# The first steps to a hybrid SAT Solver

**Dimosthenis Mpekas** D.Mpekas@student.tudelft.nl
**Michiel van Vlaardingen** michiel@2vi.nl
**Siert Wieringa** S.Wieringa@student.tudelft.nl

*Department of Software Technology,*
*Faculty of Electrical Engineering, Mathematics and Computer Sciences,*
*Delft University of Technology*
*October 5, 2006*

## Abstract

Two dominant complete architectures exist for SAT solvers: The look-ahead and the conflict-driven architecture. This paper describes our effort to merge parts of the look-ahead architecture with a conflict-driven solver (`MiniSat`) in order to create a hybrid solver that exploits the advantages of both architectures. Our contribution lies at implementing and comparing different strategies of doing look-ahead in a conflict-driven solver. Experimental results on randomly created formulas are presented along with a behavior analysis. The main conclusion is that branching on the largest reduction quality literal seems to outperform the always branch negative heuristic used by `MiniSat` and the smallest reduction quality heuristic that is optimal for the look-ahead solver `March`.

## 1. Introduction

Given a Boolean formula $\mathcal{F}$, a SAT solver determines whether an assignment of variables exists so that the formula evaluates to true. The input formula of a SAT solver is usually expressed in Conjunctive Normal Form (CNF). Thus, $\mathcal{F}$ is a conjunction of clauses. In turn, each clause is a disjunction of literals which are atomic Boolean variables $x_i$ or its negated form $\neg x_i$. A clause $c$ is satisfied if at least one of its literals has been assigned the value **true**. Finally, Boolean formula $\mathcal{F}$ is satisfiable if there exists an assignment to the Boolean variables $x_i$, in such a way that all clauses in $\mathcal{F}$ are satisfied. Otherwise, $\mathcal{F}$ is called unsatisfiable.

There exist two main architectures for SAT solvers, the look-ahead and the conflict-driven. Both approaches to solve the SAT problem have particular strengths and weaknesses on certain types of problems. Some problems that can be easily solved with a look-ahead solver, such as randomly created formulas, take far much more time or are even impossible to be solved using a conflict-driven solver. For other problems, the opposite is true.

In this paper, a hybrid solver that exploits the advantages of both look-ahead and conflict-driven architectures is introduced. Basically, our effort focuses on merging the look-ahead procedure with the conflict-driven solver `MiniSat`. Our contribution lies at implementing and comparing different strategies of doing look-ahead in a conflict-driven solver along with some optimizing heuristics. The importance of such an implementation lies at the case in which the set of formulas tested may belong to various different families. Thus, adopting a look-ahead or a conflict-driven approach is not the right decision since these approaches meet difficulties at running problems out of their range. In such cases, the hybrid solver is meant to find a solution with average computational cost.

The paper is structured as follows. The two SAT solver architectures will be discussed in section 2. The analysis of our hybrid solver along with the analysis of the heuristics for selecting the branching variable will be discussed in section 3. In section 4, the experimental results are analyzed

by comparing the different versions implemented. The experiments conducted focus exclusively on randomly created formulas with a varying number of variables. Finally, section 5 draws the conclusion and also presents some indicative future work.

## 2. SAT Solvers

A powerful framework to solve a SAT problem is Davis-Putman-Logemann-Loveland (DPLL) (Davis, Putman, Logemann and Loveland (1962)).

Solvers based on the DPLL framework use Iterative Unit Propagation (IUP) and recursive function calls to determine a formula's satisfiability. IUP is the process of propagating unit literals, that is if all but one of the literals of a clause have been assigned to false, then the remaining literal must be assigned to true in order to satisfy this clause, called unit clause. After this propagation, new unit clauses might occur and the aforementioned process will be iteratively executed. The pseudo-code of the DPLL framework is shown in algorithm 1.

The DPLL procedure depends on the choice of an effective branching variable which in respect to the other variables, causes a relatively large reduction if one would fix this variable and simplify the formula.

**Algorithm 1** DPLL($\mathcal{F}$)

```
1: F:= IterativeUnitPropagation(F)
2: if F = ∅ then
3:    return "satisfiable"
4: else if F contains empty clause then
5:    return "unsatisfiable"
6: end if
7: x := GetBranchVariable()
8: if DPLL(F ∪ {x}) = "satisfiable" then
9:    return "satisfiable"
10: else
11:    return DPLL(F ∪ {¬x})
12: end if
```

### 2.1 Look-Ahead Solvers

Look-ahead SAT solvers (LAS) usually consist of a DPLL framework and a sophisticated look-ahead procedure to determine an effective branch variable. The look-ahead procedure measures the effectiveness of branching on certain variables by performing look-ahead on a set of variables and evaluating the reduction of the formula. We refer to the look-ahead on literal $x$ as the Iterative Unit Propagation on the union of a formula with the unit clause $x$. The look-ahead is evaluated only when propagating both the positive and negative values of a variable do not result in a conflict. The effectiveness of a variable $x_i$ is obtained using a look-ahead evaluation function (in short Diff), which evaluates the differences between $\mathcal{F}$ and the reduced formulas as shown in lines 3 and 4 of algorithm 2. The Diff function can take into account entities such as the number of clause reductions, the number of satisfied clauses and the reduction of free variables. A very commonly used Diff function, proposed by Li and Anbulagan (1995), is the number of newly created binary clauses. Smaller clauses in the resulting formulas are an indicator for stronger constraints.

The selection of the branch variable is performed using the heuristic MixDiff presented at line 12 of algorithm 2 and it usually multiplies both Diff's of a specific variable and returns the one with the highest product. There are other equally well heuristics though.

Literals that result in a conflict when propagated are called failed literals. Finding a failed literal forces the assignment of the variable to the opposite value. Detection of failed literals can result in a substantial reduction of the DPLL search space. Once a failed literal is found the look-ahead procedure has to restart because the new, forced, variable assignment has voided the previously calculated reduction qualities. If failed literals occur at the beginning of the look-ahead procedure the number of variables that need to be propagated again is kept small. Therefore, the variables that are most likely to fail should be sorted first in order to detect the failed literals early.

Examples of fast look-ahead solvers are `satz` (Li and Anbulagan (1995)), `OKsolver` (Kullmann (2002)), `March` (Heule, van Zwieten, Dufour and van Maaren (2005)) and `kcnfs` (Dubois and Dequen (2001)).

