

# MiniMarch

Embedding lookahead direction heuristics in a conflict driven solver

## Research assignment

Siert Wieringa

August 2007

# MiniMarch

Embedding lookahead direction heuristics in a conflict driven solver

## **Research assignment**

Ing. S. Wieringa

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Program Computer Science

August 2007

---



## Acknowledgements

The work described in this report was carried out as part of the final year of my MSc program in Computer Science. This project would not have been possible without the help of friends and fellow researchers.

I would like to thank Hans van Maaren and Marijn Heule for supervising me and Cees Witteveen for his personal guidance. Furthermore I would like to thank the organizers of the 2007 SAT Competition, Daniel Le Berre and Laurent Simon, for the special effort they took to notify me of occurring bugs and especially for running the bug fixed version of MiniMarch after I published it on Daniel Le Berre's wonderful and up-to-date website *SAT Live!* ([www.satlive.org](http://www.satlive.org)).

Special thanks goes to my friends Minze Walvius and Jan van der Meer and everybody else involved in their respective businesses Advier and INOXA<sup>1</sup>. They were kind enough to let me work at their shared office in Delft where I was not only facilitated with a desk and computers but could also rely on the availability of coffee, lunch and most importantly on their company and support.

---

<sup>1</sup> Advier are mobility managers ([www.advier.nl](http://www.advier.nl)), INOXA builds websites ([www.inoxa.nl](http://www.inoxa.nl))

## Abstract

This paper presents the MiniMarch SAT solver which is a modified version of the conflict driven SAT solver MiniSat. During its development the focus was on branch direction heuristics. The choice made by the developers of MiniSat to always assign false to decision variables was one of the main observations that lead to this project. In satisfiable formulas a satisfying assignment can be found without backtracking if there would be a direction heuristic that would always make the correct choice. So in theory, their power is enormous. Their practical influence on solver performance is also too large to neglect.

The good performance of MiniSat's simple direction heuristic on a wide range of industrial benchmarks is probably caused by the encoding of those instances. Those properties would be lost if the benchmarks were shuffled. Making the correct direction decision is very much related to building resistance against shuffling of the input formula. By building such resistance we rely on heuristics that are more generally applicable to solving Boolean formulas.

In this document the new heuristics used in MiniMarch are presented along with results from the first stage of the SAT 2007 competition and results on other industrial benchmarks that were shuffled in various ways.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	SAT Solver design . . . . .	5
1.2	Lookahead . . . . .	7
1.3	Conflict driven . . . . .	7
1.4	Motivation for developing MiniMarch . . . . .	8
<b>2</b>	<b>Shuffling</b>	<b>9</b>
2.1	Changing indices . . . . .	9
2.2	Swapping literal polarities . . . . .	9
2.3	Clause shuffling . . . . .	10
2.4	Literal order in clauses . . . . .	11
<b>3</b>	<b>Direction Heuristic</b>	<b>12</b>
3.1	Variable balance . . . . .	12
3.2	Lookahead branch direction . . . . .	13
3.3	Lookahead criterion . . . . .	13
3.4	Lookahead implementation . . . . .	14
3.5	The static direction heuristic . . . . .	15
<b>4</b>	<b>Simplifier</b>	<b>17</b>
4.1	Simplifying operations . . . . .	17
4.2	Modifications . . . . .	17
<b>5</b>	<b>Test plan</b>	<b>20</b>
5.1	SAT Competition scoring . . . . .	20
5.2	Solution points . . . . .	20
5.3	Speed points . . . . .	20
5.4	Solvers for comparison . . . . .	21
<b>6</b>	<b>Results</b>	<b>22</b>
6.1	SAT Competition . . . . .	22
6.2	Parameter option tests . . . . .	24
<b>7</b>	<b>Conclusion</b>	<b>26</b>
<b>A</b>	<b>Office PC test environment</b>	<b>28</b>

# 1 Introduction

Many of the problems we encounter in daily life can be described in propositional logic. Computers “reason” and make decisions based on logic descriptions of the desired functionality. While crossing a street with traffic lights you expect the safety property stating that the light you are watching can only be green if the lights at crossing streets signal red to be satisfied by the traffic lights control system.

In propositional logic a property like green on street 1 implies not green on street 2 can be stated as:  $G_{S1} \rightarrow \neg G_{S2}$ . Consider a crossing of two one way streets where a traffic light can only be green or not green and where traffic may only cross in a straight line. The legal states of the traffic lights control system are the states for which both the following constraints are satisfied:

- $G_{S1} \vee G_{S2}$
- $\neg G_{S1} \vee \neg G_{S2}$  ( $= G_{S1} \rightarrow \neg G_{S2}$ )

In words: Any state is valid in which there is a light that is green and there is a light that is not green.

The satisfiability problem (SAT) is the problem of finding a satisfying assignment for a Boolean formula or concluding that no such assignment is possible. A SAT solver is a computer program that is designed for that task. SAT solvers can be used to solve potentially large problems, for example those encountered in the synthesis and testing of logic chips. Besides that they are of great theoretical interest as satisfiability was the first problem proven to be NP-complete [9].

## 1.1 SAT Solver design

Most current SAT solvers, including the one presented in this report, expect the input formula to be in the *conjunctive normal form* (CNF). In the CNF form a formula is a conjunction of clauses. A clause is a disjunction of literals. A literal is a Boolean variable  $x_i$  or its negation  $\neg x_i$ .

The traffic light example in the first paragraph is in the CNF form. Consider the case where one decides to assign  $G_{S1}$  the value *true*. In that case, the first clause is satisfied and the second clause shrinks to length 1 because it only has one undefined literal left that needs to be assigned a value. As  $\neg G_{S2}$  must be *true* to satisfy the clause  $G_{S2}$  has to be assigned *false*.

