

Resolution proof for look-ahead SAT solvers

Shanny Anoep, El Drijver, Arvind Ganga, Martijn Kirsten
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

November 12, 2006

Abstract

This paper describes our efforts to add resolution proof generation, a feature currently only available in some conflict-driven SAT solvers, to the existing look-ahead SAT solver `satz`. We demonstrate this is feasible, although our current implementation adds considerable overhead. The resolution proof is generated while backtracking, which provides runtime information that can be used to reduce execution time. We present the results of our solver on a variety of CNF formulas and compare our extended solver with `minisat`, a conflict-driven solver, showing the resolution proof size of our look-ahead solver is often considerably shorter than `minisat`'s. The proof can be used to extract an UNSAT core of the CNF formula. The UNSAT core of the proof generated by our enhanced version of `satz` is generally shorter than the one by `minisat`. With a different implementation of the resolution proof generation even shorter UNSAT cores can be extracted, although with increased overhead.

1 Introduction

For solving the Boolean satisfiability problem different kinds of methods exist. For the deterministic methods a distinction can be made between conflict-driven and look-ahead solvers. Some conflict-driven solvers can give a certificate to proof unsatisfiability of a formula, based on resolution. Currently, there are no publicly known look-ahead solvers that provide a certificate. This document describes a method to accomplish this for look-ahead solvers. In this section a brief introduction is given to the Boolean satisfiability problem, above mentioned solving methods and the problem definition.

1.1 Preliminaries

The Boolean satisfiability problem (SAT) is a well-known NP-complete problem. There is a lot of interest in solving this problem, because it can be used

to solve a range of other problems, such as planning and scheduling, hardware verification and model checking. SAT itself is the problem of deciding whether a Boolean formula has an assignment that evaluates it to true. A Boolean formula consists of a combination of Boolean variables, which can take on the values true or false, and Boolean operations AND, OR and NOT.

In SAT problems a special form of Boolean formulas is considered, the conjunctive normal form (CNF). A literal is a Boolean variable or a negated (\neg) variable. A clause is a number of literals connected by the logical OR (\vee) operator. In CNF all clauses in a formula are connected by the logical AND (\wedge) operator, for example: $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3)$. All Boolean formulas can be transformed into an equivalent formula that is in CNF.

A Boolean formula is satisfiable if there is an assignment of true and false to the variables, such that the formula evaluates to true. The SAT problem is to test whether such an assignment exists. The SAT problem can be specified by fixing the number of variables per clause, called k -SAT problems for k variables per clause. The time complexity of solving the problem depends on the value of k . For instance, 1-SAT and 2-SAT are solvable in linear time, but everything above that requires exponential time, assuming $P \neq NP$. However, algorithms have been devised that can solve SAT problems more efficiently on average.

2 Solving methods

Over the years a lot of different solvers have been developed. These can be categorized based on their solving methods. Deterministic solvers always find a solution to a SAT-instance if there is any, whereas stochastic solvers use probabilistic methods.

In this report we only discuss deterministic methods, more specifically solvers based on the Davis-Putnam-Logemann-Loveland algorithm [3]. This is a backtracking-based solution, which is used in most SAT solvers today and has proven to be an efficient method. Different variants exist of the algorithm, with the main categories being look-ahead solvers and conflict-driven solvers, of which only the former is discussed in this paper.

2.1 Look-ahead solvers

After making an incorrect decision at a node in a (recursive) DPLL search tree, it takes time to backtrack up the tree again and switch to another branch. Look-ahead algorithms can be used to make sure that the 'correct decisions' are made, in other words, 'look-ahead' what the outcome will be. These algorithms use heuristics to determine the best next literal to set.

Several heuristics exist: For example, for every literal at a certain depth of the tree, one can calculate the number of binary clauses that is generated after setting the literal either true (left) or (right) false. It compares these

by using the following formula: $left * right + left + right$, the literal with the highest value is chosen. Instead of the number of new binary clauses other metrics can be used, such as the number of variables that disappeared from the formula, or number of clauses that shrinks, after assigning true to a variable.

An optimization can be made with the concept of failed literals. A branch literal is 'failed' when it conflicts after setting it initially to true and executing a unit propagation. When a branch contains a failed literal, extra information is obtained. It means that the value of the literal is fixed. Therefore the metrics have to be recalculated, after which a possibly more effective branch literal is chosen. The literals can be sorted such that a minimal number of calculations have to be done after finding a failed literal. Another possible optimization is the use of partial look-ahead functions, which only considers a selection of literals. These algorithms can have the same result, but with less work and time.