**Algorithm 2** LOOK-AHEAD ($\mathcal{F}$)

```
1: repeat
2:   for all free variables x_i in the sorted set F do
3:     F' := IterativeUnitPropagation(F ∪ {x_i})
4:     F" := IterativeUnitPropagation(F ∪ {¬x_i})
5:     if F' contains empty clause and F" contains empty clause then
6:       return "unsatisfiable"
7:     else if F' contains empty clause then
8:       F:= F"
9:     else if F" contains empty clause then
10:      F:= F'
11:    else
12:      H(x_i) := MixDiff(Diff(F, F') , Diff(F, F"))
13:    end if
14:  end for
15: until NoNewFailedLiteralsDetected( )
16: return x_i with highest H(x_i)
```

### 2.2 Conflict-Driven Solvers

A conflict-driven solver (CDS) differs a bit from the standard DPLL framework. The approach is based on the observation that changing the last assigned variable often does not resolve the conflict. Where the standard framework backtracks only one level, the CDS backtrack many levels by analyzing conflicts (line 6, 10 of algorithm 3). This procedure determines the depth of the search-tree to which one should backtrack in order to resolve the conflict and thereby avoids a lot of unnecessary computation. The given formula is unsatisfiable if a CDS backtracks beyond the root of the tree. When a conflict is encountered, the algorithm determines not only the level to which it has to backtrack, but it also adds conflict clauses into a database in order to avoid branching to the same conflict. Conflict-driven solvers are usually very fast and efficient. Some popular implementations using this architecture are `zchaff` (Moskewicz, Madigan, Zhao, Zhang and Malik (2001)), `berkmin` (Goldberg and Novikov (2002)) and `MiniSat` (Een and Sorenssen (2004)).

**Algorithm 3** ConflictDriven ($\mathcal{F}$)

```
1: while TRUE do
2:   l_i := GetBranchLiteral()
3:   if an l_i is selected then
4:     F := IterativeUnitPropagation(F ∪ {l_i})
5:     while F contains empty clause do
6:       blevel := AnalyzeConflicts()
7:       if blevel = 0 then
8:         return "unsatisfiable"
9:       else
10:        BackTrack(blevel)
11:        F := IterativeUnitPropagation(F)
12:      end if
13:    end while
14:  else
15:    return "satisfiable"
16:  end if
17: end while
```

### 2.3 Framework Analysis

Both architectures, discussed in this section, have particular strengths and weaknesses on certain types of problems. Some problems that can be easily solved with a look-ahead solver, such as randomly created formulas, take far much more time or are even impossible to be solved using a conflict-driven solver. For other problems, the opposite is true.

Compared to a CDS, a LAS makes better variable decisions which come at a higher computational cost. The CDS will do smarter backtracking but will require conflict analyzing and database management.

## 3. Hybrid Architecture

Our goal is to investigate the possibilities of a hybrid solver, which combines both look-ahead and conflict-driven techniques. As one can understand, the final goal is to build a hybrid framework that will perform relatively well for a broad set of problems.

The hybrid solver implemented in the scope of this paper is based on the `MiniSat` conflict-driven solver extended with a look-ahead procedure. `MiniSat` solver was chosen because its implementation facilitates the extension and addition of various approaches and heuristics. Moreover, `MiniSat` is a very efficient and fast solver.

### 3.1 The Look-Ahead Procedure

The only difference between our implementation of a hybrid solver and a CDS is the replacement of the GetBranchLiteral procedure in the CDS with the LookAheadSelectLiteral procedure of the hybrid approach. The later, uses mostly existing `MiniSat` data structures to minimize the added complexity.

Since a conflict-driven solver will make more branching decisions, relative low cost was an important design criterium for the look-ahead procedure.

The full look-ahead procedure, as described in section 2.1, is expensive because it performs a look-ahead on all variables. To reduce computational costs, the look-ahead can be performed on a limited set of variables, this is called partial look-ahead. Full look-ahead may be more appropriate in

detecting failed literals and making better decisions but is very expensive. On the other hand, partial look-ahead might be faster but have a slightly higher number of decisions. Partial look-ahead on the appropriate variables might balance the detection of the failed literals with the timing performance. In our implementation, the partial look-ahead is 10% of the most active unassigned variables. Thus, the partial look-ahead gives priority and considers first the variables that participate in the highest number of conflicts.

Another optimizing feature of the propagation procedure in the hybrid look-ahead approach is to first consider the variables that become free due to the backtracking. At the next look-ahead procedure, these variables are considered first. As a further optimization of this technique, a threshold has been introduced that allows not all freed variables to be propagated first, but only those whose activity is within the same range of the activities of the variables originally selected for look-ahead propagation. The above mentioned heuristic is considered as a fixed feature of the solver.

The look-ahead procedure added to `MiniSat`'s code is fairly different from the one presented in algorithm 2 in section 2.1. The pseudo-code of the updated look-ahead procedure is presented in algorithm 4.

**Algorithm 4** LookAheadSelectLiteral ()

```
1: repeat
2:   for all free variables x_i in the sorted set F do
3:     F' := IterativeUnitPropagation(F ∪ {x_i})
4:     if F' contains empty clause then
5:       F:= HandleConflict(F')
6:       continue
7:     end if
8:     F" := IterativeUnitPropagation(F ∪ {¬x_i})
9:     if F" contains empty clause then
10:      F:= HandleConflict(F")
11:      continue
12:    end if
13:    H(x_i) := MixDiff(Diff(F, F') , Diff(F, F"))
14:  end for
15: until NoNewFailedLiteralsDetected( )
16: return x_i with highest H(x_i)
```

Whenever the algorithm finds a failed literal, it calls HandleConflict. The later handles the conflict by backtracking to a level found by a call to AnalyzeConflicts as seen in algorithm 5. In case HandleConflict finds that the formula is unsatisfiable, algorithm 4 terminates.

**Algorithm 5** HandleConflict ($\mathcal{F}$)

```
1: blevel := AnalyzeConflicts()
2: if blevel = 0 then
3:   return "unsatisfiable"
4: else
5:   BackTrack(blevel)
6:   return IterativeUnitPropagation(F)
7: end if
```

### 3.2 Variable Decision Heuristics

The original `MiniSat` solver uses a dynamic variable order heuristic based on activity, which gives priority to variables involved in most recent conflicts. Each variable has an activity attached to it. Every time a variable occurs in a recorded conflict clause, its activity is increased which is referred to as bumping. After recording the conflict, the activities of variables are decaying over time, thus, recent increments count more than old. When a new decision needs to be made, `MiniSat` chooses the variable with the highest activity. It then assigns this variable to false and propagates that.