A clause of length one is called a *unit clause* and as all clauses need to be satisfied it forces the only literal contained in it to the value *true*. This forced assignment might result in new unit clauses which force new variable assignments. This process is called *iterative unit propagation* (IUP). If an empty clause, which is a clause of length zero, occurs during the IUP process then the variable assignment is invalid. If all possible variable assignments are invalid the formula is *unsatisfiable*.

Most current SAT solvers follow the DPLL framework [1] when solving a formula. The pseudo-code for the DPLL framework is shown below. A call to the ITERATIVEUNITPROPAGATION function performs IUP, and during that process all satisfied clauses are removed and all clauses containing a literal that evaluates to *false* shrink. IUP on the input formula might lead to a satisfying assignment or the conclusion that the formula is unsatisfiable. If neither happens a *branch literal* is chosen and assigned the value *true*. The DPLL function is then called recursively, which might either result in a satisfying assignment or the conclusion that the subformula is unsatisfiable. If the subformula with the branch literal assigned the value *true* is unsatisfiable the literal is assigned the value *false* and a new recursive call is made. If both subformulas are unsatisfiable the formula that was passed to this DPLL function call is unsatisfiable.

---

**Algorithm 1.1** DPLL( $\mathcal{F}$ )

---

```

1:  $\mathcal{F} := \text{ITERATIVEUNITPROPAGATION}(\mathcal{F})$ 
2: if  $\mathcal{F} = \emptyset$  then
3:   return “satisfiable”
4: else if  $\mathcal{F}$  contains empty clause then
5:   return “unsatisfiable”
6: end if
7:  $x_i := \text{GETBRANCHLITERAL}()$ 
8: if DPLL( $\mathcal{F} \cup \{x_i\}$ ) = “satisfiable” then
9:   return “satisfiable”
10: else
11:   return DPLL( $\mathcal{F} \cup \{\neg x_i\}$ )
12: end if

```

---

The branch literal is chosen by the GETBRANCHLITERAL function. As a literal is a variable with a polarity that function has to make two important decisions, which branch variable to choose and whether or not to negate it, or in other words what branch direction to take.

To minimize the size of the search tree the branch variable has to be heuristically selected to cause a relatively large reduction of the formula. Whilst developing MiniMarch research focused on the branch direction heuristic. Research into branch direction heuristics is useful as they are very powerful in theory. A satisfying assignment will be found without backtracking if it exists and the heuristic always takes the right decision. In practice however predicting the right decision is difficult as the design of direction heuristics is subjected to two complementary theories. The first is to estimate which of the two possible subformulas will require the least computational time to solve and to branch in that direction. In general, this will be the direction of the most constraint subformula. The second theory is to estimate which of the two has the largest probability of containing a satisfying solution which is generally in the direction of the least constraint subformula.

## 1.2 Lookahead

In a lookahead SAT solver an expensive branch variable decision heuristic is used that attempts the propagation of multiple variables in both directions and measures the resulting reduction of the formula for each of them. Let the reduced formula resulting of the propagation of literal  $x_i$  and the iterative unit propagation that follows be  $\mathcal{F}^* = \text{IUP}(\mathcal{F} \cup \{x_i\})$ .

The reduction quality of a literal  $x_i$  is a metric that measures the reduction of  $\mathcal{F}^*$  compared to  $\mathcal{F}$ . It is denoted  $\text{DIFF}(\mathcal{F}, \mathcal{F}^*)$ . A variables reduction quality is defined as a function of the reduction quality of the two corresponding literals, usually the product. A lookahead solver will choose the variable with the highest reduction quality measurement as the next branch variable.

The measured formula reductions for both polarities of a branch variable are useful to the branch direction heuristic as well. Here, the heuristic can either select the polarity with the highest DIFF if it was designed to exploit the advantages of the subformula that requires the least time to solve or the polarity with the lowest DIFF if it was designed to branch in the direction with the highest probability of finding a satisfying assignment. In a lookahead solver the chosen branching direction is only of influence on satisfiable formulas because in unsatisfiable formulas both subtrees need to be investigated to conclude that they both lead to an invalid assignment.

## 1.3 Conflict driven

---

**Algorithm 1.2** CONFLICTDRIVENDPLL ( $\mathcal{F}$ )

---

```
1: while TRUE do
2:    $x_i := \text{GETBRANCHLITERAL}()$ 
3:   if an  $x_i$  is selected then
4:      $\mathcal{F} := \text{ITERATIVEUNITPROPAGATION}(\mathcal{F} \cup \{x_i\})$ 
5:     while  $\mathcal{F}$  contains empty clause do
6:        $blevel := \text{ANALYZECONFLICTS}()$ 
7:       if  $blevel = 0$  then
8:         return “unsatisfiable”
9:       else
10:         $\text{BACKTRACK}(blevel)$ 
11:         $\mathcal{F} := \text{ITERATIVEUNITPROPAGATION}(\mathcal{F})$ 
12:      end if
13:    end while
14:  else
15:    return “satisfiable”
16:  end if
17: end while
```

---

A conflict driven SAT solver applies a slightly modified DPLL framework. These solvers use non-chronological backtracking and derive new clauses, *conflict*

*clauses*, while doing so.

Because of conflict learning and non-chronological backtracking the branch direction is no longer only of influence on satisfiable formulas. The main conclusion of previous work [6] is that in contrary to a lookahead solver in a conflict driven solver it is rewarding to branch in the most constraining direction. This is probably because the conflict learning procedures will find more and shorter conflict clauses if the formula is more constrained.

#### 1.4 Motivation for developing MiniMarch

The state-of-the-art conflict driven SAT solver MiniSat [3] does not use a complex heuristic for the branch direction instead it assigns false to each decision variable<sup>2</sup>. This setting appears to be optimal on a wide range of industrial benchmarks and is probably effective due to the encoding of those instances. Making consequent choices seems to be a very important factor as can also be seen from the results with random branch direction in previous work [6].

