

Using conflict clauses in a look-ahead solver

Maarten Bezemer Joost Cassee

16 September 2008

Abstract

The satisfiability problem is often solved with either a look-ahead solver or a conflict-driven solver. In this report, we investigate a way to add conflict analysis to a basic look-ahead solver. The new solver should be able to use information from earlier conflicts for decisions and propagation during the solving process. Two adapted versions of the solver are tested on several different problem families: one using the generated conflict clauses only during normal unit propagation, the other also using the conflict clauses during the look-ahead phase. The results show that the node count can be decreased, but usually at the expense of longer run time per node. Suggestions are made to improve on the run time, and on the generation of conflict clauses.

1 Introduction

1.1 Satisfiability

The satisfiability problem is the problem of determining if it is possible to do a variable assignment for a Boolean formula, in such a way that the formula evaluates to TRUE. A formula consists of *clauses*, each containing *literals*. A literal refers to a variable x_1 or its complement $\neg x_1$. In clauses, the OR and NOT operators are used, and the formula to be evaluated to true consists of all clauses linked together with the AND operator. If there is (at least) one variable assignment for which the entire formula evaluates to TRUE, the formula is called satisfiable, or “SAT”. Otherwise, if the formula evaluates to FALSE for all possible variable assignments, the formula is called unsatisfiable, or “UNSAT”.

Formulas are in *conjunctive normal form*, for example $(x_1 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$. A formula is represented as a *list of clauses*. A clause is simply a list of literals. To convert a clause list to a formula the literals in every clause are joined using ORs and the clauses using AND. For example, the formula $(x_1 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$ is represented by the clause list $[(x_1, x_3), (x_2, -x_3, -x_4)]$.

For satisfiable formulas, it is usually enough to find only one of the possible variable assignments that causes the formula to evaluate to TRUE. For instances with

more than one solution, one of them may be found after processing only a small fraction of the search space. For unsatisfiable formulas, all possible variable assignments have to be tried before one can declare a formula “UNSAT”.

Solving a satisfiability problem is done by finding a solution for the problem or determining that there is no solution at all. Solving a satisfiability problem can take an exponential amount of time. Verifying a variable assignment which causes the formula to evaluate to TRUE can, however, be done in polynomial time. Therefore, the satisfiability problem is NP-complete.

Solving instances of the satisfiability problem can be done using the DPLL algorithm [1], named after its inventors Davis, Putnam, Logemann, and Loveland. This algorithm systematically explores the exponentially-sized search space using a backtracking algorithm. When a variable assignment for a subset of the variables is found which will cause the entire formula to evaluate to FALSE, no matter what values are assigned to the other variables, some part of the search tree can be skipped. Current SAT solvers use one of two strategies to cleverly walk through the DPLL search tree to find a solution or to determine that no such solution exists. The first is *look-ahead*, the second is *conflict-driven*. Both are discussed below. Conflict-driven solvers tend to be good at solving large (industrial) problems with large numbers of clauses and variables, but with a relatively easy instance inside. Look-ahead solvers tend to be better at solving (smaller) hard instances.

1.2 Look-ahead solvers

DPLL SAT solvers structure the search space as a decision tree. In look-ahead solvers, heuristics are used to determine an effective *branch variable*. At every step, a look-ahead procedure is used which studies the effect of assigning either TRUE or FALSE to unassigned variables. This is done by examining the effect of setting the literals related to the variables to TRUE. The goal of the look-ahead procedure is to find a variable whose literal assignments result in the largest reduction of the search space, so that a solution to the formula is found in as few steps as possible. The variable selected by the look-ahead procedure is called the *decision variable*. The solver will now *branch* on this variable by setting the related literal that lead to the largest reduction of the search space. The algorithm is now said to be *in the left branch* of the decision variable.

When assigning a value to a literal, all clauses containing that literal or its complement are updated. This can lead to *unit clauses*: clauses containing only one unassigned literal. That literal (also called a *forced literal*) is then forced to true, to ensure that the entire formula can still evaluate to true. This process is repeated until there are no more forced literals, and is called *iterative unit propagation*. A literal that is assigned is also called a *fixed literal*. Some solvers try only a subset of the unassigned literals during the look-ahead phase. Trying all unassigned literals is called doing *full look-ahead*.