The hybrid solver defines a look-ahead based variable decision heuristic. For a variable, propagated as part of the look-ahead, the variable reduction quality must be measured. This variable reduction quality measure is calculated as follows from the literal quality measures:

$$H(y) = \text{MixDiff}(\text{Diff}(\mathcal{F}, \mathcal{F}\text{'}) \text{ , } \text{Diff}(\mathcal{F}, \mathcal{F}\text{"})) = rq(y) * rq(\neg y) + rq(y) + rq(\neg y)$$

The first heuristic implemented uses the following definition for the literal quality:

$$rq(y) = |IUP(y)|$$

IUP(y) indicates the set of all literals fixed during the unit propagation process by propagating literal $y$. From now on, this heuristic will be referred as **propagations based heuristic**.

The **ternary clauses heuristic** extends the propagations heuristic by also taking into account weights determined by the number of occurrences of the literal in ternary clauses. The branch quality of propagating literal $y$ is equal to the sum of the weights of the literals in IUP(y). The weight of each one of such literals equals the number of occurrences of the negated literal in all ternary clauses (3 unassigned variables in a clause). The literal quality for this heuristic is defined as:

$$rq(y) = \sum_{x_i \in IUP(y)} 1 + w(x_i)$$

$$w(x_i) = \#occ3(\neg x_i)$$

The weights are calculated at every restart of the look-ahead procedure but will, due to HandleConflict, not always be up-to-date. This measure of literal quality will approximate the number of newly created binary clauses when propagating $y$. The expected advantage of this heuristic is that the solver looks one step beyond the direct reduction caused by propagating $y$. The downside of the ternary clauses heuristic is that compared to propagations based heuristic, it is more expensive due to the computational overhead of calculating the weights.

### 3.3 Branching Direction Heuristics

As discussed in section 3.2, `MiniSat`'s GetBranchLiteral always returns the negated literal of the variable with the highest activity. According to one of `MiniSat`'s developers, Niklas Een, the fact that `MiniSat` always propagates the negative literal associated with the selected branch variable is related to the fact that in most structured benchmarks variables occur negated more frequently.

In a typical look-ahead solver, one would return the literal with the smallest reduction quality, since this has the smallest probability of being unsatisfiable and thus has the highest probability of containing a solution. For a normal look-ahead solver, if the formula is unsatisfiable both the positive and negative branches have to be searched and the costs remains equal regardless of the order in which this is done. However, this does not hold in our hybrid solver since due to conflict clauses learned, a different order could mean different costs, even in the unsatisfiable case. Thus, it may be better to choose the literal with the highest reduction quality since it might yield new conflict clauses faster.

## 4. Experimental Results

As seen in table 1, a version may use full, partial or no look ahead. It also may or may not propagate the more recent freed variables first. Moreover, a version may adopt the activity, the propagations or the ternary clauses based heuristic. Finally, a version besides the largest (HRQ) and smallest (SRQ) reduction quality heuristic may also choose to always propagate the negative or the positive literal or even a random choice between them.

The original `MiniSat` solver, version 1.14, compiled as statically linked release (*minisat_static*) is abbreviated as **o** here. As someone can easily notice, there is no reason to apply the heuristic of propagating the freed variables first at the full look-ahead procedure. Doing the freed variables first, makes no sense, since the full look-ahead will consider these variables in any case. Moreover, the order of the variables propagated is of minimum importance since all of them are considered.

In the rest of the tables, there are some statistics presented for every family of benchmarks. The best % statistic indicates the percentage of formulas in the benchmark for which this version performs best. The min, max and standard deviation statistics are presented only for better statistics coverage. AvgDev indicates for each version, the average value of the difference (in time or decisions) of this version from the average value of all versions for all files. The conclusions about the performance of a version can be drawn from the average and average deviation values since these values indicate which version performs better over all the test files and over all the different versions. As you might notice, when a version has the best avg, has also the best avgdev value. Thus, in the following conclusions and plots, the various versions will be evaluated and compared on the basis of the average value.

Finally, the reader should know that all computation times in this report are in seconds. Two different machines have been used, one for the results in table 2 and one for all other presented results.

**Table 1.** Abbreviations of the different versions produced by the various axes and parameters

| | Look Ahead Procedure | | | Ordered | Var Decisions Heuristic | | | Branching Literal Heuristic | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | None | Full | Partial | Freed | Act | Props | Ternary | Neg | Pos | Rand | HRQ | SRQ |
| **o** | • | | | | • | | | • | | | | |
| **fo** | | • | | | • | | | • | | | | |
| **fp$^-$** | | • | | | | • | | • | | | | |
| **fp$^+$** | | • | | | | • | | | • | | | |
| **fp$^>$** | | • | | | | • | | | | | • | |
| **fp$^=$** | | • | | | | • | | | | • | | |
| **fp$^<$** | | • | | | | • | | | | | | • |
| **pfo** | | | • | • | • | | | • | | | | |
| **pfp$^-$** | | | • | • | | • | | • | | | | |
| **pfp$^+$** | | | • | • | | • | | | • | | | |
| **pfp$^>$** | | | • | • | | • | | | | | • | |
| **pfp$^=$** | | | • | • | | • | | | | • | | |
| **pfp$^<$** | | | • | • | | • | | | | | | • |
| **pft$^-$** | | | • | • | | | • | • | | | | |
| **pft$^+$** | | | • | • | | | • | | • | | | |
| **pft$^>$** | | | • | • | | | • | | | | • | |
| **pft$^=$** | | | • | • | | | • | | | • | | |
| **pft$^<$** | | | • | • | | | • | | | | | • |

