

Enhancing UnitMarch with Alternative Local  
Search Methods and Unit Propagation  
Delft University of Technology

Danny Roest 1150510  
Ferdinand Grootenboers 1149911  
Menno Lodder 1100068

July, 2007

## Abstract

This paper describes our efforts to improve the UnitMarch [4] SAT solver by combining the ideas of other SAT solvers with the UnitMarch algorithm. GSAT and WalkSAT are used as an inspiration for improving UnitMarch. We also use a combination of GSAT and WalkSAT, called RandomSAT. The idea behind our modification of UnitMarch is taking large steps towards a solution with an additional algorithm and then using UnitMarch for smaller steps. With this approach we attempt to combine the best of both. The paper also describes pre-processing using unit propagation, which showed to be a significant improvement.

## 1 Introduction

The Boolean satisfiability problem (the SAT problem) is the problem of deciding for a Boolean formula if the variables can be assigned in such a way that it evaluates to true, in other words that the formula is satisfiable. The SAT problem was the first problem proved to be NP-Complete[1]. This means that it is computationally hard to find a solution for the problem, but that verifying an assignment is in fact a solution is relatively easy. For the SAT problem it is easy to show the formula is satisfied when given a set of assignment of the variables.

The most common way to describe a Boolean logic problem is in Conjunctive Normal Form (CNF). A Boolean problem in CNF consists of a conjunction of clauses, where each clause is a disjunction of literals and each literal is a Boolean variable or its negation. Formula 1 shows an example of a CNF formula. All problems in Boolean logic can be rewritten to CNF.

$$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_4) \tag{1}$$

Since the SAT problem is NP-Complete, time restrictions prevent programs (called SAT-solvers) from doing a complete search on big problems. Therefore some solvers use local search techniques to find satisfying assignments for the formulas. Local search solvers move through the search space trying to improve the solution by making small changes at the time. Most SAT-solvers look at the number of clauses that is satisfied by an assignment and try to use local improvements to increase the number of satisfied clauses. Because a local search solver does not search the complete problem, it can not prove that a problem does not have a satisfying assignment (that it is UNSAT). It can only provide an assignment of the variables to prove it is satisfiable (SAT). Even when a problem is SAT the local search solver might not find the assignment and will be indecisive.

One of the local search solvers is UnitMarch[4], which is based on the successful UnitWalk[2, 3]. UnitWalk uses unit propagation, a method where clauses consisting of only one literal, unit clauses, decide the value of the variable in that literal. This value can then be propagated to other clauses, possibly creating other unit clauses. UnitMarch uses the same principle, but instead of looking at one solution at the time, it explores 32 solutions. By doing this, UnitMarch exploits the fact that computers use 32 bits in their calculations.

We have modified UnitMarch by adding preprocessing of the SAT problems and by alternating the UnitMarch algorithm with several greedy algorithms.

Section 2 explains the current UnitMarch algorithm, and how we have added preprocessing. Section 3 describes how we have added alternative algorithms to UnitMarch. Section 4 shows our test results and Section 5 describes our conclusions and possible future work.

## 2 UnitMarch

In this section we will shortly describe the UnitMarch solver [4]. This solver is based on the UnitWalk solver[2], which is described first. After this we describe how UnitMarch uses UnitWalk. Finally in Section 2.3 preprocessing using unit propagation is described.

### 2.1 UnitWalk

UnitWalk starts with an random assignment and modifies it step by step. The algorithm works in periods. In each period one or more variables are flipped. In each step a variable  $v$  in the current formula is given a truth value. If there are unit clauses (a unit clause is a clause with only one literal) then  $v$  is chosen from the unit clauses. If the value of  $v$  does not satisfy the unit clause or another unit clauses,  $v$  is flipped. If there are no unit clauses then  $v$  is the chosen from a previously decided ordering of the variables. When all variables have been processed a period ends. If no variable was flipped during the last period a random variable is chosen to flip. When the period has ended a new ordering chosen, the current formula is replaced and a new period is started. It is possible to set the parameters MAX\_PERIODS and MAX\_TRIES, which mean respectively the maximum number of periods per try and the maximum number of tries to get to a solution. If MAX\_PERIODS is reached a new random assignment is generated and a new try is started, this is needed because UnitWalk can end up in an infinite loop.