The development of MiniMarch is motivated by the feeling that direction heuristics have a potentia that should be exploited and that the dependance of MiniSat on currently popular encodings is largely unnecessary. In a way, MiniMarch broadens MiniSat's applicability.

---

<sup>2</sup> Since version 2.0 it actually branches in the direction set by the command line parameter "polarity-mode" first. The default setting is false but it can also be set to true or random

## 2 Shuffling

When a real life problem is translated to a CNF formula the problem is in fact *encoded*. As the task of encoding a problem requires some sort of structured approach the resulting CNF formula will hold some properties that are caused by the encoder rather than by the description of the actual problem. The same holds for translations of general Boolean formulas to CNF formulas and in fact for all possible translations, including human translations of natural languages.

Solvers that are targeted at real life applications exploit these properties. As stated before the choice made by the developers of MiniSat to always decide false as the best branch direction is an example of that. In the practical applications where solvers are used for a limited set of problems this type of heuristics is in fact very useful but they do not contribute to research into the problem of solving Boolean formulas in general.

By shuffling benchmarks randomly they might loose some of the encoders characteristics while they still describe the same problem. Due to all the encoding specific heuristics used the performance of SAT solvers, especially that of the conflict driven ones, will often be strongly influenced by such shuffling operations. By focussing on making a solver less sensitive to shuffling it was hoped to find heuristics that perform well in general.

### 2.1 Changing indices

The simplest form of formula shuffling might be to change the variable indices or in other words to change or swap the identifiers of variables. Although this is a simple change both MiniSat and MiniMarch might be influenced by this since the literals in a clause are sorted by their identifiers and there are watch pointers on the first two literals. Variable identifiers do not have any influence on the new heuristics presented in MiniMarch.

### 2.2 Swapping literal polarities

The performance of most SAT solvers is seriously affected by the swapping of the polarity of literals in the input formula (e.g.  $x_i \rightarrow \neg x_i$  and  $\neg x_i \rightarrow x_i$  for all occurrences of  $x_i$ ) because of the lack of smart branch direction heuristics. This was one of the most important observations that lead to this project. It seemed very unnecessary that a solver would be affected by such a simple change in the formula. Of course formulas remain satisfiability equivalent under this “transformation” but intuitively it seems reasonable that they should be equivalently hard to solve as well.

The main conclusion of previous work [6] was that on a variable decision in a conflict driven solver it is rewarding to branch in the direction that generates the largest number of new constraints probably because it helps the solver find conflict clauses faster. This is contrary to the strategy lookahead solvers employ which is to go in the direction that is the least constraining because that direction has the highest probability of containing a satisfying assignment.

In a lookahead solver the chosen branching direction is not of any influence on unsatisfiable formulas because as there is no conflict learning both subformulas need to be investigated anyway.

As a first step to facilitating more resistance against sign swapping the conflict driven solver could be adapted such that it branches in the direction of the literal that occurs least frequent in the input formula. The least frequent occurring literal is a rough estimate for the literal that yields the highest number of constraints when propagated because all clauses in which the literal with the most occurring polarity occurs will shrink.

This approach however is slightly naive. The first problem is that in many formulas a large number of literals occur in both signs equally often. Secondly, using literal count might be a rough estimate for the most constraining decision but if literal  $x_i$  occurs slightly more frequent than literal  $\neg x_i$  but the latter one occurs in shorter clauses then that literal might have the highest probability of being part of the largest number of new binary clauses at the time the decision is made.

An improvement to choosing the literal with the lowest literal count would be to choose the literal with the lowest weight where literal weight is defined as:

$$w_{literal}(x_i) = \sum_{c \in \mathcal{F} \wedge x_i \in c} 2^{-|c|} \quad (2.1)$$

By this definition literal weight is not only depending on the number of occurrences but also on the length of the clauses it occurs in. Note that it is still possible that a literal and its negation have an equal weight.

### 2.3 Clause shuffling

Changing the order in which clauses appear in the input formula might have some influence on the performance of a SAT solver. In the MiniSat case the watch pointer lists that are stored for each literal for example will also be shuffled if the input is shuffled. This might have influence if a literal is propagated. If the clauses in the watch pointer list are sorted with the shortest clauses first the propagation procedures might speed up because the strongest constraints will often be found faster.

Besides this adding the clauses from the input formula to the solvers set of clauses while they are sorted in increasing length will speed up the initial iterative unit propagation process.

It is clear that if clauses are sorted in the pre-processing the effect of clause shuffling can be minimized as far as the clauses can be sorted uniquely. However, it is arguable whether it is worth the extra pre-processing time and it probably is not in the current version of MiniSat.

Clause sorting is used in MiniMarch because there was another advantage besides the described advantages. It gave the opportunity to make the simplifier

found in MiniSat 2.0 ([3], first presented as the pre-processing tool SATElite in [2]) more resistant against input formula shuffling.

The first sorting criterion for clauses is their length, where clauses with the shortest length are ordered first. Amongst clauses with equal length the clauses with the highest weight are ordered first. Clause weight is defined as the sum of the weights of the literals contained in it.

$$w_{clause}(c) = \sum_{x_i \in c} w_{literal}(x_i) \quad (2.2)$$

## 2.4 Literal order in clauses

As described in paragraph 2.1 both MiniSat and MiniMarch sort the literals within the clauses, where all literals with the same polarity are sorted with the lowest identifier first and all literals that are negated proceed all unnegated literals. On the positive side this means that the shuffling of literals within the clauses of the input formula will not have any effect on both solvers. On the negative side however this is one of the main vulnerabilities of MiniMarch to shuffling of variable indices and polarities as the solver has watch pointers attached to the first two literals in the sorted clause. The choice to prefer watching negated literals with the smallest possible variable index can only be justified as a heuristic that exploits the formulas encoding. In the current version of MiniMarch this has not gotten attention but it should be given that in future versions.