### 4.1 1-Solution Formulas

**Table 2. 1-Solution**: Statistics of 30 test files with 1 satisfying partial assignment

| | Execution Time | | | | | | Decisions Number | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Best % | Min | Max | Avg | StdDev | AvgDev | Best % | Min | Max | Avg | StdDev | AvgDev |
| **o** | 3.333 | 0.750 | 137.734 | 46.559 | 37.759 | 5.746 | 0 | 20835 | 2085399 | 790633 | 557968 | 718932 |
| **fo** | 0 | 3.421 | 450.734 | 135.008 | 100.961 | 94.196 | 0 | 2171 | 137409 | 48467 | 31013 | -23233 |
| **fp$^-$** | 0 | 5.984 | 149.234 | 46.132 | 30.324 | 5.320 | 3.333 | 3515 | 56878 | 20926 | **11513** | -50775 |
| **fp$^+$** | 0 | 2.593 | 106.937 | 48.946 | 32.097 | 8.133 | 6.667 | 1737 | **43491** | 21921 | 12673 | -49780 |
| **fp$^>$** | 0 | 4.296 | 282.500 | 58.005 | 53.681 | 17.193 | 3.333 | 2694 | 98476 | 25726 | 19227 | -45974 |
| **fp$^=$** | 6.667 | **0.125** | 194.703 | 53.652 | 53.276 | 12.839 | **16.667** | **123** | 69535 | 22791 | 19587 | -48910 |
| **fp$^<$** | 3.333 | 0.234 | 263.937 | 78.124 | 64.530 | 37.311 | 6.667 | 213 | 86433 | 30981 | 21770 | -40720 |
| **pfo** | 0 | 12.859 | 243.656 | 85.315 | 57.072 | 44.502 | 0 | 16885 | 227149 | 91902 | 52623 | 20202 |
| **pfp$^-$** | 0 | 0.609 | **47.406** | **15.507** | 11.687 | **-25.305** | 3.333 | 1224 | 59643 | 22492 | 15063 | -49208 |
| **pfp$^+$** | 3.333 | 1.062 | 55.656 | 17.889 | 14.100 | -22.924 | 3.333 | 2064 | 71327 | 25509 | 17723 | -46191 |
| **pfp$^>$** | 3.333 | 2.140 | 44.406 | 16.864 | **11.257** | -23.949 | 0 | 3781 | 54391 | 23615 | 14136 | -48085 |
| **pfp$^=$** | 10 | 0.515 | 51.187 | 15.938 | 13.721 | -24.875 | 3.333 | 1065 | 65953 | 22781 | 17567 | -48920 |
| **pfp$^<$** | 6.667 | 1.515 | 70.687 | 19.345 | 14.668 | -21.467 | 6.667 | 3060 | 84574 | 27866 | 18245 | -43835 |
| **pft$^-$** | 10 | 0.718 | 51.953 | 16.566 | 13.769 | -24.246 | 6.667 | 1168 | 55797 | **20220** | 15257 | **-51481** |
| **pft$^+$** | 3.333 | 2.890 | 68.750 | 19.199 | 14.022 | -21.614 | 0 | 4327 | 74725 | 23002 | 15151 | -48698 |
| **pft$^>$** | 16.667 | 0.843 | 90.859 | 18.762 | 20.431 | -22.051 | **16.667** | 1314 | 87041 | 21438 | 20771 | -50263 |
| **pft$^=$** | **20** | 0.234 | 74.171 | 18.989 | 17.296 | -21.824 | **16.667** | 415 | 79765 | 22408 | 18611 | -49293 |
| **pft$^<$** | 13.333 | 1.843 | 83.968 | 23.829 | 20.046 | -16.984 | 6.667 | 2898 | 86234 | 27932 | 21078 | -43769 |

#### 4.1.1 Omitted Versions

In table 2, the results of the execution of 30 randomly created test files with 300 variables and one partial assignment are presented. As one can easily notice, the versions that use the activity based heuristic (**fo**, **pfo**), perform rather poorly and far worse than the average value of all versions for both execution time and number of decisions. This shows that the gain of adding a look-ahead procedure to a conflict-driven solver is not only caused by a side effect, finding new conflict clauses faster, but that it really puts smarter decisions to a use. Due to the poor performance, these versions are excluded from further testing.

It is also noticeable that versions with a full look-ahead perform rather poor in comparison to those with partial, regarding execution time. On the other hand, the full look-ahead versions perform well regarding number of decisions. However, given that (1) full look-ahead is always going to perform poor regarding execution time and (2) do not use the freed variables heuristic which is a default feature of the hybrid solver, as mentioned in section 3.1, these versions will be omitted.

Judging from the avg and avgdev values, we have decided to include for further testing all versions using partial look-ahead and propagations based or ternary clauses heuristic (**pfp$^-$**, **pfp$^+$**, **pfp$^=$**, **pfp$^>$**, **pfp$^<$**, **pft$^-$**, **pft$^+$**, **pft$^=$**, **pft$^>$**, **pft$^<$**) along with the original `MiniSat` version.

#### 4.1.2 Performance Analysis

- Regarding both execution time and number of decisions, the versions that propagate the negative literal of a variable ($^-$) perform always better than the rest of the versions, regardless of the selection heuristic followed (propagations or ternary clauses based).

- Figure 1 indicates the large timing gain induced by the partial look-ahead approach in comparison to the versions that use full look-ahead.

- Propagating the smallest reduction quality literal yields the worst performance among the partial look-ahead versions regarding both execution time and number of decisions.

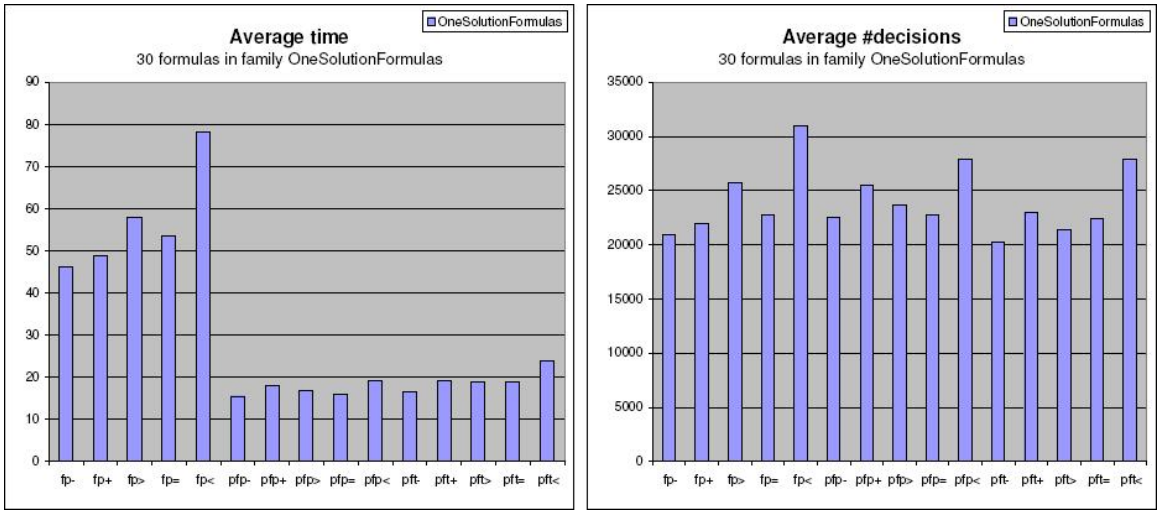


**Figure 1.** Avg time and decisions graphs of 1 solution formulas

#### 4.1.3 Branching Direction

Branching on the highest reduction quality literal performs better than branching on the smallest for both execution time and number of decisions and for both propagations and ternary clauses based heuristics. This shows that the higher probability of finding new conflict clauses is indeed more important than the higher probability of finding a satisfying assignment. This is expected since our hybrid solver is based on a typical conflict-driven solver. This is different from the general approach in look-ahead solvers where branching on the smallest reduction quality is preferred.