During the look-ahead phase, it is possible that assigning a value to a certain literal leads to a conflict during propagation. That literal is called a *failed literal*. If for example the propagation of x_2 leads to a conflict, $\neg x_2$ is forced to true. When using *iterative look-ahead*, the look-ahead phase is repeated until no more failed literals are detected.

One of the often-used heuristics is to count the number of clauses that would be reduced to a clause with only two literals (binary clause count), since setting one of these two literals to false could in turn lead to a higher binary clause count and a larger reduction of the search space in the next step of the algorithm.

Using this algorithm, it is possible that a satisfying variable assignment is found after just a few steps. But it is also likely that after fixing some literals it becomes impossible to satisfy the formula. In that case, the algorithm uses *backtracking* to go up one step in the tree, and then chooses the other value for the lastly chosen variable. It is now said to be *in the right branch* of that variable. The solver now continues the algorithm in that sub-tree. If no solution is found there either, the algorithm backtracks one more level. When both sub-trees of the root node have been visited without finding a solution, the algorithm returns UNSAT.

Examples of look-ahead SAT solvers are *kcdfs* [2], and *march* [4].

1.3 Conflict-driven solvers

Conflict-driven SAT solvers use a different technique. Instead of using a look-ahead phase to determine the best literal to assign, they choose one based on the conflicts encountered earlier. Initially, when there are no conflicts to work with, a branch literal is chosen randomly. When propagating the chosen branch literal, this may lead to a conflict when one of the updated clauses evaluates to FALSE. This *conflicting clause* is analysed to determine which of the literals on the chosen path caused the conflict. A so-called *conflict clause* is generated, and added to the clause list, enabling the solver to learn from earlier conflicts. This information can be used in later steps of the algorithm, possibly leading to conflicts earlier, and thereby limiting the search space.

Due to the analysis of the conflicts, it may be possible to backtrack multiple levels in the search tree at once. For example, the conflict may be the result of two chosen literals earlier on in the tree. In that case, it would be pointless to keep searching in a sub tree in which it is already known that there are no solutions. This multi-level backtracking is also known as *backjumping*.

Other techniques used in conflict-driven solvers include random restarts [3]. If a variable assignment fails to result in a solution within a predefined number of steps, the solver may decide to start over, and choose a different literal to assign in the first step. Usually this literal is chosen based on the number of occurrences in conflict clauses. The conflict clauses learned before restarting are preserved, and are used

to find a solution. In [9], it is stated that restarts do not harm the algorithm's ability to find a solution. Examples of conflict-driven SAT solvers are MiniSat [5] and zChaff [6].

1.4 Outline

In this report, we investigate the effects of using conflict analysis and the thereby generated conflict clauses in a look-ahead solver, thus creating a hybrid solver. We do this by building a basic look-ahead solver, testing and debugging it, and using it to create a baseline to compare the later versions of the solver to. Then, we add conflict analysis and conflict-clause generation to the solver, and start using these conflict clauses.

The remainder of this report is as structured follows. Section 2 describes the algorithm used to generate conflict clauses, and techniques we used to optimise the storage and use of conflict clauses. Section 3 describes a few other optimisations to our solver, which are not directly related to the use or the effectiveness of conflict clauses in a look-ahead solver. Section 4 describes the test set-up and testing approach we used to generate the data to compare the results of the different versions of the solver, and discusses the results. In Section 5, we summarise our work, and draw conclusions. We also present a few topics for future research, which may lead to further improved look-ahead solvers by using conflict clauses.

2 Adding conflict clauses to a look-ahead solver

We started by porting the Pisang look-ahead solver¹ to the C programming language. This solver was chosen because it is a pure look-ahead solver which is not optimised for speed. The implementation is only two pages long.

Although non-conflict optimisations were not part of our research, we did implement some unrelated speed improvements. These optimisations are discussed in section 3 on page 10. The rest of this section describes the generation and use of conflict clauses in the look-ahead solver.

2.1 Generating and using conflict clauses

In a look-ahead solver, conflict clauses can be useful in two distinct ways. First of all, conflict clauses represent conflicting combinations of literals. Finding conflicts early in the look-ahead process could help the heuristics used to determine the value of choosing a certain decision variable. For example, usually decision

¹See appendix A

variables are ranked by the number of clauses reduced to binary by propagating the chosen literal. In that case including conflict clauses could improve the heuristic.