Look-ahead solvers have proven to be most effective on proving unsatisfiability of random SAT-formulas. There's an annual competition for SAT-solvers, in which conflict-driven and look-ahead solvers compete. In recent years, there have been only four publicly known look-ahead solvers in the competition: `satz`, `kcnfs`, `march` and `OKsolver`.

3 Certificate generation for look-ahead solvers

The ultimate objective of this project is to generate a certificate of unsatisfiability for a look-ahead solver. Currently, some conflict-driven solvers can give a certificate of unsatisfiability, which serves as a proof that the solver has taken the correct steps to conclude unsatisfiability. This kind of certificate can be used to discover faulty solvers, for instance in a SAT-solving competition. A solver that always immediately outputs "unsatisfiable" for a formula will do very well for only unsatisfiable formulas. When a certificate is available, the output can be verified.

The certificates generated by conflict-driven solvers are based on resolution, a method for proving (un)satisfiability using 1 simple rule. No look-ahead solver has this kind of functionality yet. A brief introduction on resolution proof is given below, after which our method of generating such a method from a DPLL tree is explained. Since look-ahead solvers effectively construct and traverse a DPLL tree, we can generate resolution proofs for look-ahead solvers using this method.

3.1 Resolution proofs

Resolution is a complete proof procedure for propositional logic that works on formulas expressed in conjunctive normal form [4]. From two clauses (A, x) and $(B, \neg x)$, where x is a literal and A and B are sets of literals, the

resolution rule generates the new clause (A,B), which is called the resolvent of the original clauses. The elements of the resolvents are the result of the union of sets A and B. The literal x is called the variable resolved on, sometimes also called the clash literal. If duplicate literals are present in the resolvent, the duplicate is removed. The notation we will conform to for applying the resolution rule is: $C = R(C_1, C_2)$.

A resolution proof π of a CNF formula F is a sequence of clauses C_1, C_2, \dots, C_m , such that each C_i is either a member of F , or C_i is a resolvent of two previous clauses: $C_i = R(C_j, C_k)$ with $j, k < i$, and C_m is the empty clause. In other words, if there is a sequence of resolution steps that leads to the empty clause, the entire formula is always unsatisfiable. If however, no more solution steps are possible, the formula is satisfiable.

3.2 Proof algorithm

As explained in the introduction, look-ahead solvers work according to the DPLL algorithm. Essentially this algorithm considers all possible assignments as a tree. A first variable is set, and in subsequent levels another one is set, until either a satisfying assignment or a conflicting assignment is found after unit propagation. Unit propagation can be seen as a cascade effect and occurs when a single literal remains of one of the original clauses. This literal is propagated to the remaining clauses, after which another unit clause may occur, and unit propagation can continue, until all clauses are either satisfied, a conflict occurs or no further unit clauses are created. The DPLL algorithm is based upon the Davis-Putnam algorithm, which is actually based on resolution. Therefore it makes sense we can use the results of the DPLL algorithm to create our resolution proof.

The transformation from DPLL tree to resolution proof is best explained by an example. Consider the following formula in CNF:

1. (x_1, x_2, x_3)
2. $(\neg x_1, x_2, x_3)$
3. $(\neg x_2, x_3)$
4. $(x_1, \neg x_2, \neg x_3)$
5. $(\neg x_1, \neg x_2, \neg x_3)$
6. $(x_2, \neg x_3)$

Let's assume a (look-ahead) solver takes this formula as input, and decides to branch on the variable x_3 by setting it true. After setting it true all clauses that are satisfied are removed. One unit clause is generated (clause 3), so unit propagation is performed, after which a conflict is discovered. The

| | |
|--|---|
| branch on $x_3 = 0$ | branch on $x_3 = 1$ |
| remaining clauses: 1. (x_1, x_2) 2. $(\neg x_1, x_2)$ 3. $(\neg x_2)$ | remaining clauses: 4. $(x_1, \neg x_2)$ 5. $(\neg x_1, \neg x_2)$ 6. (x_2) |
| unit propagation $\neg x_2$ | unit propagation x_2 |
| remaining clauses: 1. (x_1) 2. $(\neg x_1)$ | remaining clauses: 4. (x_1) 5. $(\neg x_1)$ |
| unit propagation x_1 | unit propagation x_1 |
| remaining clauses: 2. $(\neg x_1)$ | remaining clauses: 5. $(\neg x_1)$ |
| conflict | conflict |

Table 1: *Example of DPLL-branching*

| | |
|----------------------------|--------------------------------------|
| branch on $x_3 = 0$ | branch on $x_3 = 1$ |
| r1 = R(1,2) = (x_2, x_3) | r3 = R(4,5) = $(\neg x_2, \neg x_3)$ |
| r2 = R(r1,3) = (x_3) | r4 = R(r3,6) = $(\neg x_3)$ |
| r5 = R(r4,r2) = $()$ | |

Table 2: *Example resolution resulting in empty clause*

solver will now backtrack and assign the variable x_3 to false, after which the same routine is executed. Both branches lead to conflicts, so we proved that the entire formula is unsatisfiable. The steps taken are depicted in detail in table 1.

