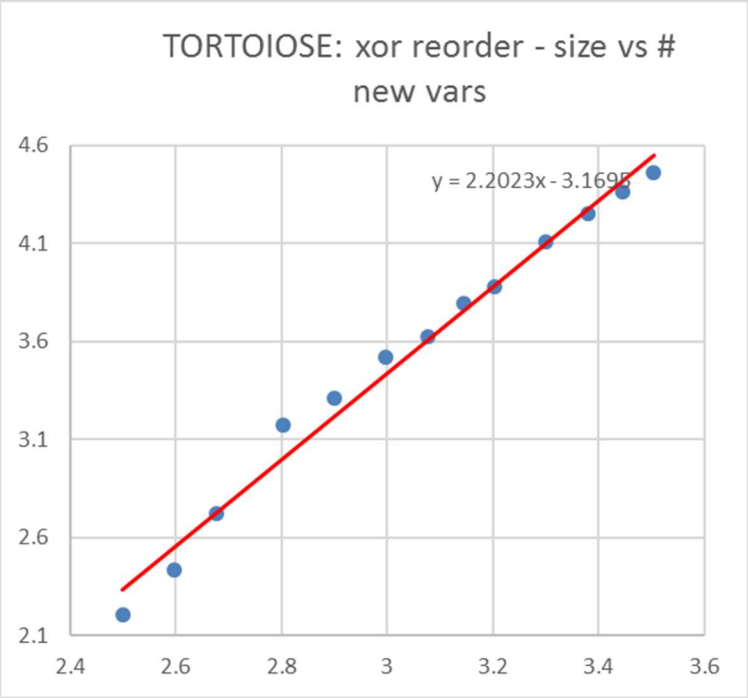
3) from equation 10 and the above data we derive
   P = 1.31
4) from equation 14 and the above we derive
   R = 2.66

5.2 XOR reordering

Satisfiability problem which asserts that N variables exclusive-or-ed together in 2 different random orders have different results for some input.

Linear regression results of log-log data from table 12 and section 2.3 give these results:

EDIT NOTE: change "TORTOISE" to "DeepSAT" below



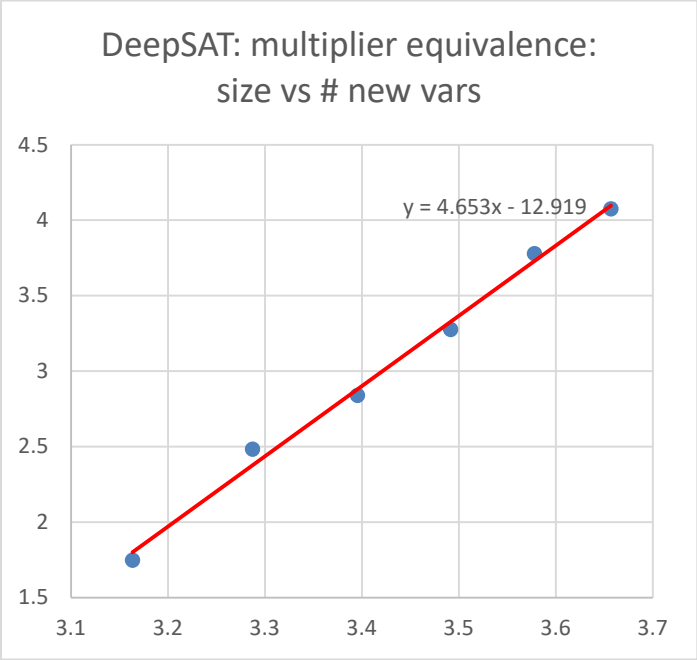
N = number of literals in the satisfiability problem.

Analysis of the data follows:
1) shortest refutation known to author: O(N^2)
2) from plot: num_new_variables = O(N^2.2)
3) from equation 10 and the above data we derive
   P = 1.1
4) from equation 14 and the above we derive
   R = 2

5.3 Multiplier equivalence/correctness

Satisfiability problem which asserts that two different randomly generated multipliers get different results for some input. To prevent the simple enumeration refutation, the product plane is used directly as input, with no input bits feeding the product plane with bit AND logic. To make the problem hard, partial sums and carry bits at each level are scrambled by a randomly generated 3-tree of full adders.