### 3 Direction Heuristic

#### 3.1 Variable balance

The branch decision heuristic in MiniMarch has been left unchanged from MiniSat’s activity based heuristic. MiniMarch however does apply a form of lookahead to the branch direction heuristic. MiniMarch will only do lookahead on variables that it regards to be *balanced*. As doing lookahead for branch direction decisions is costly the balance of a variable is an important notion in MiniMarch. Before the presentation of MiniMarch’s definition of variable balance please recall the definition of the weight of a literal (definition 2.1) which is:

$$w_{literal}(x_i) = \sum_{c \in \mathcal{F} \wedge x_i \in c} 2^{-|c|}$$

The balance of a variable is defined as:

$$bal(x_i) = \frac{w_{literal}(x_i) + 1}{w_{literal}(\neg x_i) + 1} + \frac{w_{literal}(\neg x_i) + 1}{w_{literal}(x_i) + 1} - 2 \quad (3.1)$$

This value is small if the variable is balanced, e.g. both literals have a near equal weight. As MiniMarch does lookahead on balanced variables a reference balance  $bal_{ref}$  had to be defined. Balanced variables are defined as all variables  $x_i$  for which  $bal(x_i) < bal_{ref}$ . In the following definition  $VAR(\mathcal{F})$  is the set of variables in  $\mathcal{F}$ .

$$bal_{ref} = \frac{\sum_{x_i \in VAR(\mathcal{F})} bal(x_i)}{|VAR(\mathcal{F})|} \quad (3.2)$$

When the development of MiniMarch was first started the reference balance was defined as the average of the values  $bal(x_i)$  for all  $x_i$  as stated in definition 3.2. At some point this definition was changed to definition 3.5 as it was considered to be a better analogue to variable balance. Unfortunately, it was not investigated thoroughly at the time and it turned that the earlier definition would have been a better choice. It is presented here to complete the description of solver versions for which results are presented in this report.

$$w_{poslit} = \sum_{x_i \in VAR(\mathcal{F})} w_{literal}(x_i) \quad (3.3)$$

$$w_{neglit} = \sum_{x_i \in VAR(\mathcal{F})} w_{literal}(\neg x_i) \quad (3.4)$$

$$bal_{ref} = \frac{w_{posit} + 1}{w_{neglit} + 1} + \frac{w_{neglit} + 1}{w_{posit} + 1} - 2 \quad (3.5)$$

### 3.2 Lookahead branch direction

If the branch variable heuristic in MiniMarch decides on a variable  $x_i$  with  $bal(x_i) < bal_{ref}$  the solver will do lookahead to determine the best branch direction for  $x_i$ .

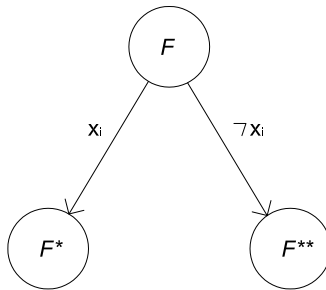


Figure 3.1: Lookahead

The solver propagates the variable in both directions leading to two different subformulas  $\mathcal{F}^*$  and  $\mathcal{F}^{**}$ . If neither  $\mathcal{F}^*$  nor  $\mathcal{F}^{**}$  contains an empty clause the subformula that is reduced the most compared to  $\mathcal{F}$  according to a metric, the *lookahead criterion*, is chosen as the branch direction. If there is an empty clause in one of the subformulas the propagated literal is called a *failed literal*. Since MiniMarch is a conflict driven solver the conflict will be analyzed, conflict clauses will be added and the solver will backtrack to the appropriate level to resolve the conflict. In MiniMarch, if  $x_i$  is a failed literal backtracking will start immediately without first analyzing the result of propagating  $\neg x_i$ .

### 3.3 Lookahead criterion

For a lookahead solver the number of new binary clauses after the propagation of a literal is a good metric to use as a lookahead criterion [8]. Unfortunately, that number can not be counted cheaply in MiniSat because of the way its datastructures are implemented.

A lookahead criterion that was easy to implement in MiniMarch was the number of propagations of other variables that result of the propagation of the branch variable. The definition of this criterion is:

$$q(x_i) = |IUP(x_i)| \quad (3.6)$$

In which  $IUP(x_i)$  is the set of variables whose value becomes fixed as a result of the propagation of  $x_i$ . The downside of this criterion is that it does not look beyond the direct effect of propagating a variable. The hard to implement new binary clauses heuristic would look a bit further since it actually attempts to find a variable that gives the most opportunities to reduce the formula on the *next* branch variable decision.

An attempt was made to come up with a metric that does look a little further but that was also easy to calculate using only the existing data structures. A metric was found that counts the watch pointer moves.

Each unsatisfied clause in MiniSat has a watch pointer on the left most two undefined literals. If a value is propagated for a variable that is pointed to by a watch pointer the watch pointer moves right to the next undefined literal. If there is no such literal the solver is left with a unit clause and will have to propagate the variable assignment that satisfies that unit clause. The new metric counts the number of steps a watch pointer has to move right before it reaches another undefined literal if there is such a literal. The idea is that this is an indication of the reduction that has already found place in this clause. The larger the reduction that has found place, the shorter the clause will be, the more worthy it is.

Let the total sum of right moves of all watch pointer literals resulting of the propagation of variable  $x_i$  be defined as  $m(x_i)$ . In MiniMarch, the two lookahead criteria described here are mixed which finally leads to a new criterion which is:

$$q_\lambda(x_i) = |IUP(x_i)|^\lambda * m(x_i)^{1-\lambda} \quad (3.7)$$

In which  $\lambda$  is a parameter which is a natural number in the range  $[0,1]$  that determines how the two lookahead criteria should be mixed. The default is to give both criteria equal weight ( $\lambda = 0.5$ ).