A resolution proof of the refutation of the formula can be built from the DPLL-tree by applying the unit propagation steps in reverse order. From the example, start in the left bottom node. The last unit propagation was x_1 on clauses 1 and 2. The resolution of the original clauses 1 and 2 on literal x_1 is (x_2, x_3) . The previous unit propagation was x_2 , originating from clause 3. The resolution of clause 3 and the resolution clause just derived on literal x_2 is (x_3) .

Continuing at the right node, in a similar way we find resolution clause $(\neg x_2, \neg x_3)$ from original clauses 4 and 5 on clash literal x_1 . The resolution of this clause and clause 6 on clash literal x_2 yields $\neg x_3$. The resolution of the two derived resolutions, $\neg x_3$ and x_3 , is the empty clause. This proves the DPLL-tree rightfully classified the formula as unsatisfiable. A step by step resolution proof is depicted in table 2.

3.3 Proof algorithm applied to a look-ahead SAT-solver

A look-ahead SAT-solver performs unit propagation after instantiating a branch literal. In every node, it is necessary to keep track of the unit propagations, the originating clause of the propagated literal, and the order in which the propagations were applied. This information needs to be added to the parts of the solver where unit propagation is performed. Whenever branching is performed, the branch variable and its originating clause need to be stored for that node as well.

When a branch is unsatisfiable backtracking is performed. While backtracking, the resolvent for the node that is left is determined. Assume the solver visits a left branch, before it visits a right branch. When backtracking from a left child node to its parent, the resolvent of the child node is determined. After that, the right branch is visited. When backtracking from this right child node, the resolvent of the right child node is determined. Now both the left and right sub tree of the current node have been visited. The resolvent of these two sub trees is determined and backtracking continues. When the DPLL-tree is unsatisfiable the root node will be visited after exploring the whole tree. When this node's left and right child resolutions resolve to the empty clause, the refutation of the formula is valid.

The following algorithm is added to the solver in order to create the resolution proof:

```
if (backtracking) {
  if (current node == leaf node) {
    determine resolvent for current node
  } else {
    determine resolvent of child nodes' resolvents
    determine resolvent for current node
  }

  if (current node == root node) {
    if (resolvent == empty clause) {
      output "Valid refutation"
    } else {
      output "Invalid refutation"
    }
    exit
  }
}
```

Special attention is needed when resolving unit propagations. The look-ahead solver can propagate literals which are irrelevant for the final resolution. We can decide to neglect a unit propagation if the propagated literal does not occur in the current resolution. After neglecting the literal the next

| skipping propagated literal | | | |
|-----------------------------|---------|--------------|--------|
| 1. | 2. | 3. | 4. |
| Node X | Node X | Node X | Node X |
| UP | UP | UP | UP |
| 1 | 1 | 1 | 1 |
| 4 | 4 | 4 | (-1) |
| 5 | 5 | skip literal | |
| 3 | (-4,-1) | | |
| (-3,-4,-1) | | | |

Table 3: *Example of skipping unit propagations*

propagation literal can be resolved with the current resolution. An example of skipping propagated literals when creating resolutions is given in table 3. Here unit propagation of literal 5 is performed, but 5 doesn't occur in the node's resolution so it is skipped in step 3.

Another problem which can arise when creating a node resolution is that the solver chose and branched on a literal that doesn't occur in one of the two resolutions needed to create that node's resolution.

This problem can be overcome by ignoring the resolution in which the node literal is contained. By setting the resolution in which the node literal is not contained as the node resolution, the correct resolution can be obtained.

We noticed that for so-called pigeon hole formulas this problem doesn't occur at all, while for random instances the number of ignored branch literals can vary.

3.4 Choosing a solver

We had the choice of several look-ahead solvers to modify for this project. We looked at the following four candidates: KCNFS, March, OKsolver and `satz`. The sourcecode was available from all of these, so we could judge them equally. Because we were going to have to edit the existing source code, the comments and readability of the original code was very important. We scratched two of the options for having either no comments at all or comments in a foreign language which would just further complicate adding modifications. The remaining two were March and `satz`. March seemed to have better commenting and readable code, but had more complex features. We chose `satz` eventually because of this last aspect. It's a bit smaller and simpler, so it wouldn't take too much time to learn how the solver worked and we could focus on implementing our resolution proof generation.