Linear regression results of log-log data from table 13 and section 2.4 give these results:



N = number of literals in the satisfiability problem.

Analysis of the data follows:

1) shortest refutation known to author: O(N^(3/2))
2) from plot: num_new_variables = O(N^4.65)
3) from equation 10 and the above data we derive
   P = 3.1
4) from equation 14 and the above we derive
   R = 1.38

5.4 Analysis

The results for XOR reordering (P = 1.1, R = 2) seem reasonably good, since P cannot be less than 1, and R = 2 is expected from random 3-SAT clauses. But the PIGEONHOLE results (P = 1.31, R = 2.66), are less comforting, and the MULTIPLIER EQUIVALANCE results (P = 3.1, R = 1.38), while not exponential (yeah!), seem pretty bad.

Hopefully better heuristics for new variable generation will lead to better values of P and R for these problems in the future.

6 DeepSAT and the P ?= NP QUESTION

The central claim of this paper is independent of the P ?= NP question.

Specifically,

LEMMA 10:
If CONJECTURE 7 and CONJECTURE 8 are true, then:

1) if co-NP == NP, then DeepSAT runs in polynomial time in
   the size of the input problem for all problems. And thus

P == NP.[3]
2) if co-NP != NP (which implies P != NP), then DeepSAT
requires worse than polynomial time for some problem types.

PROOF:

For case #1, co-NP == NP means that polynomial-sized refutations exist for all NP-complete problems. Since CONJECTURE 8 and LEMMA 9 imply that DeepSAT can solve all problems in polynomial time vs the shortest refutation size, the conclusion follows.

For case #2, no algorithm, including DeepSAT, can hope to solve problems with shortest refutation of worse than polynomial size in polynomial time, because solving the problem requires writing a refutation.

END_OF_PROOF

(Note that "co-NP == NP" is just a fancy way of saying that all unsatisfiable problems have a polynomial length refutation).

LEMMA 10 is not really new nor revolutionary. Adapting [Levin] to satisfiability gives an algorithm which must find a satisfying assignment in polynomial time if P == NP and there is a satisfying assignment. If we accept CONJECTURE 8, then we can further modify [Levin] to detect a refutation in extended resolution as well as a satisfying assignment, yielding an algorithm which is fully polynomial time for satisfiability (if P == NP). The algorithm is utterly impractical, because it has a constant factor of 2^B, where B is the number of bits required to express some program which solves satisfiability in polynomial time.

The adaptation of [Levin] works by trying all possible bit combinations, starting with shortest first, and trying to run each as a program, with a predetermined runtime limitation, and testing whether the result of the run is a satisfying assignment, or an extended resolution refutation. We successively try more bit combinations, and allow them to run for longer runtime before killing them. If we arrange the details carefully the claimed polynomial time result follows [Levin].

What is new about DeepSAT vs [Levin] is that DeepSAT can actually solve real problems in acceptable runtimes, which our adaptation of [Levin] cannot. Unfortunately, our result also requires CONJECTURE 7, as well as CONJECTURE 8.

The author has been unable to find any class of satisfiability problem instances which violate CONJECTURE 7. This suggests that either CONJECTURE 7 is true, or else classes of problems which violate CONJECTURE 7 are rare and/or contrived. CONJECTURE 8 is believed by many but is unproven.

If we accept CONJECTURE 7 and CONJECTURE 8 as true, and thus we also accept LEMMA 10, we might hope that we can settle the P ?= NP question empirically. That is, by running DeepSAT on satisfiability problem instances chosen randomly, in a manner which we think samples the space of all problems in a reasonable way. If we empirically find exponential growth of runtime with problem size, (and equivalently exponential growth in number of new variables needed with problem size), this implies P != NP. If, on the other hand, we empirically find polynomial growth of runtime with problem size, (and equivalently polynomial growth in number of new variables needed with problem size), this implies that either P == NP, or else violating classes of problems requiring exponential runtime are rare and/or contrived.

It is not possible to rule out by empirical measurements the possibility of rare and/or contrived classes of problems that go exponential, but this fact probably has little practical significance. If DeepSAT almost always solves problems in polynomial time, and nobody can find a counterexample, this is all we need to know to use DeepSAT as a practical solution to satisfiability problem.