## 2.2 Multi-bit Processing in UnitMarch

The basic idea is that UnitMarch[4] is a parallel version of UnitWalk[2]. Consider the following functions on three variables:

```
x := 0 1 0 0 1 1 0 1
y := 0 0 1 0 1 0 1 1
z := 0 0 0 1 0 1 1 1
```

This can be considered as an eight parallel Boolean assignment on the variables. Primitive operations like AND and NOT can be done on these Boolean assignments in parallel, which only cost one CPU cycle to process the eight assignments. In UnitMarch 32-bit assignments are used. UnitMarch uses the UnitWalk Sat solver, which was explained in Section 2.1.

To explain how UnitMarch works, we give an example:

```
 $\varphi_{\text{master}}$  := { $x_1 = 0110, x_2 = 1100, x_3 = 1010, x_4 = 0110$ }
 $\varphi_{\text{active}}$  := { $x_1 = ****, x_2 = ****, x_3 = ****, x_4 = ****$ }
 $\pi$  := ( $x_2, x_1, x_4, x_3$ )
```

Here is  $\pi$  the ordering and  $\varphi_{\text{master}}$  is the initial (random) valuation. The algorithm picks the first variable, in this case  $x_2$ , and assigns to the value in  $\varphi_{\text{master}}$  followed by unit propagation. This results in two unit clauses:

$$\begin{aligned} (x_1 = **** \vee x_2 = 1100) &\Rightarrow x_1 := **11 \\ (\neg x_2 = 0011 \vee \neg x_3 = ****) &\Rightarrow x_3 := 00** \end{aligned}$$

Now one of them,  $x_1$ , is selected and its value is assigned which results in assigning more bits in  $x_3$ :

$$(\neg x_1 = **00 \vee x_2 = 1100 \vee x_3 = 00**) \Rightarrow x_3 := 0011$$

After  $x_3$  is assigned, three other clauses are triggered:

$$\begin{aligned} (\neg x_2 = 0011 \vee x_3 = 0011 \vee \neg x_4 = ****) &\Rightarrow x_4 := 00** \\ (\neg x_2 = 0011 \vee x_3 = 0011 \vee x_4 = 00**) &\Rightarrow \mathbf{bit\text{-}conflict} \\ (\neg x_3 = 1100 \vee \neg x_4 = 11**) &\Rightarrow x_4 := 0000 \end{aligned}$$

After the unit propagation is done, the bits that are not set, are assigned the value from  $\varphi_{\text{master}}$ . Duplicate assignments are detected and a possible double is given a random assignment.

## 2.3 Preprocessing Using Unit Propagation

The total execution time of the UnitMarch algorithm depends among other things on the number of clauses and variables, so it is a good idea to do some preprocessing before executing the algorithm.

To do preprocessing we used unit propagation to minimize the number of clauses and variables. The basic idea is to remove useless literals and clauses that can be left out so that UnitMarch has fewer clauses and variables to solve.

The algorithm works as follows. First the clauses with only one literal (variable which is positive or negative) are selected. These are called *unit clauses* and the literal in it is called a *unit literal*. A unit literal must be positive, otherwise the problem can not be satisfied. Every clause that contains a unit literal is removed, because the clause is already satisfied (because of the unit literal). From all other clauses containing the negation of the unit literal, the literal is removed from the clause. This is done because that literal can never make the clause positive, thus can be left out. If a modified clause now became a clause with only one literal, this clause is added to the unit clauses and also processed.

During the execution of UnitMarch the variables removed by unit propagation can be ignored. When the algorithm is finished and a solution is found, the removed variables can be inserted into the solution.

## 3 Greedy, WalkSAT and RandomWalk

Trying to improve the UnitMarch algorithm we added three alternatives to the program: Greedy[7], WalkSAT[6] and a combination of the two. Because the UnitMarch algorithm is working parallel on a specified number of assignments and the alternative algorithms are not, we have extract each instance before we can start. This means that if UnitMarch is running on 32 bits, we have to extract of the 32 assignments, and run our alternative algorithm 32 times on a single assignment. When the alternative algorithm has finished, we have to place the instance back into the 32-bits instance.