The second way conflict clauses can be useful is to introduce backjumping in look-ahead solvers. In a normal look-ahead solver the recursive solving algorithm backtracks when the chosen path reaches a dead end. If the last conflict generates a conflict clause which does not contain the last (or some last) chosen literals, these decision variables may be skipped during backtracking.

However, conflict clauses can also slow down the look-ahead solver since they use memory and need to be propagated.

Our research focuses on whether conflict clauses can be useful in a look-ahead solver. Important points in the effectiveness of conflict clauses are clause generation, use and management. This section describes the way we implemented conflict clauses in our simple look-ahead solver.

2.1.1 Generating conflict clauses

During the look-ahead procedure the solver may encounter a failed literal. If look-ahead is not performed on all clauses (for example, one of the variants of our solver does not use conflict clauses during look-ahead) the solver can encounter a conflict during the propagation on the chosen path. In either case, when a conflicting clause is found, the solver has to determine the cause of the conflict. There are several methods for determining the cause of the conflict. Our algorithm will consider the decision variables on the chosen path and the look-ahead variable (if the conflict is found during look-ahead), also called *last UIP* scheme [9]. The algorithm is described in Algorithm 2.1.

Each conflict clause consists of literals on the chosen path. On the other hand, probably none of the literals in the conflicting clause are on the chosen path but are forced by the propagation of fixed literals. To find the literals on the chosen path, each forced literal is associated with an *antecedent clause* consisting of all literals that forced it. The algorithm uses the antecedent clauses to find the literals on the chosen path that are responsible for the conflict. These literals form the new conflict clause.

For conflict-driven solvers, Zhang et. al. [9] suggest that conflict analysis could be improved by not iterating back to the decision variables, but by keeping the forced literals in a conflict clause. This is called the *first UIP cut*, as opposed to the *last UIP cut* we use. Whether this technique also leads to improvements for look-ahead solvers using conflict analysis is left for future research.

Algorithm 2.1 Generating a conflict clause

Require: Conflicting clause C_u , set of decided literals *decided*.

Ensure: Returns conflict clause C_c .

function GENERATE-CONFLICT-CLAUSE(C_u)

$C_c \leftarrow$ empty clause

$processed \leftarrow \emptyset$

$literals \leftarrow \emptyset$

 ADD-INVERSE-LITERALS($literals, C_u, processed$)

for all $lit \in literals$ **do**

$literals \leftarrow literals - lit$

if IS-DECIDED(lit) **then**

 ADD-LITERAL($C_c, -lit$)

else

 ADD-ANTECEDENT($literals, lit, processed$)

end if

end for

return C_c

end function

function ADD-INVERSE-LITERALS($literals, C, processed$)

for all lit in clause C **do**

if $lit \notin processed$ **then**

$literals \leftarrow literals \cup \{-lit\}$

$processed \leftarrow processed \cup \{lit, -lit\}$

end if

end for

end function

function ADD-ANTECEDENT($literals, lit, processed$)

$C_a \leftarrow$ ANTECEDENT-CLAUSE(lit)

if C_a contains only decision variables **then**

$depth \leftarrow$ FIX-DEPTH(lit)

 ADD-DECISION-VARIABLES($literals, depth, processed$)

else

 ADD-INVERSE-LITERALS($literals, C_a, processed$)

end if

end function

Algorithm 2.1 Generating a conflict clause (continued)

```
function ADD-DECISION-VARIABLES(literals, depth, processed)  
  for all lit  $\in$  decided do  
    if FIX-DEPTH(lit) < depth then  
      if lit  $\notin$  processed then  
        literals  $\leftarrow$  literals  $\cup$  {lit}  
        processed  $\leftarrow$  processed  $\cup$  {lit,  $\neg$ lit}  
      end if  
    end if  
  end for  
end function  
  
function ANTECEDENT-CLAUSE(lit)  
  return the clause that forced literal lit to be assigned  
end function  
  
function ADD-LITERAL(C, lit)  
  Add literal lit to clause C  
end function  
  
function FIX-DEPTH(lit)  
  return the depth in the search tree at which literal lit was assigned  
end function
```

2.1.2 Using conflict clauses

Conflict clauses are propagated either during look-ahead, on chosen path, or both. Propagating during look-ahead is done to enhance the heuristic, on the chosen path to reduce the decision tree. The latter effect is pretty much a given when using conflict clauses. However, we expected much less effect on the look-ahead heuristic. Look-ahead is performed so often that the extra work of propagating the conflict clauses seemed prohibitive.