### 3.4 Lookahead implementation

In figure 3.1 the concept of propagating both literals for lookahead was shown. In the actual implementation of the lookahead procedure the solver will first have to propagate  $x_i$  to get to the subformula  $\mathcal{F}^*$ , then backtrack to  $\mathcal{F}$  and propagate  $\neg x_i$  to get to  $\mathcal{F}^{**}$ . This is shown in the steps A, B and C of figure 3.2.

If  $\mathcal{F}^{**}$  turns out to be the most reduced subformula the propagation of  $\neg x_i$  has already found place and the solver can continue. If on the other hand  $\mathcal{F}^*$  turns out to be the most reduced subformula the solver will have to backtrack to  $\mathcal{F}$  and then propagate  $x_i$  again. This is shown in steps D and E of the figure. In that scenario three literal propagations and two backtrack operations are required before the solver can continue. To minimize the number of times



The idea is that in general branching in the direction of the least occurring literal will strengthen the highest number of clauses. Since MiniMarch was intended to be less affected by shuffling it would not be wise to just make a default choice in case the number of occurrences are equal since if the literal polarities are swapped ideally the branch direction should change as well. It is therefore left undefined in that case.

If at a later stage the solver wants to make a branch direction decision for a variable  $x_i$  for which the entry in the  $ph$  array is undefined it will have to be defined based on either the number of watch pointers attached to  $x_i$  and  $\neg x_i$  or on the polarity of the first occurrence of the variable in the input formula.

Definition 3.9 states how  $ph$  should be defined if it was not defined by 3.8. The number of watch pointers attached to literal  $x_i$  is defined as  $\#watch(x_i)$  and it is used to give an indication of the expected formula reduction. If the number of watch pointers is equal for both literals the `FIRSTOCC` function is used. `FIRSTOCC` is a function that returns the literal of the first occurrence of variable  $x_i$  in the input formula. It is mainly used as a tie breaker but as clause sorting is used it still is resistant to sign swapping as long as the clauses containing the decision variable can be uniquely sorted.

$$\begin{aligned}
 ph[x_i] = & \quad x_i & \quad & \text{if } \#watch(\neg x_i) > \#watch(x_i) & \quad (3.9) \\
 & \neg x_i & \quad & \text{if } \#watch(\neg x_i) < \#watch(x_i) \\
 & \text{FIRSTOCC}(x_i) & \quad & \text{if } \#watch(\neg x_i) = \#watch(x_i)
 \end{aligned}$$

## 4 Simplifier

MiniSat 2.0 comes with an integrated simplifier that was first presented as the stand alone formula simplification tool SATElite [2]. It attempts to eliminate variables and clauses from the input formula in order to reduce its size before the actual solving takes place. The simplifier is also included in MiniMarch in a slightly modified form. The modifications that were made drastically reduce the influence that polarity swapping and clause shuffling had.

### 4.1 Simplifying operations

This paragraph describes the basic operation of the simplifier. Please note that the contents of this paragraph have been largely taken from [2], so refer to that paper for a more detailed description.

The two clauses  $C_1 = \{x, a_1, \dots, a_n\}$  and  $C_2 = \{\neg x, b_1, \dots, b_m\}$  imply the clause  $C = \{a_1, \dots, a_n, b_1, \dots, b_m\}$  which is called the *resolvent* of the two original clauses by performing *resolution* on the variable  $x$ . This is written as  $C = C_1 \otimes C_2$ . The  $\otimes$  operator can also be defined for sets of clauses as:

$$S_1 \otimes S_2 = \{ C_1 \otimes C_2 \mid C_1 \in S_1, C_2 \in S_2 \} \quad (4.1)$$

Let  $S_x$  be the set of clauses in which  $x$  occurs,  $S_{\neg x}$  the set of clauses in which  $\neg x$  occurs and  $S = S_x \cup S_{\neg x}$ . The variable  $x$  can be eliminated by pairwise resolving every clause in  $S_x$  with every clause in  $S_{\neg x}$ . The resolvents  $S' = S_x \otimes S_{\neg x}$  replace the original clauses in  $S$ . This is referred to as simplification by *clause distribution*.

The simplifier uses another simplification rule based on the observation that clause distribution leads to many subsumed clauses. A clause  $C_1$  subsumes  $C_2$  if  $C_1 \subseteq C_2$ , in that case  $C_2$  is redundant and can be removed. Besides this they observed that it often occurs that a clause  $C_2$  almost subsumes a clause  $C_1$ , except for a single literal  $\neg x$  which occurs with opposite sign in  $C_1$ . Resolving these clauses on  $x$  will produce a clause  $C'_1$  which subsumes, and therefore can replace,  $C_1$ . For example if  $C_1 = \{x, a, b\}$  and  $C_2 = \{\neg x, a\}$  then the resolvent is  $C'_1 = \{a, b\}$ . In that case  $C_1$  has been strengthened by the *self-subsumption* using  $C_2$ . This simplification rule is referred to as *self-subsumption resolution*.

### 4.2 Modifications

Eliminating variable  $x$  using simplification by clause distribution requires pairwise resolving of the clauses in  $S_x$  with the clauses in  $S_{\neg x}$ . In the simplifier integrated in MiniSat 2.0 the pairs are resolved using the scheme described in algorithm 4.1.

In case a resolvent contains two literals with opposite polarities it will always be satisfied and can be discarded. In other cases the resolvent is added to the set of clauses by RESOLVE. In case a unit clause is added iterative unit propagation

---

**Algorithm 4.1** ORIGINAL RESOLUTION SCHEME

---

```
1: for all clauses  $C_1$  in  $S_x$  do  
2:   for all clauses  $C_2$  in  $S_{\neg x}$  do  
3:     RESOLVE(  $C_1, C_2$  )  
4:   end for  
5: end for
```

---

is done directly. If the set of clauses is sorted with the shortest clauses first the resolvents will be shorter and stronger and will be more likely to force new variable assignments as a result of iterative unit propagation. The more variable assignments that have taken place, the shorter the other resolvents will become. As these processes are repeated multiple times the simplifier is affected quite heavily by a different clause order. By sorting the clauses as described in paragraph 2.3 most of this effect can be undone.