### 4.2 Random Formulas

In this section, the results of executing randomly created test files will be presented. The random files are divided in three categories regarding the number of variables they include. Thus, there are random files with 250, 300 and 350 variables. Moreover, the random files are divided in two other categories regarding whether the formulas are satisfiable or not.

#### 4.2.1 Random Files: 250 Variables

**Table 3. 250 Vars SAT**: Statistics of 1000 randomly created SAT formulas with 250 variables

| | Execution Time | | | | | | Decisions Number | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Best % | Min | Max | Avg | StdDev | AvgDev | Best % | Min | Max | Avg | StdDev | AvgDev |
| **o** | 6 | 0.004 | 12.029 | 1.524 | 1.550 | 0.441 | 0.2 | 252 | 449567 | 64651 | 61827 | 56078 |
| **pfp$^-$** | 8.8 | 0.004 | 12.257 | 0.935 | 0.935 | -0.148 | 8.2 | 44 | 33124 | 2970 | 2717 | -5603 |
| **pfp$^+$** | 7.8 | 0.004 | 5.096 | 0.996 | 0.849 | -0.087 | 6.4 | 50 | 15564 | 3169 | 2557 | -5404 |
| **pfp$^>$** | 8 | 0.004 | 5.120 | 0.940 | **0.839** | -0.143 | 7.2 | **34** | **14265** | 2885 | **2435** | -5688 |
| **pfp$^=$** | 11.2 | 0.008 | **4.820** | **0.837** | 0.845 | **-0.246** | 9.8 | 48 | 14286 | 2697 | 2520 | -5876 |
| **pfp$^<$** | 10.6 | **0.001** | 6.452 | 1.162 | 1.161 | 0.079 | 7.8 | 60 | 18884 | 3770 | 3517 | -4803 |
| **pft$^-$** | 8.6 | 0.004 | 12.293 | 0.955 | 1.050 | -0.128 | 11.2 | 40 | 28257 | **2466** | 2538 | **-6107** |
| **pft$^+$** | 7.6 | 0.004 | 6.936 | 1.107 | 1.078 | 0.024 | 9.6 | 44 | 16770 | 2848 | 2644 | -5725 |
| **pft$^>$** | 5.6 | 0.012 | 7.232 | 1.172 | 1.131 | 0.089 | 8 | 57 | 16914 | 2918 | 2693 | -5655 |
| **pft$^=$** | 10.4 | 0.004 | 7.184 | 1.110 | 1.169 | 0.027 | 12 | 33 | 17053 | 2873 | 2860 | -5700 |
| **pft$^<$** | **18.2** | 0.004 | 8.533 | 1.175 | 1.387 | 0.092 | **19.6** | 69 | 19896 | 3056 | 3398 | -5517 |

- According to table 3, regarding execution time, the best version is **pfp**$^{=}$ while regarding number of decisions, the best version is **pft**$^{-}$.
- Regarding execution time, propagating negative literals (**pfp**$^{-}$, **pft**$^{-}$) outperforms all other branching heuristics, while regarding number of decisions, outperforms propagating positive ones.
- Versions **pfp**$^{<}$ and **pft**$^{<}$ that branch on the smallest reduction quality literal perform rather poor.
- In 250 SAT problems, propagations based heuristic is by far more appropriate than ternary clauses heuristic. The later does not result in a lower number of decisions while it is also more expensive regarding time.

**Table 4. 250 Vars UNSAT**: Statistics of 1000 randomly created formulas with 250 variables

| | Execution Time | | | | | | Decisions Number | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Best % | Min | Max | Avg | StdDev | AvgDev | Best % | Min | Max | Avg | StdDev | AvgDev |
| **o** | 0 | 0.632 | 18.805 | 4.258 | 2.303 | 1.502 | 0 | 30148 | 682058 | 174115 | 84889 | 151718 |
| **pfp**$^{-}$ | 28.4 | 0.516 | 7.080 | 1.987 | 0.849 | -0.769 | 17.8 | 1783 | 20604 | 6240 | 2485 | -16158 |
| **pfp**$^{+}$ | 32.4 | 0.420 | 6.492 | 1.988 | **0.844** | -0.768 | 17.2 | 1552 | 19096 | 6260 | 2478 | -16138 |
| **pfp**$^{>}$ | **37.4** | **0.396** | **6.336** | **1.949** | 0.850 | **-0.807** | **34.2** | **1367** | **17772** | **5912** | **2384** | **-16486** |
| **pfp**$^{=}$ | 2 | 0.672 | 6.464 | 2.445 | 1.027 | -0.312 | 1 | 2292 | 19013 | 7540 | 2936 | -14858 |
| **pfp**$^{<}$ | 0 | 0.556 | 13.021 | 2.998 | 1.416 | 0.242 | 0 | 2042 | 37044 | 9418 | 4077 | -12980 |
| **pft**$^{-}$ | 0.4 | 0.636 | 8.141 | 2.477 | 1.089 | -0.279 | 12.2 | 1761 | 19399 | 6303 | 2613 | -16095 |
| **pft**$^{+}$ | 0.4 | 0.628 | 8.757 | 2.474 | 1.086 | -0.282 | 10.8 | 1741 | 20808 | 6291 | 2599 | -16107 |
| **pft**$^{>}$ | 0.2 | 0.672 | 9.893 | 2.662 | 1.213 | -0.095 | 7 | 1840 | 23099 | 6554 | 2814 | -15844 |
| **pft**$^{=}$ | 0 | 0.816 | 11.829 | 3.334 | 1.555 | 0.578 | 0 | 2299 | 26969 | 8298 | 3599 | -14100 |
| **pft**$^{<}$ | 0 | 0.968 | 13.133 | 3.746 | 1.744 | 0.990 | 0 | 2630 | 31034 | 9446 | 4084 | -12952 |

- Table 4 indicates that the best version regarding both execution time and number of decisions is version **pfp**$^{>}$ in UNSAT instances which is also close to the the best in SAT ones.
- Regarding propagations based heuristic, versions **pfp**$^{-}$ and **pfp**$^{+}$ perform almost equally and are the second best approaches for this family. Notice that in table 3, versions **pfp**$^{-}$ and **pft**$^{-}$ perform much better than versions **pfp**$^{+}$ and **pft**$^{+}$.
- Versions using ternary clauses heuristic perform much worse than those using the propagations one.
- In both propagations and ternary clauses based heuristics, versions that use the random branching and the smallest reduction quality heuristic perform extremely poorly. This fact strengthens the opinion that (1) random branching is not suitable for UNSAT instances since, due to conflict clauses, the costs for different branching orders are different and (2) smallest reduction quality heuristic is not appropriate for both SAT and UNSAT instances.