Therefore we decided to run benchmarks once using conflict clauses only during decision propagation, and once also using conflict clauses during look-ahead.

2.1.3 Conflict clause management

The main downside to using conflict clauses is the extra work needed during propagation. Good conflict clause management could reduce that burden. If unchecked, the list of conflict clauses can grow significantly during the operation of the solver. In many satisfiability problems we encountered, the number of generated conflict clauses was a multiple of the number of clauses in the original problem.

The goal of good management is keeping only useful conflict clauses. We use two techniques to keep the list short. We do not generate conflict clauses that would contain all decision variables at that point in the tree, because such a clause would not be useful in reducing the decision tree. Also, we remove all conflict clauses when the decision on the root node is reversed. This approach is consistent with our focus on reducing the decision tree.

2.2 Conflict clause techniques

The previous section described the basic mechanism for generating and using conflict clauses in a look-ahead solver. We experimented with a number of other conflict clause ideas. Most of these ideas try to minimise the overhead that comes with the addition of the extra clauses. We focused mainly on keeping the list of conflict clauses small by keeping only those clauses that can still be useful to the algorithm in the unvisited part of the decision tree.

2.2.1 Conflict clauses and decision variables

Sometimes a newly generated conflict clause contains all decision variables. Such clauses are useless in the sense that it can never cause a failed literal in the rest of the search tree because every variable allocation will appear in the search tree only once. As an example consider the following situation:

- Currently decided variables are $x_1 = true$, $x_2 = true$ and $x_3 = false$.
- The algorithm has already attempted $x_3 = true$. (In other words, the position in the search tree is: left branch of x_1 , left branch of x_2 and right branch of x_3 .)
- The problem is unsatisfiable with the current variable assignment. The generated conflict clause is $(-x_1, -x_2, x_3)$.

The new conflict clause will never cause a failed literal in the search tree in the right branch of x_2 because $x_2 = false$ satisfies the clause. In the right branch of x_1 the assignment of $x_1 = false$ satisfies the clause.

Many generated conflict clauses contain all decision variables. In fact on average only 4.8% of conflict clauses do not contain all decision variables and can be considered useful (median 17%). Our algorithm only adds conflict clauses to the database if it does not contain all decision variables.

There is, however, another way conflict with only decision variables can be very useful, which is with clause subsumption.

2.2.2 Clause subsumption

As the look-ahead algorithm searches the decision tree, many conflict clauses are generated. We theorised that it could happen that a conflict clause is strictly stronger than another. For example, the clause $(x_1, -x_3)$ is strictly stronger than the clause $(x_1, -x_2, -x_3)$. Any variable assignment that falsifies the first clause will falsify the second clause. In other words, the first clause contains more information. The algorithm could remove clauses that are strictly weaker from the clause database.

To discover whether clause subsumption was important we performed analysis on the conflict clause database after running problem. Unfortunately, we found much fewer cases of subsumed conflict clauses than we expected. In fact, on average only a small percentage of conflict clauses was strictly stronger than another conflict clause. For this reason we felt that the result of subsumption on the speed of the algorithm would be negligible.

Unfortunately, this idea is flawed because of our decision to discard conflict clauses containing all decision variables (see above). It is very probable that a conflict clause with few but all decision variables will subsume many previously generated longer conflict clauses. As an example, consider that the last conflict of the left branch of the first decision variable is caused by that literal alone. The resulting conflict clause will only contain the inverse of that literal, i.e. if the variable assignment is $x_1 = true$, then the generated conflict clause would be $(-x_1)$. Although that conflict clause would not be useful in the rest of the search tree, it would subsume all conflict clauses that contain $-x_1$. Therefore, all of those clauses could be removed as well.