In the following subsections we will describe the three alternatives.

### 3.1 Greedy

The first part we added to improve the UnitMarch algorithm, is the Greedy algorithm based on GSAT [7]. The Greedy algorithm searches for the variable with the highest score and will flip it. The score of a variable is defined as the the number of clauses that will become satisfied when being flipped minus the number of clauses that will become unsatisfied when being flipped. So the variable with the highest

score can be said to be the variable which has the biggest positive effect towards a solution if it is flipped. Because we think the name *score* does not reflect the meaning of it well, we will call it *flip effect* from now on.

Each variable keeps track of its flip effect and the algorithm flips the variable with the highest flip effect. To efficiently find the highest flip effect we use a heap data structure. Each time a variable is flipped, the flip effect of the flipped variable and those of the variables that exist in the same clauses of the flipped variable can be modified. So only these variables have to be updated in the heap. When we want to retrieve the variable with the best flip effect we can just take the first in the heap.

We choose to go back to UnitMarch when there is no improvement between two successive tries of the Greedy algorithm. We also experimented with a parameter called MAX\_GREEDY\_TRIES but we set this parameter to infinity, because we want Greedy to go on until there is no improvement.

We added two parameters to control the Greedy part, namely START\_GREEDY and INTERVAL\_GREEDY. The first one defines the UnitMarch period in which the Greedy algorithm will be executed for the first time. The second parameter defines after how much UnitMarch periods the Greedy algorithm will be executed. The parameters can have a great influence on the time to come to a solution.

### 3.2 WalkSAT

The second alternative is based on the WalkSAT algorithm [6]. WalkSAT takes a random unsatisfied clause and then flips the variable in that clause with the highest flip effect. The idea for this alternative comes from the method UnitWalk used to integrate WalkSAT [2]. They conclude that UnitWalk and WalkSAT are in fact opposites and that a combination of these two gives even better results. The method performs a run of the WalkSAT algorithm when there are enough (according to a certain threshold) opposite clauses in the instance of the UnitWalk phase and the number of opposite clauses is not decreasing.

Again we had to choose an exit condition on which we go back to UnitMarch. After testing some conditions we have chosen the same exit condition as UnitWalk[2]:

$$(n^2/2) \tag{2}$$

We use the same control options as we did for the Greedy alternative to alternate between UnitMarch and WalkSAT. So we can define the first period in which WalkSAT has to run and we can define the interval between two runs of WalkSAT. We do not take into account

the number of opposite clauses, because UnitMarch just ignores these and therefore we are not able to count the opposites.

An alternative option in the WalkSAT algorithm is the exit condition it uses. Because UnitMarch takes big steps trying to satisfy the formula in all bits and WalkSAT takes little steps, we think we have to exit the WalkSAT algorithm when it moves too much away from the solution it has as input. We calculate the hamming distance between the input solution and the current solution after every step and exit the algorithm for the current bit, when the hamming distance equals a threshold.

### 3.3 Combinations of Greedy and WalkSAT

As the third alternative we combined Greedy and WalkSAT. This combination can be made in different ways. We have used the method of RandomWalk [5] in which, per try, the Greedy algorithm is executed with a chance of  $p$  and WalkSAT is executed with a chance of  $1-p$ .

Another approach is a method to combine them in parallel. We could for example run the Greedy alternative on 16 of 32 assignments and run the WalkSAT algorithm on the other 16 assignments. With this approach the algorithms can be used next to each other.

Because we had a time limit and because initial tests did not show us any improvement, we did not further test this alternative.

## 4 Comparing the Methods

This Section describes how we have tested the solving strategies given above, see Section 2.3 and 3. We start with explaining our benchmark set and then give the results of our tests.