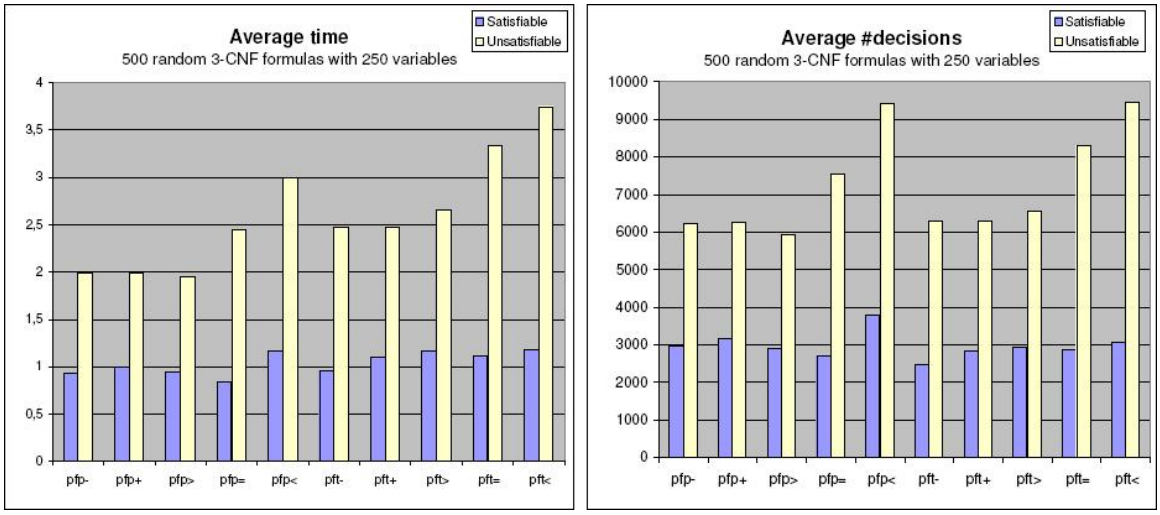


**Figure 2.** Avg time and decisions graphs of random SAT and UNSAT formulas with 250 variables

#### 4.2.2 Random Files: 300 Variables

**Table 5. 300 Vars SAT**: Statistics of 500 randomly created SAT formulas with 300 variables

| | Execution Time | | | | | | Decisions Number | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Best % | Min | Max | Avg | StdDev | AvgDev | Best % | Min | Max | Avg | StdDev | AvgDev |
| **o** | 5 | 0.044 | 109.951 | 14.918 | 17.910 | 6.444 | 0 | 2308 | 2899549 | 452962 | 490890 | 397788 |
| **pfp$^-$** | 10.2 | 0.072 | 36.206 | 6.859 | 6.644 | -1.615 | 9.6 | 287 | 69984 | 14986 | 13534 | -40187 |
| **pfp$^+$** | 7.6 | 0.016 | 34.390 | 7.084 | **6.464** | -1.390 | 8.2 | 84 | 67081 | 15495 | 13199 | -39678 |
| **pfp$^>$** | 5.8 | 0.012 | **32.458** | 6.865 | 6.286 | -1.608 | 6.6 | 67 | **61309** | 14590 | 12409 | -40584 |
| **pfp$^=$** | 11 | 0.012 | 55.143 | **6.612** | 7.454 | **-1.862** | 9.6 | 80 | 86889 | 14342 | 14684 | -40831 |
| **pfp$^<$** | 10.4 | **0.008** | 76.081 | 9.546 | 10.283 | 1.073 | 8.4 | 80 | 134049 | 20524 | 20249 | -34649 |
| **pft$^-$** | 8.8 | 0.016 | 40.055 | 7.629 | 7.627 | -0.845 | 10.4 | 80 | 68223 | 13891 | 13054 | -41283 |
| **pft$^+$** | 11.6 | 0.028 | 46.859 | 7.018 | 7.251 | -1.455 | 12.8 | 122 | 76626 | **12816** | **12396** | **-42357** |
| **pft$^>$** | 5.8 | 0.016 | 54.115 | 9.601 | 9.483 | 1.128 | 6.6 | 55 | 83515 | 16711 | 15390 | -38463 |
| **pft$^=$** | 7.6 | 0.032 | 54.199 | 8.085 | 9.036 | -0.389 | 8.6 | 128 | 87189 | 14583 | 15141 | -40591 |
| **pft$^<$** | **16.4** | 0.020 | 81.697 | 8.992 | 11.274 | 0.519 | **19.2** | 101 | 129281 | 16007 | 18673 | -39166 |

- Table 5 indicates that regarding execution time, the best version is **pfp$^=$** while regarding number of decisions, the best one is **pft$^+$**.
- Regarding both execution time and number of decisions, propagating negative literals outperforms propagating positive ones in case of the propagations based heuristic. Regarding ternary clauses heuristic, the opposite is the case. The later is a difference between tables 3 and 5.
- Both in 250 and 300 SAT problems, propagations based heuristic is by far more appropriate than ternary clauses heuristic. Moreover, randomly chosen literals heuristic seems to outperform the highest reduction quality heuristic which seems however to be the most stable one.

**Table 6. 300 Vars UNSAT**: Statistics of 500 randomly created UNSAT files with 300 variables

| | Execution Time | | | | | | Decisions Number | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Best % | Min | Max | Avg | StdDev | AvgDev | Best % | Min | Max | Avg | StdDev | AvgDev |
| **o** | 0 | 6.400 | 163.326 | 40.950 | 21.297 | 18.737 | 0 | 228337 | 4113537 | 1209105 | 554297 | 1064094 |
| **pfp$^-$** | 30.4 | 3.184 | 56.488 | 15.280 | 7.102 | -6.932 | 19 | 7809 | 108676 | 32729 | 13533 | -112281 |
| **pfp$^+$** | 28.8 | 3.616 | 60.320 | 15.217 | **6.856** | -6.995 | 17.2 | 8854 | 111245 | 32701 | 13149 | -112309 |
| **pfp$^>$** | **38.6** | **3.096** | **55.540** | **14.778** | 6.898 | **-7.434** | **38.6** | **7375** | **99993** | **30812** | **12778** | **-114198** |
| **pfp$^=$** | 1.6 | 4.200 | 79.397 | 19.757 | 9.227 | -2.455 | 0.6 | 9987 | 144122 | 40703 | 16868 | -104307 |
| **pfp$^<$** | 0 | 5.036 | 91.906 | 24.453 | 11.897 | 2.240 | 0 | 12251 | 168098 | 50960 | 22039 | -94051 |
| **pft$^-$** | 0.4 | 4.436 | 71.405 | 18.714 | 8.687 | -3.498 | 12.2 | 7614 | 112815 | 33352 | 14020 | -111658 |
| **pft$^+$** | 0.6 | 3.688 | 67.276 | 18.751 | 8.632 | -3.461 | 11.4 | 7444 | 105397 | 33343 | 13799 | -111668 |
| **pft$^>$** | 0 | 3.964 | 86.145 | 20.952 | 9.871 | -1.261 | 1 | 7831 | 130181 | 36016 | 15242 | -108995 |
| **pft$^=$** | 0 | 5.236 | 81.261 | 25.985 | 11.937 | 3.772 | 0 | 10265 | 127900 | 44467 | 18430 | -100544 |
| **pft$^<$** | 0 | 5.180 | 100.290 | 29.500 | 13.811 | 7.288 | 0 | 10309 | 153220 | 50928 | 21498 | -94082 |