2.2.3 Throwing away conflict clauses

In order to benefit from the added conflict clauses the algorithm needs to balance the potential benefit of finding unsatisfiable paths in the search tree against the added burden of propagating literals for the extra clauses. Many conflict-driven solvers throw away conflict clauses that are not useful when the conflict clause database becomes too large. For example, MiniSat [5] increments a counter on all clauses that are the antecedent of a fixed literal of a conflicting clause. When the clause database reaches some limit, 50% of the clauses, namely those with the lowest counter values, are deleted from the database. Our algorithm does not contain antecedent counters, so we did not implement this idea. However, we expect large benefits from it.

One insight, related to the discussion on clause subsumption, is that most of the conflict clause database become useless after the left branch of the first decision variable has been searched. This is really only true for those clauses that contain the first decision variable, but because many conflict clauses are expected to contain

it, a useful rule-of-thumb is to throw away the whole conflict clause database. Our algorithm implements this naive mechanism and throws away the conflict clause database before searching the right branch of the first decision variable.

2.2.4 Backjumping

When a conflict clause is added that contains only decision variables higher up the search tree, large parts of the tree can be skipped quickly. To see why consider the following situation:

- Currently decided variables are $x_1 = true$, $x_2 = true$, $x_3 = false$ and $x_4 = true$.
- The position in the search tree is: left branch of x_1 , left branch of x_2 , left branch of x_3 and left branch of x_4 .
- The problem is not satisfiable with the current variable allocation. The generated conflict clause is $(-x_1, -x_2)$.

Notice that the algorithm has deduced that the conflict was caused by setting $x_1 = true$ and $x_2 = true$. Therefore the complete left branch of x_2 can be skipped and the algorithm can continue by deciding $x_1 = true$, $x_2 = false$. Although the look-ahead algorithm would find this out itself after adding the conflict clause to the database, it can use backjumping to avoid it altogether.

A backjump occurs when the last-assigned decision variable is not the antecedent of a conflicting clause. In that case the algorithm can jump back in the search tree to the last decision variable that *is* an antecedent. All intermediary variables are skipped. In the example, a backjump to level two would occur. Our algorithm implements backjumping.

In practice backjumping happens only rarely, and then only one or two levels. Additionally, because the newly generated conflict clause is falsified below the last-assigned decision variable that is an antecedent, the solver will quickly backtrack anyway. The speed gains from backjumping are therefore quite small.

3 Other optimisations

We used a couple of other optimisations to increase the speed of our algorithm. Because they were implemented in both the baseline algorithm and the conflict clause versions these optimisations did not influence the results.

The implementation of our look-ahead solver used ordering to separate assigned from unassigned literals. A small optimisation was made to the ordering of the

variables in newly created conflict clauses. When the algorithm backtracks, the literals of the clause will become unassigned in order of decreasing depth. It makes sense to order the variables of a newly created conflict clause by ascending depth, so that they will not need to be reordered when literals become unassigned during backtracking.

The conflict clause database was optimised for write-once, read-often access. Removing individual clauses from the database was not implemented, although the database can be cleared completely. This is used for throwing away all conflict clauses before searching the right branch of the first decision variable. The database is implemented as a dynamic array with $O(1)$ complexity for clause access and amortised $O(1)$ complexity for clause addition.

We used iterative unit propagation (IUP), which was not implemented in Pisang. Most real SAT-solvers implement IUP, so using IUP in our solver makes a more fair comparison to other solvers.

4 Experiments and Results

The performance of a look-ahead solver can be measured in two different ways. The easiest measure is the time it takes to solve a problem. Unfortunately this measure is dependent on the (hardware and software) environment the solver runs in. Furthermore, most solvers are heavily optimised in ways not related to their algorithmic basis (look-ahead or conflict-driven).

A better measure, which is specific to look-ahead solvers, is the number of nodes in the decision tree, the *node-count*. Remember that look-ahead solvers work by investing time in deciding the right choice of decision variables. Using conflict clauses means even more work for the solver. Therefore, the node-count must go down if conflict clauses are to have a positive effect.

4.1 Benchmarks

To put our ideas and implementation to the test, we ran benchmarks on a number of different problems sets from the Satisfiability Library [8] and the International SAT Competitions web page [7]. We first took a generic approach to test a lot of different problem families. Not all families could be solved by our solver within an acceptable amount of time, and using an acceptable amount of memory. To be able to present a comprehensive set of results, we selected a number of families, for which instances did run on all variants of our solver within acceptable bounds. We did not include families such as Pigeonhole, where conflict analysis obviously would not be helpful in solving the instance faster.