### 4.1 Benchmarks

To test our solver we have constructed a benchmark set of 44 CNF benchmarks. Our benchmark set consists of the set used by Heule et al. in the original UnitMarch paper[4] combined with a series of larger, more difficult test cases (*9vliw\_bp\_mc\_bug*) created by Velev[8]. The benchmarks and a description can be found Velev's homepage<sup>1</sup> (*dlx* and *9vliw*) and SatLib<sup>2</sup>(all other benchmarks). Our benchmark set contains a wide range in number of variables and clauses, both random and non-random, see Table 1.

---

<sup>1</sup><http://www.ece.cmu.edu/~mvelev>

<sup>2</sup><http://www.satlib.org/>

| Test                   | Variables | Clauses | Unit Clauses | Random |
|------------------------|-----------|---------|--------------|--------|
| 9vliw_bp_mc_bug012.cnf | 18080     | 273820  | 1            | No     |
| 9vliw_bp_mc_bug025.cnf | 19833     | 280523  | 1            | No     |
| 9vliw_bp_mc_bug037.cnf | 19539     | 274385  | 1            | No     |
| 9vliw_bp_mc_bug059.cnf | 19653     | 278661  | 1            | No     |
| 9vliw_bp_mc_bug074.cnf | 17346     | 232880  | 1            | No     |
| 9vliw_bp_mc_bug098.cnf | 20084     | 287926  | 1            | No     |
| aim-200-2_0-yes1-1.cnf | 200       | 400     | 0            | Yes    |
| aim-200-2_0-yes1-2.cnf | 200       | 400     | 0            | Yes    |
| aim-200-2_0-yes1-3.cnf | 200       | 400     | 0            | Yes    |
| aim-200-2_0-yes1-4.cnf | 200       | 400     | 0            | Yes    |
| aim-200-3_4-yes1-1.cnf | 200       | 680     | 0            | Yes    |
| aim-200-3_4-yes1-2.cnf | 200       | 680     | 0            | Yes    |
| aim-200-3_4-yes1-3.cnf | 200       | 680     | 0            | Yes    |
| aim-200-3_4-yes1-4.cnf | 200       | 680     | 0            | Yes    |
| bw_large.b.cnf         | 1087      | 13772   | 0            | No     |
| bw_large.c.cnf         | 3016      | 50457   | 0            | No     |
| dlx2_cc_bug17.cnf      | 2388      | 22101   | 1            | No     |
| dlx2_cc_bug39.cnf      | 1482      | 12112   | 1            | No     |
| dlx2_cc_bug40.cnf      | 1520      | 12811   | 1            | No     |
| flat200-24.cnf         | 600       | 2237    | 0            | No     |
| flat200-39.cnf         | 600       | 2237    | 0            | No     |
| flat200-48.cnf         | 600       | 2237    | 0            | No     |
| flat200-5.cnf          | 600       | 2237    | 0            | No     |
| flat200-64.cnf         | 600       | 2237    | 0            | No     |
| logistics.a.cnf        | 828       | 6718    | 0            | No     |
| logistics.b.cnf        | 843       | 7301    | 0            | No     |
| logistics.c.cnf        | 1141      | 10719   | 0            | No     |
| logistics.d.cnf        | 4713      | 21991   | 0            | No     |
| par16-1.cnf            | 1015      | 3310    | 76           | No     |
| par16-2.cnf            | 1015      | 3374    | 76           | No     |
| par16-3.cnf            | 1015      | 3344    | 76           | No     |
| par16-4.cnf            | 1015      | 3324    | 76           | No     |
| par16-5.cnf            | 1015      | 3358    | 76           | No     |
| qg1-08.cnf             | 512       | 148957  | 29           | No     |
| qg2-08.cnf             | 512       | 148957  | 29           | No     |
| qg3-08.cnf             | 512       | 10469   | 141          | No     |
| qg4-09.cnf             | 729       | 15580   | 37           | No     |
| qg5-11.cnf             | 1331      | 64054   | 56           | No     |
| qg7-13.cnf             | 2197      | 97072   | 79           | No     |
| uf250-054.cnf          | 250       | 1065    | 0            | Yes    |
| uf250-062.cnf          | 250       | 1065    | 0            | Yes    |
| uf250-071.cnf          | 250       | 1065    | 0            | Yes    |
| uf250-072.cnf          | 250       | 1065    | 0            | Yes    |
| uf250-093.cnf          | 250       | 1065    | 0            | Yes    |

Table 1: Details of the benchmark set

All benchmarks in this paper were ran on a dual-core Intel(R) Pentium(R) 4 CPU 3.06GHz under Linux taking the average of 20 benchmark runs on different seeds. It is important to note that the same local search solver with a different seed can have vastly different results. Also it is possible to optimize solvers for specific benchmarks, in this paper we have tried to get an good overall performance.