- Table 6 indicates that the best version regarding both execution time and number of decisions is version **pfp$^>$**. Alike table 4, in table 6 versions **pfp$^-$** and **pfp$^+$** perform rather equally and are the second best versions of this family.
- Regarding ternary clauses heuristic, version **pft$^+$** performs equally to version **pft$^-$**. In total though, these versions perform much worse than those using the propagations based heuristic.
- Versions **pfp$^<$** and **pft$^<$** perform quite poor in comparison to the other versions of their corresponding heuristic. The fact that these two versions perform so poor, further strengthen the assumption that the smallest reduction quality heuristic is not appropriate for any kind of random instances, SAT and UNSAT.

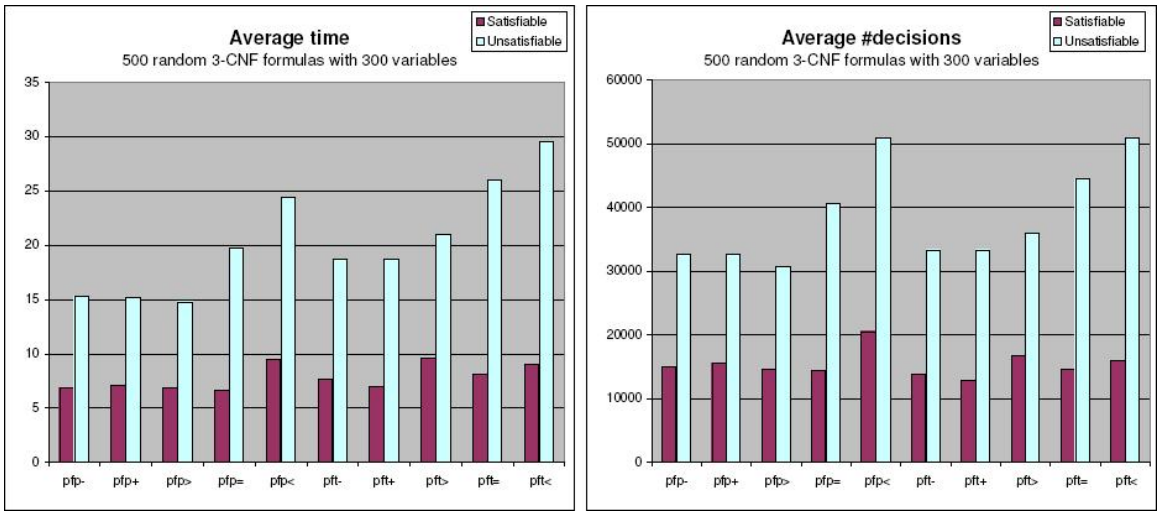


**Figure 3.** Avg time and decisions graphs of random SAT and UNSAT formulas with 300 variables

#### 4.2.3 Random Files: 350 Variables

As far as the execution of random files with 350 variables is concerned, there are no statistics for the ternary clauses heuristic. This is because we lacked the time required to run all versions for both SAT and UNSAT formulas. We feel that excluding the versions using the ternary clauses based heuristic can be justified since a scaling of the aforementioned results predicts poor performance.

**Table 7. 350 SAT**: Statistics of 500 randomly created SAT files with 350 variables

| | Execution Time | | | | | | Decisions Number | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Best % | Min | Max | Avg | StdDev | AvgDev | Best % | Min | Max | Avg | StdDev | AvgDev |
| **pfp$^-$** | 25.6 | 0.212 | **419.942** | 101.880 | **74.018** | -16.232 | 21.2 | 556 | **485540** | 135590 | 87373 | -16279 |
| **pfp$^+$** | 24.2 | 0.092 | 563.731 | 106.038 | 81.803 | -12.074 | 21.2 | 282 | 528526 | 138829 | 90985 | -13040 |
| **pfp$^>$** | **28.6** | 0.256 | 501.023 | **98.875** | 74.927 | **-19.237** | **37.2** | 645 | 512439 | **126473** | **81678** | **-25396** |
| **pfp$^=$** | 11.6 | **0.032** | 820.747 | 125.450 | 105.323 | 7.338 | 11 | **128** | 733423 | 157929 | 114116 | 6060 |
| **pfp$^<$** | 10 | 0.096 | 728.362 | 158.317 | 131.607 | 40.205 | 9.2 | 435 | 862700 | 200526 | 147804 | 48656 |

- Unlike tables 3 and 5, in which version **pfp$^=$** performed surprisingly well, in table 7, version **pfp$^>$** is by far the best regarding both execution time and number of decisions. This was expected since the performance of a random approach is decreasing as the complexity of the problem increases.

- This fact becomes visible by comparing figures 2 and 3. One could observe that the difference in execution time between versions **pfp$^=$** and **pfp$^>$** was decreasing for SAT instances while it remained fixed for UNSAT instances. The same was also the case for versions **pfp$^-$** and **pfp$^+$**.

**Table 8. 350 UNSAT**: Statistics of 151 randomly created UNSAT files with 350 variables

| | Execution Time | | | | | | Decisions Number | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Best % | Min | Max | Avg | StdDev | AvgDev | Best % | Min | Max | Avg | StdDev | AvgDev |
| **pfp$^-$** | 30.4 | 30.950 | 587.361 | 140.355 | 76.748 | -24.865 | 24.4 | **44211** | 556547 | 171386 | 77283 | -26350 |
| **pfp$^+$** | 30.2 | 30.006 | 468.749 | 139.358 | 73.011 | -25.862 | 27.6 | 47620 | 472349 | 171468 | 74792 | -26268 |
| **pfp$^>$** | **35.8** | **27.118** | **395.213** | **136.263** | **71.310** | **-28.957** | **46.8** | 44661 | **410923** | **162809** | **69405** | **-34927** |
| **pfp$^=$** | 3.2 | 40.243 | 1027.800 | 179.200 | 99.719 | 13.979 | 1 | 64336 | 806504 | 212160 | 94816 | 14424 |
| **pfp$^<$** | 0.2 | 45.479 | 1020.910 | 230.925 | 128.864 | 65.704 | 0 | 72573 | 899851 | 270856 | 125019 | 73121 |

- Table 8 also indicates that the best version regarding both execution time and number of decisions is version **pfp$^>$** in UNSAT instances.