4 Experimental results

We compared the performance of the modified `satz`-program to the original version of the program to determine the difference in execution time. We also compared with `minisat`, a conflict-driven solver that has the option to generate a resolution proof. Our primary interest is in differences in the size of the resolution proofs generated by these different solver architectures. From the resolution proof the unsat core of the formula is generated, for both solvers. The sizes of the unsat cores are also compared. Tests were carried out on a computer with an Intel Pentium D at 3,2Ghz with 2GB of memory.

`satz` was run with the `-s` switch to turn off preprocessing, because in the modified `satz` version the preprocessing function is disabled by default. This is because preprocessing adds complexity to the clause management but doesn't alter the branching, backtracking or resolution proof generating algorithms, so we chose to ignore this feature for our proof-of-concept implementation. Another feature of `satz` is the double look-ahead function, which is also disabled in the modified `satz` version. The implementation of the double look-ahead would also add complexity to generating the resolution proof. By disabling both features, the performance of `satz` will most certainly be decreased.

4.1 Instances

For testing purposes the formulas were selected from the benchmark problems found at SATLIB [2]. We were interested in the performance of both random instances and structured instances for comparing between the different architectures. We used the resolution proof checker and its format as proposed by Allen van Gelder for the 2005 SAT Competition [1]. As the resolution proof checker has a 2 GB input file size limit, our test set contains only formulas that generated resolution proofs smaller than 2 GB and that were solvable in a reasonable amount of time on our test machine. All of the instances used are unsatisfiable. The division into categories and the amount of variables and clauses of the instances are shown in table 4. The test instances are divided into the following categories:

Uniform Random-3-SAT We used nine unsatisfiable random 3-SAT instances. These can be divided in three groups with either 100, 200 or 250 variables. These different problem sizes were picked to see what the impact of a higher amount of variables has on the resolution proof.

Single Stuck At Fault These instances are from a circuit fault analysis. A test-pattern generation program, Nemesis, generated these instances to test circuits on faults that aren't detectable by default simulation. Even though these instances contain a lot of clauses they are relatively

| instance | type | variables | clauses |
|-----------------|-----------------------|-----------|---------|
| uuf100-0116.cnf | Uniform random 3-SAT | 100 | 430 |
| uuf100-0154.cnf | | 100 | 430 |
| uuf100-0186.cnf | | 100 | 430 |
| uuf200-020.cnf | | 200 | 860 |
| uuf200-059.cnf | | 200 | 860 |
| uuf200-090.cnf | | 200 | 860 |
| uuf250-021.cnf | | 250 | 1065 |
| uuf250-053.cnf | | 250 | 1065 |
| uuf250-076.cnf | | 250 | 1065 |
| ssa0432-003.cnf | Single Stuck at Fault | 435 | 1027 |
| ssa6288-047.cnf | | 10410 | 34238 |
| hole6.cnf | Pigeon hole | 42 | 133 |
| hole7.cnf | | 56 | 204 |
| hole8.cnf | | 72 | 297 |
| hole9.cnf | | 90 | 415 |
| pret60-25.cnf | Encoded 2-coloring | 60 | 160 |
| pret60-75.cnf | | 60 | 160 |

Table 4: *Instances of CNF files with their properties*

easy to solve. They contain clauses of lengths 1-6 and usually more than half are binary

Pigeonhole Pigeonhole formulas are well known UNSAT problems for which the (deterministic) solver will have to search the entire search space before concluding unsatisfiability. This property makes them relatively hard UNSAT-instances

Encoded 2-colouring These instances are generated two-colouring graph problems, forced to be unsatisfiable.

4.2 Results

4.2.1 Timing performance

Table 5 shows the time performance of `satz`, `satz` with proof and `minisat` on the selected instances. We compared the CPU-time used by the original version of `satz` and the resolution-proof enhanced version. We also compare `minisat` with and without the option of generating a resolution proof enabled.

We see `satz`'s slowdown when generating a resolution proof is dramatic, especially when compared to `minisat`'s slowdown when generating a resolution proof. The overhead of generating the proof for `minisat` is almost