## 4.2 Unit Clause Propagation

Table 2 contains a comparison of running the UnitMarch algorithm with and without preprocessing using unit clause propagation.

When there are a large number of variables or clauses, preprocessing improves the time of the greedy algorithms significantly. Our implemented version of preprocessing is not optimal, because it was just

| Test                   | Variables removed | Clauses removed | Time before | Time after | % Faster |
|------------------------|-------------------|-----------------|-------------|------------|----------|
| 9vliw_bp_mc_bug012.cnf | 4 (0.02%)         | 29 (0.01%)      | 145.49      | 84.23      | 42.11    |
| 9vliw_bp_mc_bug025.cnf | 4 (0.02%)         | 29 (0.01%)      | 101.40      | 66.15      | 34.76    |
| 9vliw_bp_mc_bug037.cnf | 4 (0.02%)         | 29 (0.01%)      | 281.90      | 270.47     | 4.05     |
| 9vliw_bp_mc_bug059.cnf | 4 (0.02%)         | 29 (0.01%)      | 8.00        | 12.93      | -61.63   |
| 9vliw_bp_mc_bug074.cnf | 4 (0.02%)         | 29 (0.01%)      | 323.28      | 267.60     | 17.22    |
| 9vliw_bp_mc_bug098.cnf | 4 (0.02%)         | 29 (0.01%)      | 215.41      | 63.28      | 70.62    |
| dlx2_cc_bug17.cnf      | 11 (0.46%)        | 75 (0.34%)      | 0.12        | 0.14       | -16.67   |
| dlx2_cc_bug39.cnf      | 8 (0.54%)         | 41 (0.34%)      | 0.81        | 0.2        | 75.31    |
| dlx2_cc_bug40.cnf      | 5 (0.33%)         | 16 (0.12%)      | 0.12        | 0.17       | -41.67   |
| par16-1.cnf            | 408 (40.20%)      | 1466 (44.29%)   | 0.54        | 0.37       | 31.48    |
| par16-2.cnf            | 383 (37.73%)      | 1416 (41.97%)   | 4.41        | 1.6        | 63.72    |
| par16-3.cnf            | 395 (38.92%)      | 1440 (43.06%)   | 1.65        | 2.4        | -45.45   |
| par16-4.cnf            | 396 (39.01%)      | 1442 (43.38%)   | 2.11        | 0.42       | 80.09    |
| par16-5.cnf            | 388 (38.23%)      | 1426 (42.47%)   | 8.06        | 3          | 62.78    |
| qg1-08.cnf             | 231 (45.12%)      | 125193 (84.05%) | 107.93      | 36.46      | 66.22    |
| qg2-08.cnf             | 216 (42.19%)      | 123700 (83.04%) | 542.12      | 195.81     | 63.88    |
| qg3-08.cnf             | 239 (46.68%)      | 7640 (72.98%)   | 0.04        | 0.01       | 75.00    |
| qg4-09.cnf             | 294 (40.33%)      | 9102 (58.42%)   | 0.63        | 0.25       | 60.32    |
| qg5-11.cnf             | 507 (38.09%)      | 35566 (55.53%)  | 4.36        | 2.08       | 52.29    |
| qg7-13.cnf             | 785 (35.73%)      | 51105 (52.65%)  | 166.44      | 151.72     | 8.84     |

Table 2: The effect of unit propagation

for testing how much preprocessing influences the time of the Unit-March algorithm. For the largest instance we tested (*9vliw*) the preprocessing took only 0.05 seconds. For the instance that removed most clauses (*qg1-08*) preprocessing took 22.77 seconds.