We need a complete theoretical proof that relies on no conjectures to definitively rule out any exponential behavior, and this author doubts such a result will appear any time soon.

Mathematical purists might be offended by our lack of complete proof of CONJECTURE 7 and LEMMA 10, and our reliance instead on empirical support for CONJECTURE 7. Please remember that the main support for the widely believed P != NP conjecture is also empirical, namely, the failure of many researchers to find any algorithm which solves satisfiability problem in polynomial runtime.

The situation of DeepSAT and SATISFIABILITY reminds this author of the situation of the simplex method and linear programming before the 1970's. The simplex method seemed to run fast on most or all linear programming problem instances, but no proof existed explaining why this is.

[3] DeepSAT is a randomized algorithm and the randomization seems important for good performance. Strictly speaking, the P ?=NP questions refers to deterministic algorithms only, not randomized algorithms. I am ignoring that technicality in this paper.

Then in 1972 Klee&Minty published a class of linear programming problem instances that cause the simplex algorithm to require exponential runtime [Klee&Minty1972]. But then finally in 1979 the complexity of linear programming was settled with the publication of the ellipsoid algorithm, which has provably polynomial runtime on all problem instances [khachian1979]. Nevertheless, the simplex algorithm continues to be widely used in practice in spite of the Klee-Minty result, and the ellipsoid algorithm is never used.

7 SUMMARY AND CONCLUSION

DeepSAT is a new program for the satisfiability problem which extends the conflict-driven-clause-learning method to solve many problems previously unsolvable by general-purpose satisfiability solvers. The key innovation is introducing new variables during the run, as well as a few other features needed to make introduction of new variables effective. Empirical data and theoretical analysis give hope that DeepSAT runs in polynomial time for most or all problem classes, and also hope of shedding some light on the P ?= NP question.

The polynomial growth exponents of DeepSAT on several problem classes are much worse than what we might hope for from theory, and so we hope that future improvements in the heuristics of DeepSAT can further improve the runtime polynomial growth exponents.

8 APPENDIX - PROOF of THEOREM 1

THEOREM 1:
It is always possible to transform "shortest_proof" into another refutation which has these properties:
1) The new refutation uses only length-3 and smaller clauses
2) The new refutation requires the definition of no more than shortest_proof_length new variables.

PROOF:

We start by compressing the input satisfiability problem into length-3 and smaller clauses. We do this by defining new variables which can be substituted into clauses of the satisfiability problem to shorten the clauses. Every substitution shortens a clause by 1 literal.

For example, if the original problem contains the clause

(A|B|C|D|E)

then we can define the new variable

NEWVAR = A|B

and then substitute into the clause, giving

(NEWVAR|C|D|E)

Strictly speaking, we allow only resolution and new variable definitions in our proofs. But we can accomplish substitution by resolution. See the formal extended resolution proof below:

```
(s1)          A|B|C|D|E       - axiom;

/* NEWVAR = A | B */
(def_newvar) define NEWVAR = truth_table(A,B) : 0111 ;
(def_newvar.s1) !NEWVAR | A | B - from definition (def_newvar);
(def_newvar.s2) NEWVAR | !A     - from definition (def_newvar);
(def_newvar.s3) NEWVAR | !B     - from definition (def_newvar);

/* substitute NEWVAR into (s1) using resolution */
(p1)          NEWVAR|B|C|D|E  - resolve (def_newvar.s2),(s1);
(p2)          NEWVAR|C|D|E    - resolve (def_newvar.s3),(p1);
```

Our formal extended resolution proof is in the form:

```
(label)      <statement>    - <reason statement is true>;
```

Note that if we substitute into a length-3 clause, the resolution steps involve only length-3 and smaller clauses. For example, if we have the clause

```
(A|B|C)
```

and the new variable definition

```
NEWVAR = A|B
```

and we wish to substitute to obtain

```
(NEWVAR|C)
```

we can do this with the following extended resolution proof:

```
(s1)          A|B|C          - axiom;

/* NEWVAR = A | B */
(def_newvar) define NEWVAR = truth_table(A,B) : 0111 ;
(def_newvar.s1) !NEWVAR | A | B - from definition (def_newvar);
(def_newvar.s2) NEWVAR | !A     - from definition (def_newvar);
(def_newvar.s3) NEWVAR | !B     - from definition (def_newvar);

/* substitute NEWVAR into (s1) using resolution */
(p1)          NEWVAR|B|C      - resolve (def_newvar.s2),(s1);
(p2)          NEWVAR|C        - resolve (def_newvar.s3),(p1);
```