- Alike table 4 and 6, version using random branching and smallest raduction quality heuristic are among the ones with the worst performance in both execution time and number of decisions.

### 4.3 Results Synopsis

From the experimental results presented above, we have reached to the following conclusions:

- According to figure 1, partial look-ahead is far more suitable for the hybrid solver causing a large timing gain in comparison to the full look-ahead.

- According to figures 2 and 3, the ternary clauses heuristic does not result in a lower number of decisions and since it is more expensive to compute will only slow down the performance.

- On random 3-SAT instances, the always negative and always positive branching performs comparable.

- On all random 3-SAT instances, branching on the largest reduction quality literal yields better results than branching on the smallest.

- In general, version **pfp$^>$** seems to be the best regarding both SAT and UNSAT instances.

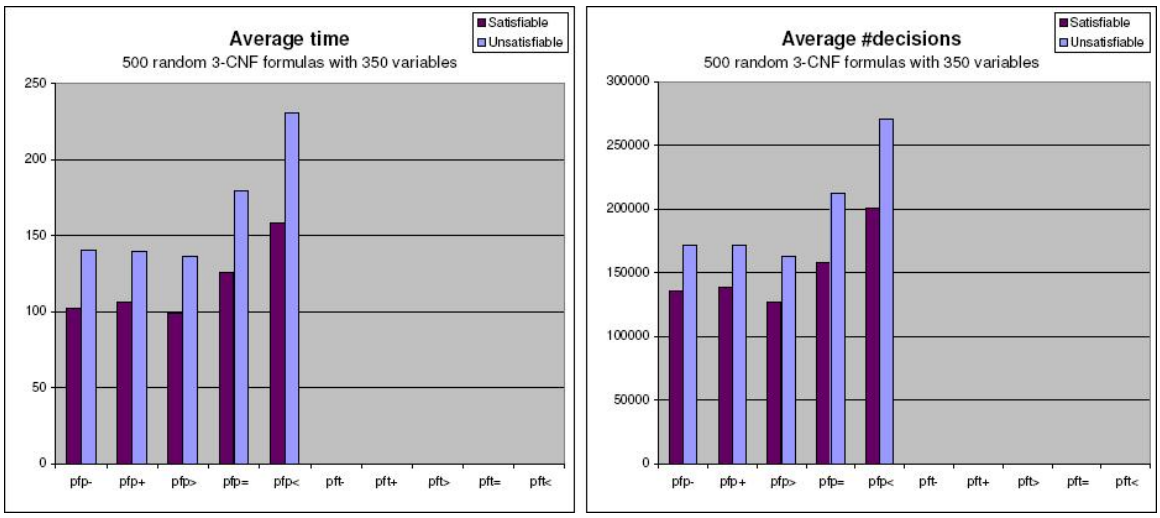


**Figure 4.** Avg time and decisions graphs of random SAT and UNSAT formulas with 350 variables

## 5. Conclusion

In this work several ways of integrating a look-ahead procedure in a conflict-driven solver have been investigated. We have tested different branching heuristics and found some useful ones and some others that seem inappropriate for this application. From the results presented in section 4, version **pfp**$^{>}$ seems the best approach for our hybrid solver for both SAT and UNSAT instances.

A rather surprising conclusion was found on the branching heuristic. Since in a look-ahead solver the branching direction is only of influence on satisfiable formulas the lowest reduction quality heuristic will exploit the higher probability of finding a satisfying assignment. In a conflict driven solver the branching direction is always of influence and going for the higher reduction quality means a higher change of finding new conflict clauses faster. This heuristic outperforms both the lower reduction quality and the branch always negative heuristic on satisfiable and unsatisfiable random formulas and might even proof the best in general.

In the foregoing chapters we have shown the performance of our hybrid solver on benchmarks of random formulas. Since look-ahead solvers typically perform well on random formulas, it came as no surprise that the hybrid solver performed better than the original conflict-driven `MiniSat` solver. Thus, our hybrid solver clearly exploits the advantages of the look-ahead procedure. Unfortunately, other formulas that used to be solved easily with `MiniSat` are hard or near impossible to solve using our hybrid solver. In those cases the look-ahead procedure will typically have to restart many times because of the failed literals it finds. Without the "smart" decision by the look-ahead procedure but by exploiting the backtracking features, the purely conflict-driven solver would have resolved many of these conflicts much faster.

This is why we feel that future work should focus on a heuristic that dynamically alters the solver behavior from purely conflict-driven to strongly influenced by the look-ahead procedure, for example based on some measurement of progress. The solver behavior could be altered by changing the percentage of variables involved in the look-ahead procedure or by simply "turning on or off" the look-ahead procedure. We feel confident that strengthened with a good heuristic for dynamically altering solver behavior a widely applicable SAT solver can be developed using this hybrid set-up.

## References


[1] Davis M., Logemann G., and Loveland D. (1962), 'A machine program for theorem proving', Communications of the ACM, 5(7) pp. 394-397.

[2] Dubois, O. and Dequen G. (2001), 'A backbone-search heuristic for efficient solving of hard3-sat formulae', In Proc. of the 17th International Joint Conference on Artificial Intelligence (IJCAI 01).

[3] Een N. and Sorenson N. (2003), 'An extensible SAT solver', In Proc. of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 03).

[4] Heule M., van Zwieten J., Dufour M and van Maaren H. (2005), March_eq: Implementing Additional Reasoning into an Efficient Lookahead Sat Solver. SAT 2004 Springer LNCS 3542, 345-359.

[5] Goldberg, E. and Novikov Y. (2002), 'BerkMin: A fast and robust SAT-solver', In Design, Automation, and Test in Europe (DATE 02), pages 142-149.

[6] Kullmann, O. (2002), 'Investigating the behaviour of a SAT solver on random formulas', Submitted to Annals of Mathematics and Artificial Intelligence.

[7] Li, Anbulagan, (1997), 'Look-Ahead versus Look-Back for Satisfiability Problems', Springer-Verlag, LNCS 1330, pages 342-356, Autriche.

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