### 4.3 Alternation between UnitMarch and the Alternatives

The parameters we noted in Section 3 can have a big influence on the results of the benchmarks. All alternatives use the parameters `START_GREEDY` and `INTERVAL_GREEDY` to determine the alternation between the UnitMarch phase and the alternative phase.

The `INTERVAL_GREEDY` parameter determines the number of periods UnitMarch will run before the program switches to one of the alternatives. The `START_GREEDY` parameter determines the first period one of the alternatives will run. For example, if we set `START_GREEDY=10` and `INTERVAL_GREEDY=20`, then the alternative will run in periods 10, 30, 50, etc.

It is hard to find a *best* value for this parameters, but after some testing we found it was expensive (in time) to start the program with an alternative. The (serial) alternative methods are quite expensive in comparison with the (parallel) UnitMarch method, so assignments of a SAT problem that can be solved by UnitMarch in relatively few periods should not use an alternative, because that will give a significant overhead. To run an alternative at the very end of the execution is also not a good idea, because then it would not contribute a lot to obtaining a solution. After performing some tests we concluded that `START_GREEDY` set to half of the interval gives the best results:

$$START\_GREEDY = 0.5 * INTERVAL\_GREEDY \quad (3)$$

The interval between two alternative runs must not be too small, also because of its relatively expensiveness. Therefore we made the decision to set `INTERVAL_GREEDY=200` and `START_GREEDY=100`. The `INTERVAL_GREEDY` parameter is chosen so that the interval is not too large and not too small (according to the benchmarks) and the `START_GREEDY` parameter is chosen according to Formula 3.

#### 4.4 Original Method, Greedy or WalkSAT

After deciding when and how often the alternative search method should be ran, we have ran exhaustive benchmarks for four different versions of UnitMarch, all versions have preprocessing as described in Section 2.3 turned on. All results are average values of 20 runs. For the alternation of UnitMarch with the alternative algorithm we have chosen to start the alternative algorithm after 100 periods of UnitMarch and repeat it every 200 periods (see Section 3). This means that if the algorithm was solved in less than 100 periods, no alternation has been used so all alternatives will give the same results.

We have tested the following versions of UnitMarch ( $n$  denotes the number of variables in the benchmark):

- 1 UnitMarch alternated with Greedy, see Section 3.1.
- 2 UnitMarch alternated with WalkSAT (exit condition  $n/4$ ), see Section 3.2.
- 3 UnitMarch alternated with WalkSAT (exit condition  $n^2/2$ ), see Section 3.2.
- 4 Only UnitMarch.

Table 3 shows the results of our test. The first part denotes the average time it took each solver to find a satisfying assignment. The second part denotes the average number of periods UnitMarch has used. The third part shows the percentage of time that was spent in the alternative solving method.

Table 3 also shows that 9 of the 44 tests did not perform any alternative solving method, we have ignored those tests when drawing our conclusions. The bottom row in Table 3 shows the number of times a particular method was fastest, ignoring the 9 results that did not perform any alternative solving method. When two versions scored the same, both are counted as the winner. Our tests show that the Greedy and WalkSAT alternatives reduce the number of periods that UnitMarch needs. From this we can conclude that the alternative search methods help UnitMarch to find the solution. The times show that the original UnitMarch and our Greedy version perform best. The