The shuffling operation that might have even more effect on the simplifier is polarity swapping as it will in fact swap the sets  $S_x$  and  $S_{\neg x}$ . Consider the example where  $S_x$  contains the clauses  $C_1$  and  $C_2$  furthermore  $S_{\neg x}$  contains clauses  $C_3$  and  $C_4$ . If the original pairwise resolution scheme described in algorithm 4.1 is used the pairs will be resolved in the order displayed in the left column of the following table.

$S = \{C_1, C_2, C_3, C_4\}$	
$S_x = \{C_1, C_2\}$	$S_x = \{C_3, C_4\}$
$S_{\neg x} = \{C_3, C_4\}$	$S_{\neg x} = \{C_1, C_2\}$
$C_1 \otimes C_3$	$C_3 \otimes C_1$
$C_1 \otimes C_4$	$C_3 \otimes C_2$
$C_2 \otimes C_3$	$C_4 \otimes C_1$
$C_2 \otimes C_4$	$C_4 \otimes C_2$

Table 4.1: Order of pairwise resolving, original scheme

In the right column the order of resolution after swapping  $S_x$  and  $S_{\neg x}$  is shown. The first resolution is on the same pair of clauses but the other three are in a different order. As can be seen from this example clause sorting does not resolve the influence of polarity swapping. It is however a prerequisite for the solution that was implemented in MiniMarch.

The implementation of the modified pairwise resolution scheme is stated in algorithm 4.2. Rather than having a loop for both sets  $S_x$  and  $S_{\neg x}$  a clause  $C_1$  is now selected from the sorted set  $S$  and resolved with a clause  $C_2$  with opposite polarity that is sorted after  $C_1$ . From the example in table 4.2 it can be seen that the swapping of literal polarities no longer has any effect on the simplifier.

---

**Algorithm 4.2** MODIFIED RESOLUTION SCHEME

---

```
1:  $S := S_x \cup S_{\neg x}$ 
2: for  $i = 0$  to  $|S|-1$  do
3:    $C_i = i$ 'th clause in  $S$ 
4:   for  $j = i + 1$  to  $|S|$  do
5:      $C_j = j$ 'th clause in  $S$ 
6:     if  $C_i \in S_x$  xor  $C_j \in S_x$  then
7:       RESOLVE(  $C_i, C_j$  )
8:     end if
9:   end for
10: end for
```

---

$S = \{C_1, C_2, C_3, C_4\}$	
$S_x = \{C_1, C_2\}$	$S_x = \{C_3, C_4\}$
$S_{\neg x} = \{C_3, C_4\}$	$S_{\neg x} = \{C_1, C_2\}$
$C_1 \otimes C_3$	$C_1 \otimes C_3$
$C_1 \otimes C_4$	$C_1 \otimes C_4$
$C_2 \otimes C_3$	$C_2 \otimes C_3$
$C_2 \otimes C_4$	$C_2 \otimes C_4$

Table 4.2: Order of pairwise resolving, modified scheme

## 5 Test plan

In the following chapter the result of testing MiniMarch is presented. This chapter describes how those results were acquired. A version of MiniMarch was submitted to the 2007 SAT competition and after resolving some problems a working version was run on the benchmarks from the first round of that competition. The results of those runs are analyzed here along with results of tests run on office PC's.

Troughout all tests SAT competition like scoring was used to be able to compare solvers fairly. In the following paragraphs it will be briefly explained. For a more detailed explanation along with information about the competition platform please refer to the rules for the 2007 SAT competition<sup>3</sup>. Information about the test environment used on the office PC's can be found in appendix A.

### 5.1 SAT Competition scoring

In the SAT competitions points are awarded to the competing solvers. For each benchmark points for finding a solution within the time available and points for speed are granted. For each series of benchmarks of the same type a number of points is awarded to each solver that solved at least one of the benchmarks from that series.

For all the data presented in this report speed and solution points have also been calculated in order to allow fair comparison of the solvers used. Series points are not awarded since they would have to be awarded to all solvers at all presented tests which would not be useful.

### 5.2 Solution points

For each formula a number of points, the *solution purse*, is divided equally amongst all solvers that solve that formula. The sum of all points from the *solution purse* awarded to a solver on a set of benchmarks can be found in the row *solution points* of all result tables in this document. In the 2005 SAT competition the *solution purse* was 1000 points. For this report, since only a few solvers are compared, it suffices to divide just 1 point over the solvers for each benchmark.

### 5.3 Speed points

For each formula a *speed purse* is divided unequally over all solvers that solved the problem where faster solvers are rewarded by getting more points. The sum of all points from the *speed purse* awarded to a solver on a set of benchmarks can be found in the row *speed points* of all result tables in this document. As for the *solution purse* the *speed purse* will only be 1 point throughout this document.

---

<sup>3</sup><http://www.satcompetition.org/2007/rules07.html>

To calculate the number of speed points awarded to solver  $s$  for solving problem  $p$  the *speed factor* needs to be calculated:

$$speedFactor(p, s) = \frac{timeLimit(p)}{1 + timeUsed(p, s)} \quad (5.1)$$

Where  $timeLimit(p)$  is the run time limit used for problem  $p$  and  $timeUsed(p, s)$  is the time solver  $s$  took to solve problem  $p$ .  $speedFactor(p, s)$  is 0 if solver  $s$  did not solve problem  $p$ .

The speed points to be awarded are calculated as:

$$speedAward(p, s) = speedPurse(p) * \frac{speedFactor(p, s)}{\sum_i speedFactor(p, i)} \quad (5.2)$$

## 5.4 Solvers for comparison

In all results presented in this document MiniMarch's result will be compared to that of MiniSat 2.0 and RSat [7]. MiniSat is MiniMarch's well known ancestor which has been a prize winner for a couple of years and finished third in the industrial category at the 2007 SAT competition. At that competition RSat was the winner in that category.