| instance | CPU-time used | | | |
|-----------------|---------------|------------|---------|---------------|
| | satz | satz proof | minisat | minisat proof |
| uuf100-0116.cnf | 0.019s | 1.23s | 0.012s | 0.012s |
| uuf100-0154.cnf | 0.020s | 1.23s | 0.016s | 0.012s |
| uuf100-0186.cnf | 0.015s | 1.272s | 0.008s | 0.008s |
| uuf200-020.cnf | 0.0146s | 3.480s | 0.252s | 0.284s |
| uuf200-059.cnf | 0.175s | 4.277s | 0.380s | 0.416s |
| uuf200-090.cnf | 0.173s | 4.379s | 0.476s | 0.532s |
| uuf250-021.cnf | 1.289s | 40.000s | 11.949s | 10.321s |
| uuf250-053.cnf | 0.995s | 26.867s | 9.541s | 7.184s |
| uuf250-076.cnf | 0.936s | 22.038s | 3.588s | 3.516s |
| ssa0432-003.cnf | 0.029s | 1.750s | 0.004s | 0.004s |
| ssa6288-047.cnf | 0.225s | 0.082s | 0.032s | 0.032s |
| hole6.cnf | 0.0155s | 1.531s | 0.008s | 0.016s |
| hole7.cnf | 0.043s | 2.761s | 0.080s | 0.088s |
| hole8.cnf | 0.259s | 14.502s | 0.4920s | 0.5040s |
| hole9.cnf | 1.176s | 2m7s | 5.843s | 5.052s |
| pret60-25.cnf | 21.282s | 5m57s | 0.004s | 0.008s |
| pret60-75.cnf | 21.282s | 5m46s | 0.008s | 0.008s |

Table 5: *Comparison of the performance of the solvers used*

negligible, while `satz` with proof is at least an order of magnitude slower. An important reason for this is our primary goal was to add a fully functional correct resolution proof generator to `satz`, without worrying about creating the most optimal implementation. Our implementation can most certainly be improved upon by using more efficient data structures and better manipulation of those structures in the code that generate the resolution proof.

4.2.2 Resolution proof size

Next we compared the resolution size of the proofs generated by `satz` and `minisat`, shown in table 6. `minisat` generates a binary proof file that is not readable by van Gelder’s resolution proof checker program. It uses chaining as a proof format which is a more compact way of representing a resolution proof. An example of such a proof and how to convert it to a resolution proof readable by van Gelder’s proof checker is included in appendix C. Sample proofs by `satz` and `minisat` of one of the instances are included in appendices B and C.

The proofs generated by `satz` are considerably shorter than those generated by `minisat`, except for the Encoded 2-Coloring instances and half of the Pigeonhole instances.

| instance | resolution size | |
|-----------------|-----------------|-----------|
| | satz | minisat |
| uuf100-0116.cnf | 2,738 | 9,969 |
| uuf100-0154.cnf | 2,733 | 12,049 |
| uuf100-0186.cnf | 2,586 | 6,012 |
| uuf200-020.cnf | 60,303 | 248,154 |
| uuf200-059.cnf | 81,362 | 358,652 |
| uuf200-090.cnf | 84,152 | 446,893 |
| uuf250-021.cnf | 972,046 | 7,186,946 |
| uuf250-053.cnf | 808,396 | 5,237,578 |
| uuf250-076.cnf | 649,722 | 2,555,313 |
| ssa0432-003.cnf | 5,207 | 1,178 |
| ssa6288-047.cnf | 24 | 107 |
| hole6.cnf | 6,492 | 9,519 |
| hole7.cnf | 45,500 | 53,863 |
| hole8.cnf | 364,072 | 344,135 |
| hole9.cnf | 3,276,738 | 2,752,551 |
| pret60-25.cnf | 11,965,311 | 6,291 |
| pret60-75.cnf | 11,965,311 | 7,072 |

Table 6: *Size of the resolution proofs generated by satz and minisat, measured in resolution steps*

| instance | # UPs | # UPs used | # UPs ignore | % ignored |
|-----------------|------------|------------|--------------|-----------|
| uuf100-0116.cnf | 4,922 | 2,562 | 2,360 | 48% |
| uuf100-0154.cnf | 4,750 | 2,565 | 2,185 | 46% |
| uuf100-0186.cnf | 4,637 | 2,410 | 2,227 | 48% |
| uuf200-020.cnf | 126,665 | 57,400 | 69,265 | 55% |
| uuf200-059.cnf | 168,884 | 77,473 | 91,411 | 54% |
| uuf200-090.cnf | 173,004 | 80,134 | 92,870 | 54% |
| uuf250-021.cnf | 2,185,594 | 929,789 | 1,255,805 | 57% |
| uuf250-053.cnf | 1,778,605 | 773,212 | 1,005,393 | 57% |
| uuf250-076.cnf | 1,446,052 | 621,099 | 824,953 | 57% |
| ssa0432-003.cnf | 7,719 | 5,032 | 2,687 | 35% |
| ssa6288-047.cnf | 97 | 24 | 73 | 75% |
| hole6.cnf | 6,103 | 5,773 | 330 | 5% |
| hole7.cnf | 42,771 | 40,461 | 2,310 | 5% |
| hole8.cnf | 342,233 | 323,753 | 18,480 | 5% |
| hole9.cnf | 3,080,179 | 2,913,859 | 166,320 | 5% |
| pret60-25.cnf | 10,851,200 | 10,851,200 | 0 | 0% |
| pret60-75.cnf | 10,851,200 | 10,851,200 | 0 | 0% |