4.2 Comparing different solvers

To compare the effects of adding and using conflict clauses, we ran the test sets with three different versions of the solver: plain look-ahead, look-ahead using the generated conflict clauses only when propagating on the chosen path, and look-ahead using the generated conflict clauses both during look-ahead and on the chosen path. As a baseline we disabled the generation and use of conflict clauses in our solver. We then tested two versions of the conflict-enhanced look-ahead solver, because using conflict clauses during propagation in the look-ahead phase may take a lot of extra time when the conflict clause database has grown large. By disabling the use of conflict clauses during the look-ahead phase, we could also study the effect of using conflict clauses on solving the instances, without changing the branch heuristics. When using the generated conflict clauses also during the look-ahead phase, the branch heuristics are influenced by the extra clauses, which may or may not be helpful.

4.3 SAT@Home

Because running the different problem sets on all versions of the solver took a long time, we created a distributed version of the solver set-up. Computing nodes query the central database for a problem instance to solve. The central database selects an instance that has not been solved yet, and assigns it to that computing node. When a computing node does not report back the results within a reasonable amount of time, the central database assumes that the node is lost, and may re-assign its instance to another computing node.

The compute nodes we used for this experiment all had comparable hardware specifications, enabling us to compare not only node count but also run time of each instance. Hardware specifications: Intel Pentium4 1.6GHz with 128MB RAM, running a recent 2.6 version of the Linux kernel.

4.4 Results

The results for all tested instances are accumulated in the database, and summarised in Table 1. The rows show the different families of problems, and the column shows the average node count and run time for the given family, for all three versions of the solver. For the conflict-enhanced solvers, a percentage is shown for both node count and run time, which shows the values relative to the plain look-ahead solver. The results listed in Table 1 only show the node counts and run times for instances that all variants of the solver were able to solve. For a number of instances, the conflict-enhanced solvers were not able to succeed in solving these instances within the allotted time frame, whereas the plain-look-ahead solver did solve them. Most of the time, this was due to a rapidly growing clause database, requiring more work

during propagation, and more time spent in memory management for every step in the algorithm. The values listed for node count and run time are an average of all (completed) instances of the given family.

4.5 Discussion

Even with the optimisation described in the previous sections, our algorithm performed poorly. We see that generating and using conflict clauses can reduce the node count on the given problem sets. Using conflict clauses during both look-ahead and during normal propagation can further improve the node count, but improvements are usually not as big as from plain look-ahead to using conflict clauses only on the chosen path. On the other hand, we also observe that the reduced node count does not imply a shorter run time. In fact, the extra work required to generate and process conflict clauses usually takes so much time, that the total run time is longer than with the plain look-ahead solver.

We also see that, although the overall trend is to using more extra time per decision node than the drop in node count can compensate for, some families actually may gain from conflict analysis, for example Goldberg/bmc1 and Anton/l4k3. For the Bierre/cmpadd family, both the node count and the run time have very small improvements. An example for which conflict analysis is most definitely not an improvement, is the Pehoushek/graphcolors4 family.

5 Conclusions and Future Work

In this report we have presented our work on generating and using conflict clauses in a look-ahead solver. We first created a basic look-ahead solver, to be able to compare the results of generating and using conflict clauses to those of a plain look-ahead solver using the same heuristics but no conflict clauses. Later, we added code to do conflict-analysis, and use the generated conflict clauses during the solving process. Two variants of the conflict-enhanced solver are tested: one using the conflict clauses only during propagation on the chosen path, and the other using the conflict clauses also during the look-ahead phase.

We tested these solvers against a number of problem sets, and presented the results. For some types of problems, generating and using conflict clauses only marginally improved the node count or not at all. For other types of problems, using the conflict clauses resulted in a lower node count, sometimes even significantly lower. The improved node count did, however, not result in shorter run times for most problems.

The main conclusions of our work are that conflict-analysis and the use of the generated conflict clauses in look-ahead solvers may help in reducing the node

count, but comes at the expense of longer run times. This can be explained by the extra work needed to do conflict analysis, the extra work needed to do propagation on the added conflict clauses, and the added overhead of the memory management needed to store the generated conflict clauses.