RSat is a MiniSat derivative as well. In RSat, if variable  $x$  which had been previously forced to assign a value becomes free again due to backtracking the old assignment is stored. If at a later stage that variable becomes a decision variable it is assigned the old value. Another important property of RSat is that it uses a restart strategy that makes the solver restart much more often than MiniSat does.

## 6 Results

### 6.1 SAT Competition

In the tables 6.1 to 6.6 results from the first round of the 2007 SAT competition are presented. There are three categories of benchmarks each with satisfiable and unsatisfiable instances. All Velev benchmarks were left out because MiniMarch will try to simplify and sort these very large formulas which is unnecessary and can not be done in reasonable time. In a future version a heuristic cut-off like the one in MiniSat is required on the clause sorter and simplifier in MiniMarch as well. Further more, only benchmarks that were solved by at least one of the three solvers under comparison were taken into account.

random SAT			
	<i>minimarch 1.1</i>	<i>minisat</i>	<i>RSat</i>
solved	82	59	29
winners	61	22	3
average time (s)	238.71	232.3	156.81
std. deviation on time	311.45	316.91	204.49
Solution purse	49.33	26.33	10.33
Speed award	56.56	23.65	5.79
Total points	<b>105.89</b>	49.98	16.12

Table 6.1: Result for 86 random satisfiable formulas

In the first table the results for the random satisfiable formulas are shown. On these formulas the branch direction heuristics in MiniMarch make it perform much better than MiniSat and RSat. For unsatisfiable satisfiable formulas for which results are presented in table 6.2 the direction heuristics have less influence and MiniMarch and MiniSat perform about equal. RSat's techniques seem not to be fit for random formulas as it performs badly on both sets of those formulas.

random UNSAT			
	<i>minimarch 1.1</i>	<i>minisat</i>	<i>RSat</i>
solved	43	42	14
winners	16	27	0
average time (s)	388.3	337.31	183.58
std. deviation on time	393.87	325.38	286.09
Solution purse	19.67	18.67	4.67
Speed award	20.28	20.16	2.55
Total points	<b>39.95</b>	<b>38.83</b>	7.22

Table 6.2: Result for 43 random unsatisfiable formulas

In table 6.3 the result for crafted satisfiable formulas is shown. Unfortu-

nately, MiniMarch does slightly worse than MiniSat. It does however still do much better than RSat which seems to be very much targeted at industrial formulas. The same goes for the results for unsatisfiable crafted formulas shown in table 6.4.

<b>crafted SAT</b>			
	<i>minimarch 1.1</i>	<i>minisat</i>	<i>RSat</i>
solved	20	22	11
winners	7	14	3
average time (s)	174.67	138.65	294.92
std. deviation on time	320.42	189.98	353.88
Solution purse	8.83	10.83	4.33
Speed award	9.19	12.1	2.71
Total points	18.02	<b>22.93</b>	7.04

Table 6.3: Result for 24 crafted satisfiable formulas

<b>crafted UNSAT</b>			
	<i>minimarch 1.1</i>	<i>minisat</i>	<i>RSat</i>
solved	47	49	26
winners	7	43	2
average time (s)	275	205	349
std. deviation on time	300.38	258.75	348.83
Solution purse	20.33	22.33	9.33
Speed award	18.28	28.71	5
Total points	38.61	<b>51.04</b>	14.33

Table 6.4: Result for 53 crafted unsatisfiable formulas

For both sets of industrial formulas shown in tables 6.5 and 6.6 MiniMarch does slightly less good than its ancestor MiniSat. As both RSat and MiniMarch are based on MiniSat these results seem to indicate that RSat narrowed MiniSat’s applicability in favor of excellent performance on industrial benchmarks while MiniMarch broadened MiniSat’s applicability.

In itself it is not useful to improve MiniSat’s performance on random benchmarks as the performance on those benchmarks will remain modest compared to that of lookahead solvers. However, MiniMarch performs nearly equal to MiniSat on industrial benchmarks despite the use of more expensive heuristics. This seems to indicate that the heuristics do indeed steer the solver in the right direction.

<b>industrial SAT</b>			
	<i>minimarch 1.1</i>	<i>minisat</i>	<i>RSat</i>
solved	26	27	36
winners	8	15	19
average time (s)	246	319	227
std. deviation on time	303.16	298.76	282.81
Solution purse	10.67	11.67	19.67
Speed award	10.56	10.47	20.96
Total points	21.23	22.14	<b>40.63</b>

Table 6.5: Result for 42 industrial satisfiable formulas

<b>industrial UNSAT</b>			
	<i>minimarch 1.1</i>	<i>minisat</i>	<i>RSat</i>
solved	60	59	58
winners	8	35	23
average time (s)	194	197	130
std. deviation on time	299.94	298.31	256.88
Solution purse	22.17	21.67	22.17
Speed award	19.51	22.59	23.9
Total points	41.68	44.26	<b>46.07</b>

Table 6.6: Result for 66 industrial unsatisfiable formulas

## 6.2 Parameter option tests

The SAT Competition version of MiniMarch might have lacked the required parameter tuning. To find the most optimal parameter settings the set of benchmarks from the SAT Race<sup>4</sup> that was held in 2006 was acquired for further testing. For reasons explained in the previous paragraph the Velev benchmarks were also left out of these tests.

Each run of MiniSat was compared to 6 runs of MiniMarch with a different parameter setting in each run. The  $\lambda$  parameter is described in paragraph 3.3. The “lookahead vars” parameter determines on which variables lookahead is done. Four options are possible: all variables (100%), no variables (0%), or all balanced variables with balanced variables defined either by equation 3.2 or by equation 3.5.

The eighty benchmarks were used as input to the solver in four different forms.