Table 7: *Number of unit propagations used for the proof construction*

4.2.3 Useful unit propagations

As described in section 3.3 not all literals the solver branched on are needed to create a node’s resolution. These literals need to be ignored when constructing the resolution proof, and are not included in the resolution proof. Table 7 shows the number of unit propagations and the number of unit propagations ignored in the proof constructing process. The results show that the number of ignored unit propagations is substantial. This indicates that **satz** performs more unit propagations than are necessary to generate a correct resolution proof.

4.2.4 Useful resolution steps and backtracks

The proof generated by **satz** sometimes contains resolution steps that are superfluous as they are not needed for the proof. These steps are included in the proof file, because it is generated on-the-fly. At the moment of writing these steps it is unknown whether these resolution steps are still useful higher up in the DPLL-tree. These steps can be omitted to reduce the length of a proof, but for our implementation this would require storing the entire DPLL-tree and generating the resolution proof afterwards. For reasonable size instances this requires more memory than currently available on desktop machines. The number of superfluous steps is shown in table 8.

| instance | # resolution steps | # superfluous steps | % superfluous steps |
|-----------------|--------------------|---------------------|---------------------|
| uuf100-0116.cnf | 2,738 | 97 | 4% |
| uuf100-0154.cnf | 2,733 | 130 | 5% |
| uuf100-0186.cnf | 2,586 | 113 | 4% |
| uuf200-020.cnf | 60,303 | 2,363 | 4% |
| uuf200-059.cnf | 81,362 | 2,532 | 3% |
| uuf200-090.cnf | 84,152 | 2,638 | 3% |
| uuf250-021.cnf | 972,046 | 44,921 | 5% |
| uuf250-053.cnf | 808,396 | 31,633 | 4% |
| uuf250-076.cnf | 649,722 | 26,991 | 4% |
| ssa0432-003.cnf | 5,207 | 68 | 1% |
| ssa6288-047.cnf | 24 | 0 | 0% |
| hole6.cnf | 6,492 | 0 | 0% |
| hole7.cnf | 45,500 | 0 | 0% |
| hole8.cnf | 364,072 | 0 | 0% |
| hole9.cnf | 3,276,738 | 0 | 0% |
| pret60-25.cnf | 11,965,311 | 0 | 0% |
| pret60-75.cnf | 11,965,311 | 0 | 0% |

Table 8: *Number of superfluous resolution steps*

Another interesting property is the number of ignored backtracks. The construction of a resolution proof can enable the solver to ignore certain branches as described in section 3.3, which cause the superfluous resolutions. Table 9 shows the number of ignored backtracks.

4.2.5 Unsat core

From a resolution proof an unsat core of a CNF formula can be extracted. An unsat core is an unsatisfiable subset of the clauses of the CNF formula. It consists of the clauses in the CNF formula that are used in the resolution proof. We extracted the unsat cores of the proof generated by `satz` and `minisat`. The results are in table 10.

It is interesting to see that on instances 14 through 17, where `satz`' resolution proof is larger than `minisat`'s, the unsat cores are equal in size. For most other instances `satz`' unsat core is smaller than `minisat`'s, although the difference is not as large as with the resolution proof size.

As shown in table 8 not all resolution steps written to the proof are necessary for proving unsatisfiability. This property could be used to reduce `satz`' unsat cores using an alternative implementation as mentioned in 4.2.4. In the superfluous resolution steps some variables could be used that are not present in a minimal unsat core, so the unsat cores generated could be

| instance | # backtracks total | # backtracks ignored | % backtracks ignored |
|-----------------|-----------------------|-------------------------|-------------------------|
| uuf100-0116.cnf | 187 | 11 | 6% |
| uuf100-0154.cnf | 179 | 11 | 6% |
| uuf100-0186.cnf | 190 | 14 | 7% |
| uuf200-020.cnf | 3,086 | 183 | 6% |
| uuf200-059.cnf | 4,100 | 211 | 5% |
| uuf200-090.cnf | 4,232 | 214 | 5% |
| uuf250-021.cnf | 45,589 | 3,331 | 7% |
| uuf250-053.cnf | 37,575 | 2,391 | 6% |
| uuf250-076.cnf | 30,633 | 2,010 | 7% |
| ssa0432-003.cnf | 179 | 4 | 2% |
| ssa6288-047.cnf | 0 | 0 | 0% |
| hole6.cnf | 719 | 0 | 0% |
| hole7.cnf | 5,039 | 0 | 0% |
| hole8.cnf | 40,319 | 0 | 0% |
| hole9.cnf | 362,879 | 0 | 0% |
| pret60-25.cnf | 1,114,111 | 0 | 0% |
| pret60-75.cnf | 1,114,111 | 0 | 0% |