We can also reverse substitute using resolution. For example, if we have the clause

```
(NEWVAR|C)
```

and we want to reverse substitute

```
NEWVAR = A|B
```

to obtain

```
(A|B|C)
```

we can do this using extended resolution as follows:

```
/* NEWVAR = A | B */
(def_newvar) define NEWVAR = truth_table(A,B) : 0111 ;
(def_newvar.s1) !NEWVAR | A | B - from definition (def_newvar);
(def_newvar.s2) NEWVAR | !A     - from definition (def_newvar);
(def_newvar.s3) NEWVAR | !B     - from definition (def_newvar);

(s1)          NEWVAR|C        - axiom;

/* reverse substitute NEWVAR into (s1) using resolution */
(p1)          A|B|C          - resolve (def_newvar.s1),(s1);
```

Note that reverse substituting to eliminate a new variable from a length-2 clause also requires resolution steps involving length-3 and smaller clauses.

We now show how to compress the body of shortest_proof into another proof involving only length-3 and shorter clauses, using substitution of new variables into length-3 clauses, and reverse substitution to eliminate new variables from length-2 clauses.

Of course shortest_proof consists of a series of resolution steps. For example

(A|B|C|D|E|R)&(!R|F|G|H|I|J) --> (A|B|C|D|E|F|G|H|I|J)

Assume the 2 antecedent clauses have already been compressed to 3 clauses. If we are lucky, the resolvent variable R is still exposed, for example:

NEWVAR_AB = A|B
NEWVAR_CD = C|D
NEWVAR_ABCD = NEWVAR_AB | NEWVAR_CD
NEWVAR_FG = F|G
NEWVAR_HI = H|I
NEWVAR_FGHI = NEWVAR_FG | NEWVAR_HI

Substituting into the antecedent clauses gives

(NEWVAR_ABCD|E|R)&(!R|NEWVAR_FGHI|J)

Doing resolution to combine these clauses is illegal since it produces a length-4 clause. So we must define another new variable to shorten one of the antecedent clauses, for example

NEWVAR_ABCDE = NEWVAR_ABCD | E

Substituting into the first antecedent clauses allows us to derive a length-3 clause which is equivalent to the derived clause of the original step:

(NEWVAR_ABCDE|R)&(!R|NEWVAR_FGHI|J) --> (NEWVAR_ABCDE|NEWVAR_FGHI|J)

If we are not lucky, the antecedent clauses have been compressed so that the resolvent variable is hidden in the new variables. For example, original step

(A|B|C|D|E|R)&(!R|F|G|H|I|J) --> (A|B|C|D|E|F|G|H|I|J)

compressed with

NEWVAR_CD = C|D
NEWVAR_ER = E|R
NEWVAR_CDER = NEWVAR_CD | NEWVAR_ER
NEWVAR_nRF = !R | F
NEWVAR_GH = G|H
NEWVAR_nRFGH = NEWVAR_nRF | NEWVAR_GH

giving us antecedent clauses

(A|B|NEWVAR_CDER)&(NEWVAR_nRFGH|I|J)

There is no way to combine these clauses to give the derived clause. The problem is that the resolvent variable R is hidden in the new variables that have been substituted into the antecedent clauses.

To fix this, we use a series of substitutions and reverse substitutions to expose R in both antecedent clauses. For example, for the clause

(A|B|NEWVAR_CDER)

we can first compress by defining

NEWVAR_AB = A|B

and substituting to yield

(NEWVAR_AB | NEWVAR_CDER)

Now we can reverse substitute to eliminate NEWVAR_CDER:

(NEWVAR_AB | NEWVAR_CD | NEWVAR_ER)

Now define

NEWVAR_ABCD = NEWVAR_AB | NEWVARCD

and substitute, yielding

(NEWVAR_ABCD | NEWVAR_ER)

Now we can reverse substitute to eliminate NEWVAR_ER, yielding

(NEWVAR_ABCD | E | R)

Note that all of the substitutions and reverse substitutions require resolution involving length-3 and smaller clauses. Also note that the number of new variables defined is <= the number of literals of the antecedent clause.