Future work on adding conflict analysis to look-ahead solvers could focus on finding better strategies to store conflict clauses, and dropping conflict clauses when they are no longer useful. Another suggestion is related to the conflict clause generation. As discussed in Section 2.1.1, we use only decision variables in the generated conflict clauses. This is also known as using the last UIP cut. However, according to [9] it may be better to use the first UIP cut.

Problem	No conflict clauses		Conflict clauses on chosen path		Conflict clauses also during look-ahead					
	NodeCount	Time (secs)	NodeCount	Time (secs)	NodeCount	Time (secs)				
<i>aloul/Bart</i>	19196.2n	2.993s	11986.8n	62.4%	452.388s	15114.9%	11893.4n	62.0%	479.099s	16007.3%
<i>anton/l3k3</i>	6133.0n	308.027s	6090.0n	99.3%	264.672s	85.9%	6058.2n	98.8%	230.274s	74.8%
<i>anton/l4k3</i>	124433.0n	1760.428s	9493.0n	7.6%	653.390s	37.1%	3354.5n	2.7%	146.304s	8.3%
<i>biere/cmpadd</i>	1522.3n	0.261s	1519.0n	99.8%	0.251s	96.2%	1472.3n	96.7%	0.254s	97.3%
<i>chu-min-li/urquhart</i>	1052671.0n	82.412s	1052671.0n	100.0%	88.230s	107.1%	1052671.0n	100.0%	87.657s	106.4%
<i>goldberg/bmc1</i>	1049599.0n	864.785s	574465.0n	54.7%	460.125s	53.2%	432.2n	0.0%	0.408s	0.0%
<i>goldberg/bmc2</i>	2888.0n	12.260s	1492.0n	51.7%	6.939s	56.6%	463.0n	16.0%	1.700s	13.9%
<i>goldberg/hanoi</i>	3474.0n	16.000s	3268.0n	94.1%	18.000s	112.5%	1821.0n	52.4%	12.000s	75.0%
<i>hirsch/hgen8</i>	42040.3n	9.807s	34458.3n	82.0%	2388.571s	24356.6%	25596.0n	60.9%	1756.233s	17908.6%
<i>jarvisal05/moad2-rand3bip-sat</i>	3638762.8n	1351.053s	3636359.5n	99.9%	1421.882s	105.2%	3636305.5n	99.9%	1422.842s	105.3%
<i>kukuta/tdadm-bench</i>	45889.0n	16.079s	42478.0n	92.6%	20.165s	125.4%	40846.0n	89.0%	19.884s	123.7%
<i>maris05/DriverLog</i>	16.3n	1.637s	16.3n	100.0%	1.656s	101.2%	16.3n	100.0%	1.641s	100.2%
<i>maris05/Ferry</i>	12823.8n	102.515s	10841.5n	84.5%	687.821s	670.9%	9099.5n	71.0%	587.861s	573.4%
<i>moore/hidden</i>	32629.3n	93.082s	32552.6n	99.8%	300.378s	322.7%	33363.1n	102.2%	401.667s	431.5%
<i>pehoushek/graphcolors3</i>	11619.3n	18.421s	8239.1n	70.9%	244.612s	1327.9%	2794.9n	24.1%	55.602s	301.8%
<i>pehoushek/graphcolors4</i>	120278.8n	130.280s	115330.1n	95.9%	13110.908s	10063.6%	77180.0n	64.2%	9779.936s	7506.8%
<i>preswitch/mediator</i>	4077.0n	8.194s	2877.0n	70.6%	21.052s	256.9%	2854.5n	70.0%	24.901s	303.9%
<i>ricci-terenghi/glassy-sat-sel</i>	4763.0n	2.820s	4755.0n	99.8%	3.412s	121.0%	4695.0n	98.6%	3.724s	132.1%
<i>sabharwal05/counting/fp/hp/unsat/easier</i>	22771.0n	3.507s	22744.0n	99.9%	74.958s	2137.4%	22795.0n	100.1%	56.464s	1610.0%
<i>vangelder/GridM/bench</i>	15783.0n	0.344s	10153.0n	64.3%	0.948s	275.6%	8833.0n	56.0%	1.060s	308.1%
<i>vangelder/Rope/Bench</i>	33694241.0n	2780.092s	33694241.0n	100.0%	3554.883s	127.9%	33388408.0n	99.1%	3499.312s	125.9%
<i>zhang-hantao/qgbench</i>	9120.1n	431.291s	1896.2n	20.8%	832.325s	193.0%	1760.9n	19.3%	599.170s	138.9%
<i>zhang-lintao/xor-chain</i>	8421375.0n	228.913s	8421375.0n	100.0%	255.523s	111.6%	8421375.0n	100.0%	256.228s	111.9%