Table 9: *Number of ignored backtracks*

| instance | unsat core size | |
|-----------------|-----------------|---------|
| | satz | minisat |
| uuf100-0116.cnf | 395 | 412 |
| uuf100-0154.cnf | 407 | 414 |
| uuf100-0186.cnf | 386 | 385 |
| uuf200-020.cnf | 851 | 854 |
| uuf200-059.cnf | 860 | 860 |
| uuf200-090.cnf | 859 | 860 |
| uuf250-021.cnf | 1055 | 1065 |
| uuf250-053.cnf | 1044 | 1049 |
| uuf250-076.cnf | 1062 | 1064 |
| ssa0432-003.cnf | 360 | 492 |
| ssa6288-047.cnf | 25 | 119 |
| hole6.cnf | 133 | 133 |
| hole7.cnf | 204 | 204 |
| hole8.cnf | 297 | 297 |
| hole9.cnf | 415 | 415 |
| pret60-25.cnf | 160 | 160 |
| pret60-75.cnf | 160 | 160 |

Table 10: *Unsat core sizes of the generated proofs*

smaller, when these steps are removed. As discussed in 4.2.4, this creates additional overhead.

5 Conclusion

The objective for this project was to develop a look-ahead solver that could generate a proof of unsatisfiability for a given CNF formula. We have chosen to extend an existing solver, `satz`, which has good performance, but still is easy enough to understand and extend. By using the DPLL tree and recording all the unit propagations in that tree, we were able to construct a resolution proof on-the-fly. From the experiments we ran on this extended look-ahead solver, it is evident that the performance hit is significant. This is explainable by the heavy use of file I/O and inefficient manipulation of data structures, and disabling preprocessing and double look-ahead.

The resolution proof shows that for our varied test set, a significant number of the unit propagations `satz` performs are not necessary to prove the unsatisfiability of a CNF formula. If the performance hit could be decreased, the resolution clauses could be used to speed up the original solver by using the information on which branches are unnecessary for proving unsatisfiability. The unsat core of the resolution proofs generated by `satz` are generally shorter (and otherwise equal) in size than `minisat`'s.

This project has been a successful proof of concept for the certificate generation for look-ahead solvers. To be usable in practice more work has to be done concerning efficiency.

References

- [1] <http://www.satcompetition.org/2005>.
- [2] <http://www.satlib.org>.
- [3] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394-397, 1962.
- [4] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23-41, 1965.

A Sample input file

In appendices B and C we give proofs generated by `satz` and `minisat` on the following small example instance. The instance is in DIMACS CNF-format, which should be read as follows: it starts with the header "p cnf 6 16" indicating it is a cnf-file with 6 variables in 16 clauses. Following are the 16 clauses, given as literals in the clause and terminated by a 0.

```
p cnf 6 16
-3 -2 -1 0
 3  2 -1 0
 2  1 -3 0
 3  1 -2 0
-5 -4 -1 0
 5  4 -1 0
 4  1 -5 0
 5  1 -4 0
-6 -4 -3 0
 6  4 -3 0
 4  3 -6 0
 6  3 -4 0
 6  5  2 0
 6 -5 -2 0
 5 -6 -2 0
 2 -6 -5 0
```

B Resolution proof generated by satz

Below is the resolution proof generated by `satz` from the instance given in appendix A. It is in the format proposed by Allen van Gelder, used at the 2005 SAT Competition¹. The format is as follows: the file starts out with a header indicating its format (`%RESA32`: a resolution file in 32 bits ASCII representation). This is followed by the number of variables (6) and clauses (16). Next is some space reserved for comments. Following that is the proof itself.