We do this process on the 2nd antecedent clause as well, and we have 2 length-3 clauses both with resolvent variables exposed. Now we do what we did in the "lucky" case above: define a new variable to compress one of the antecedent clauses into a length-2 clause, and the perform resolution on the 2 antecedent clauses to derive a compressed derived clause.

We can do this process on all the resolution steps of shortest_proof, starting from the original satisfiability problem compressed to length-3 clauses, and working down the proof in order of the steps of shortest_proof.

The resulting compressed proof satisfies the claims of THEOREM 1.

END_OF_PROOF

ACKNOWLEDGEMENTS

This work was supported by my employer, Synopsys, Inc., by paying my salary as I did this work. Numerous individuals also provided invaluable assistance. Lawrence Ryan spent hours helping me understand conflict-driven clause learning and other aspects of modern SAT solvers, and kicking around ideas for introducing new variables most effectively. Ross Donnelly helped speed up the code of DeepSAT in numerous ways and provided valuable feedback on this paper. Discussions with Jay Adams led to the idea for the maximum compression heuristic for new variable definition. Brent Gregory supported and encouraged me in the early days of this work, when it was obvious to everybody else that I was just another crackpot trying to prove P=NP, and in the dark days when nothing I tried worked. Brent also spent countless hours reading various manuscripts and providing feedback on the manuscripts and the algorithms. Chris Wilkins worked as an intern on the project for one summer, in the dark days when we had some promising results on XOR EQUIVALENCE, but we could not get robust performance on other problems and we did not know why. Chris ran numerous testcases and worked tirelessly analyzing what was going wrong. Jerry Burch provided several verification testcases of commercial interest and helped me run them and analyze the results, and also provided valuable feedback on the algorithms of DeepSAT. Paul Filseth was a great sounding board over many years to help me refine and perfect many of the ideas in DeepSAT. Robert Damiono, Ted Stanion, Jerry Burch, Reilly Jacobi, Paul Filseth, and Lawrence Ryan provided me with numerous interesting testcases that helped me improve DeepSAT.

Finally, thanks to Shankar Krishnamoorthy, who provided corporate support for this work and who suggested the name “DeepSAT”. DeepSAT runs slower than traditional SAT solvers because it has more overhead, but this overhead pays off in the end on more difficult problems.

REFERENCES CONSULTED

[cook1976] Stephen Cook, "A Short Proof of the Pigeon Hole Principle Using Extended Resolution", SIGACT News, Oct-Dec 1976, pp 28-32.

[chaff2001] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an Efficient SAT Solver, 39th Design Automation Conference (DAC 2001), Las Vegas, ACM 2001.

[chvatal1988] Vasek Chvatal and Endre Szemeredi, "Many Hard Examples for Resolution", Journal of the ACM, vol 35 (1988), pp. 759-768

[Garey&Johnson] Garey, M.R.; Johnson, D.S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness.

[khachian1979] L. G. Khachiyan, A polynomial algorithm in linear programming, Dokl. Akad. Nauk SSSR 244 (1979), no. 5, 1093-1096. MR 522052 (80g:90071)

[Klee&Minty1972] Klee, Victor; Minty, George J. (1972). "How good is the simplex algorithm?". In Shisha, Oved. Inequalities III (Proceedings of the Third Symposium on Inequalities held at the University of California, Los Angeles, Calif., September 19, 1969, dedicated to the memory of Theodore S. Motzkin). New York-London: Academic Press. pp. 159-175

[levin] https://en.wikipedia.org/wiki/P_versus_NP_problem#Polynomial-time_algorithms

[tseitin1970] G S Tseitin, "On the complexity of derivations in the propositional calculus", Studies in Mathematics and Mathematical Logic, Part II, A O Slisenko, ed., 1970, pp 115-125.

[urquahart1987] Alasdair Urquahart: "Hard Examples for Resolution", Journal of the ACM, Vol 34, 1987, pp 209-219.

[urquahart1995] Alasdair Urquahart: "The Complexity of Propositional Proofs" in The Bulletin of Symbolic Logic, Volume 1, Number 4, Dec 1995, pages 425-467