Table 1: SAT@Home results for three variants of our solver.

References

- [1] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
- [2] Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *IJCAI*, pages 248–253, 2001.
- [3] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 431–437, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.
- [4] Marijn J. H. Heule, Joris E. van Zwieten, Mark Dufour, and Hans van Maaren. March_eq: Implementing additional reasoning into an efficient lookahead sat solver. In Holger H. Hoos and David G. Mitchell, editors, *SAT 2004*, volume 3542 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2005.
- [5] MiniSat - A minimalistic, open-source sat solver. <http://minisat.se/>.
- [6] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535, New York, NY, USA, 2001. ACM.
- [7] The International SAT Competitions web page. <http://www.satcompetition.org/>.
- [8] SATLIB - The Satisfiability Library. <http://www.satlib.org/>.
- [9] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. *iccad*, 00:279, 2001.

A The Pisang look-ahead solver

```
#!/usr/bin/env python
from sys import exit, argv

cnf = [l.strip().split() for l in file(argv[1]) if l[0] not in 'c%0\n']
clauses = [[int(x) for x in m[:-1]] for m in cnf if m[0] != 'p']
nrofvars = [int(n[2]) for n in cnf if n[0] == 'p'][0]
vars = range(nrofvars+1)
occurrence = [[] for l in vars+range(-nrofvars,0)]
for clause in clauses:
    for lit in clause: occurrence[lit].append(clause)
fixedt = [-1 for var in vars]

def solve_rec():
    global nodecount
    nodecount += 1
    if not -1 in fixedt[1:]:
        print 's SATISFIABLE'
        print 'v', ' '.join([str((2*fixedt[i]-1)*i) for i in vars[1:]]), ' 0'
        return 1

    la_mods = []
    var = lookahead(la_mods)
    #print 'select', var
    if not var: return backtrack(la_mods)

    for choice in [var, -var]:
        prop_mods = []
        if propagate(choice, prop_mods) and solve_rec(): return 1
        backtrack(prop_mods)

    return backtrack(la_mods)

def propagate(lit, mods):
    global bincount

    current = len(mods)
    mods.append(lit)

    while 1:
        if fixedt[abs(lit)] == -1:
            fixedt[abs(lit)] = (lit > 0)
            for clause in occurrence[-lit]:
                length, unfixed = info(clause)

                if length == 0: return 0
                elif length == 1: mods.append(unfixed)
                elif length == 2: bincount += 1

        elif fixedt[abs(lit)] != (lit > 0): return 0
```

```

        current += 1
        if current == len(mods): break
        lit = mods[current]

    return 1

def lookahead(mods):
    global bincount

    dif = [-1 for var in vars]
    for var in unfixed_vars():
        score = []
        for choice in [var, -var]:
            prop_mods = []
            bincount = 0
            prop = propagate(choice, prop_mods)

            backtrack(prop_mods)
            if not prop:
                if not propagate(-choice, mods): return 0
                break
            score.append(bincount)
        dif[var] = reduce(lambda x, y: 1024*x*y+x+y, score, 0)

    return dif.index(max(dif))

def backtrack(mods):
    for lit in mods: fixedt[abs(lit)] = -1
    return 0

def info(clause):
    len, unfixed = 0, 0
    for lit in clause:
        if fixedt[abs(lit)] == -1: unfixed, len = lit, len+1
        elif fixedt[abs(lit)] == (lit > 0): return -1, 0
    return len, unfixed

def unfixed_vars(): return [var for var in vars[1:] if fixedt[var] == -1]

nodecount = 0
if not solve_rec():
    print 's UNSATISFIABLE', nodecount

```