Consider the first line: `17 4 5 11 4 3 -6 -1 -5 4`. 17 is this clause's number, one higher than the highest previous clause number. Following is the clash literal, 4. Then 5 and 11 are the two clauses that are resolved upon with clash literal 4. The following sequence of numbers, `4 3 -6 -1 -5 4`, starts and ends with 4, this being the number of literals of this new clause 17. These four literals are then specified: 3, -6, -1, -5. Resolution continues until we come to the last line: `47 1 45 46 0 0`. The final clause number is 47, generated by performing resolution on clash literal 1 with clauses 45 and 46. The resolving clause has no literals (0 0) and thus the refutation is valid.

```
%RESA32
6 16
```

```
17 4 5 11 4 3 -6 -1 -5 4
18 5 17 15 4 -2 -6 3 -1 4
19 3 1 18 3 -1 -2 -6 3
20 4 9 6 4 -3 -6 -1 5 4
21 5 16 20 4 2 -6 -3 -1 4
22 3 21 2 3 -1 2 -6 3
23 4 12 6 4 3 6 -1 5 4
24 5 14 23 4 -2 6 3 -1 4
25 3 1 24 3 -1 -2 6 3
26 4 5 10 4 -3 6 -1 -5 4
27 5 26 13 4 2 6 -3 -1 4
28 3 27 2 3 -1 2 6 3
29 2 19 22 2 -1 -6 2
30 2 25 28 2 -1 6 2
31 4 9 7 4 -3 -6 1 -5 4
32 5 31 15 4 -2 -6 -3 1 4
33 3 32 4 3 1 -2 -6 3
34 4 8 11 4 3 -6 1 5 4
```

¹Allen van Gelder's proof checker and the format can be found at <http://www.satcompetition.org/2005>

35 5 16 34 4 2 -6 3 1 4
36 3 3 35 3 1 2 -6 3
37 4 8 10 4 -3 6 1 5 4
38 5 14 37 4 -2 6 -3 1 4
39 3 38 4 3 1 -2 6 3
40 4 12 7 4 3 6 1 -5 4
41 5 40 13 4 2 6 3 1 4
42 3 3 41 3 1 2 6 3
43 2 33 36 2 1 -6 2
44 2 39 42 2 1 6 2
45 6 29 30 1 -1 1
46 6 43 44 1 1 1
47 1 45 46 0 0

C Resolution proof generated by minisat

Below is the resolution proof of the instance in [A](#) generated by `minisat`. It's format uses chaining which is a more compact representation of a resolution proof. Multiple resolution steps are put together into one chain and only the final clause of the chain is assigned a new clause number. Intermediate clauses are not assigned a clause number as they are needed only once for constructing the final clause of the chain. Thus the number of clauses generated for the proof is smaller than with the proof format described in [appendix A](#).

Consider the first line for example: 16: CHAIN 9 [3] 3 [4] 7 [2] 12 => 1 5 6. Again the first number is the new clause number. `minisat` labels the clauses starting at 0 instead of `satz` that starts at 1, thus the new clause is labeled 16. The numbers without brackets are clause numbers (again starting at zero, not at one), the numbers between brackets are clash literals. Line 16 should be read as follows: clause 9 and 3 are resolved on literal 3, generating temporary clause with literals 1 -2 4 6. This clause is resolved on with clause 7 on literal 4, yielding 1 -2 5 6. This temporary clause is then resolved on with clause 12 on literal 2 and the resulting clause 1 5 6 is derived. This clause is numbered 16 and will be referenced by that number later in the proof (in chain 17 for example). The number of resolution steps performed in this chain is thus 3. The proof ends with chain 27, which deducts the empty clause by resolving on derived clauses 25 and 26. The total number of resolution steps performed is 41. For comparison: the proof generated by `satz` counts 28 resolution steps.

To convert this format to the format readable by Van Gelder's proof checker, all the resolution steps performed in a chain need to be labeled explicitly. Thus clause 16 would be the result of performing resolution on clauses 9 and 3, clause 17 would be the result of performing resolution on 16 and 7, clause 18 of resolution on 17 and 12, etc. This conversion is trivial.

```
16: CHAIN 9 [3] 3 [4] 7 [2] 12 => 1 5 6
17: CHAIN 11 [3] 2 [4] 6 [2] 13 [5] 16 => 1 6
18: CHAIN 15 [5] 7 [2] 3 [4] 10 [6] 17 => 1 3
19: CHAIN 14 [5] 6 [2] 2 [4] 8 [3] 18 [6] 17 => 1
20: CHAIN 14 [6] 11 [2] 1 [1] 19 [4] 5 [1] 19 => 3 5
21: CHAIN 12 [2] 0 [1] 19 [6] 8 [3] 20 [4] 5 [1] 19 => 5
22: CHAIN 4 [5] 21 [1] 19 => -4
23: CHAIN 9 [4] 22 [6] 15 [5] 21 [3] 1 [1] 19 => 2
24: CHAIN 0 [2] 23 [1] 19 => -3
25: CHAIN 13 [2] 23 [5] 21 => 6
26: CHAIN 10 [3] 24 [4] 22 => -6
27: CHAIN 25 [6] 26 =>
```