- 1) The original formula
- 2) The polarity swapped of all variables in the input formula

---

<sup>4</sup><http://fmv.jku.at/sat-race-2006>

- 3) The polarity swapped of some variables in the input formula
- 4) As 3 + clause order shuffling + variable renumbering

Different parameter settings							
$\lambda$	<i>MiniSat</i>	<i>MiniMarch 1.1</i>					
lookahead vars	-	0.5	0.5	0	1	0.5	0.5
	-	eq 3.5	eq 3.2	eq 3.2	eq 3.2	0%	100%
sat original	10.59	9.25	9.88	8.85	7.97	6.62	8.85
sat all inv	9.14	8.73	9.48	9.52	8.76	6.53	7.85
sat some inv	10.3	8.33	8.69	9.5	8.49	6.24	8.45
sat shuffled	7.55	7.14	11.59	7.4	8.54	9.23	8.56
unsat original	11.74	6.96	6.67	7.72	7.04	7.54	6.33
unsat all inv	5.69	8.19	7.67	10.32	12.4	10.87	8.87
unsat some inv	7.58	7.32	7.4	9.98	12.35	10.81	8.54
unsat shuffled	8.14	7.96	7.45	8.98	9.91	13.01	10.56
Total points	70.73	63.88	68.83	72.27	<b>75.46</b>	70.85	68.01

Table 6.7: Points scored for different forms of the input formulas

Judging from the results in table 6.7 the definition of balanced variables by equation 3.2 is clearly the better one. On the original benchmarks MiniSat still performs the strongest but MiniMarch with  $\lambda = 1$  using equation 3.2 for balanced variables scores the most points overall. It is surprising to see that under the heaviest shuffling the 0% lookahead versions, using the static occurrence based heuristics presented in paragraph 3.4 for all variables, does rather well. It can probably be concluded from that the number of variables defined as balanced might still be made smaller.

Both lookahead criteria ( $\lambda = 0$ ,  $\lambda = 1$ ) seem to work. The version that scores the second highest number of points uses the combined criterium ( $\lambda = 0.5$ ) and has the advantage of a much smaller deviation of performance. The deviation over the four different versions is still quite large for MiniMarch which seems to indicate some more work is needed to make it more stable. As described in paragraph 2.4 the solver is still influenced by shuffling because of the way literals in a clause are sorted. This is the most important vulnerability to input formula shuffling and it must be resolved in a future version of MiniMarch.

## 7 Conclusion

MiniMarch is a SAT solver with a performance that for all benchmarks is at least close to its ancestor MiniSat's while drastically improving performance on satisfiable random benchmarks. The direction heuristics are strong enough to steer the solver in the right direction and yet not too expensive to degrade performance in general. After making a different choice for the definition of balanced variables MiniMarch's performance is now even better than it was at the SAT Competition making it ever more competitive for the future.

MiniMarch performs better on shuffled benchmarks than MiniSat does which was one of the main design goals. On the down side, the difference in performance on the original or shuffled benchmarks might have gotten smaller, they are still quite large. This is largely caused by the variable index based literal sorting within clauses that was inherited from MiniSat. In a future version this should get attention as it influences which variables are watched and therefore how the solver progresses.

A larger gain from the lookahead on balanced variables should be possible by further improving the definition of balanced variables. The number of variables that is regarded as balanced by definition 3.2 is probably still often on the high side.

MiniMarch is a good step in the direction of building a high performance conflict driven solver without depending heavily on encoding specific heuristics. Some modifications or improvements for future versions have been suggested throughout this text and we feel confident that those will make MiniMarch what it was always intended to be. Fast in general, and hardly influenced by shuffling of the input formula in particular.

## 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] Eén N. and Biere A. (2005), *Effective Preprocessing in SAT through Variable and Clause Elimination*, proceedings of SAT'05
- [3] Eén N. and Sörensson N. (2003), *An extensible sat solver*, In Proc. of the 6th Int. Conference on Theory and Applications of Satisfiability Testing
- [4] Heule M. (2004), *March: Towards a lookahead solver for general purposes*, Master's thesis, Delft University of Technology
- [5] Kibria R.H. (2006), *Actin solver description*, submitted to SAT Race
- [6] Mpekas D., van Vlaardingen M. and Wieringa S. (2006), *The first steps to a hybrid SAT solver*, Delft University of Technology
- [7] Pipatsrisawat K., Darwiche A. (2006), *A lightweight component caching scheme for satisfiability solvers*, University of California, Los Angeles
- [8] Li C.M., Anbulagan (1997), *Look-Ahead versus Look-Back for Satisfiability Problems*. Springer-Verlag, LNCS 1330, pages 342-356, Autriche
- [9] Cook S.A. (1971), *The Complexity of Theorem Proving Procedures*. In Proceedings of 3rd ACM Symposium on Theory of Computing, pp. 151-158, Ohio

## A Office PC test environment

For the purpose of testing my current projects I make use of the computational power of the PC's at the office in Delft where I am working. There are three equal Pentium IV machines on 2.93 GHz that used to be idle most of the time.

An old AMD 900 Mhz machine running Debian Linux was hooked up to the network to “handout” tests to the office machines. In fact, it is only assigned the task of running a Samba server (or in other words sharing its harddisk). On the harddisk there are three directories with test scripts, one for each “client” machine.

The three Pentium IV's normally run Windows but for this purpose they are booted from a CD-Rom. The CD-Rom contains a Slax Linux ISO which is a small Linux system specifically designed to boot from CD. I stripped most of the modules that come with Slax off and added a Samba client module. At boot time the disk on the AMD machine is mounted on the client machine which then finds its own directory with test scripts and starts executing them.

All test scripts check what they have already done and what is still left to do. In that way, the other people working at the office can reboot the machine into Windows whenever they like and whenever they stop working they help me by rebooting the machine from the CD-Rom and it will continue where it left off.

As the office network is not under a heavy load and attention is paid that no two computers use the same file the delays caused by network file i/o should remain constant on average.