| Test                   | Time(s) |        |        |        | Periods of UnitMarch |         |          |         | Time(%) in alternative solver |       |       |
|------------------------|---------|--------|--------|--------|----------------------|---------|----------|---------|-------------------------------|-------|-------|
|                        | 1       | 2      | 3      | 4      | 1                    | 2       | 3        | 4       | 1                             | 2     | 3     |
| 9vliw_bp_mc_bug012.cnf | 82.71   | 58.31  | 58.75  | 46.63  | 130.35               | 122.7   | 122.7    | 123.55  | 7.79                          | 8.16  | 8.35  |
| 9vliw_bp_mc_bug025.cnf | 82.1    | 73.84  | 72.93  | 58.53  | 148.55               | 135     | 132      | 154.75  | 7.5                           | 7.85  | 7.97  |
| 9vliw_bp_mc_bug037.cnf | 97.93   | 271.87 | 96.64  | 63.27  | 182.05               | 432.9   | 176.7    | 168.45  | 12.86                         | 14.72 | 13.38 |
| 9vliw_bp_mc_bug059.cnf | 9.08    | 9.05   | 9.05   | 9.06   | 22.2                 | 22.2    | 22.2     | 22.2    | 0                             | 0     | 0     |
| 9vliw_bp_mc_bug074.cnf | 96.85   | 112.69 | 103.28 | 63.76  | 200.55               | 234.15  | 208      | 193.1   | 15.61                         | 15.63 | 16.71 |
| 9vliw_bp_mc_bug098.cnf | 77.5    | 78.28  | 74.42  | 53.46  | 139.15               | 146.3   | 136.15   | 136.45  | 10.02                         | 8.9   | 9.7   |
| aim-200-2_0-yes1-1.cnf | 0.09    | 0.1    | 0.1    | 0.1    | 301.3                | 330.3   | 300.4    | 323.5   | 1.37                          | 3.23  | 5.33  |
| aim-200-2_0-yes1-2.cnf | 3.02    | 1.34   | 3.44   | 1.29   | 9717.35              | 4071.25 | 10151.65 | 4258.95 | 2.23                          | 6.45  | 10.37 |
| aim-200-2_0-yes1-3.cnf | 0.04    | 0.04   | 0.04   | 0.04   | 135.05               | 138.6   | 128.25   | 130.55  | 0.64                          | 0.95  | 1.29  |
| aim-200-2_0-yes1-4.cnf | 0.07    | 0.07   | 0.06   | 0.08   | 205.35               | 204.45  | 170.55   | 248.85  | 1.16                          | 1.85  | 4.04  |
| aim-200-3_4-yes1-1.cnf | 0.72    | 0.82   | 0.92   | 0.96   | 1082.35              | 1193.35 | 1298.85  | 1497.8  | 1.64                          | 4.14  | 6.63  |
| aim-200-3_4-yes1-2.cnf | 2.61    | 2.62   | 2.54   | 3.77   | 4009.45              | 3781.55 | 3603.55  | 5935.75 | 1.83                          | 5.2   | 8.39  |
| aim-200-3_4-yes1-3.cnf | 0.19    | 0.25   | 0.22   | 0.19   | 283.95               | 364.6   | 323.45   | 297.5   | 0.78                          | 2.38  | 3.99  |
| aim-200-3_4-yes1-4.cnf | 3.05    | 2.37   | 2.13   | 2.04   | 4547.1               | 3478.85 | 2997     | 3220.05 | 1.76                          | 5.09  | 7.83  |
| bw_large.b.cnf         | 0.06    | 0.05   | 0.06   | 0.06   | 10.05                | 10.05   | 10.05    | 10.05   | 0                             | 0     | 0     |
| bw_large.c.cnf         | 5.86    | 6.53   | 4.89   | 5.7    | 259.6                | 277.65  | 201.9    | 271.05  | 3.56                          | 5.45  | 7.64  |
| dlx2_cc_bug17.cnf      | 0.08    | 0.08   | 0.08   | 0.08   | 3.25                 | 3.25    | 3.25     | 3.25    | 0                             | 0     | 0     |
| dlx2_cc_bug39.cnf      | 0.12    | 0.12   | 0.12   | 0.12   | 10.5                 | 10.5    | 10.5     | 10.5    | 0                             | 0     | 0     |
| dlx2_cc_bug40.cnf      | 0.13    | 0.13   | 0.13   | 0.13   | 10.9                 | 10.9    | 10.9     | 10.9    | 0                             | 0     | 0     |
| flat200-24.cnf         | 0.09    | 0.09   | 0.1    | 0.09   | 70                   | 70      | 75.5     | 74.25   | 0.12                          | 0.24  | 0.55  |
| flat200-39.cnf         | 0.17    | 0.15   | 0.17   | 0.16   | 120.05               | 109.75  | 118.25   | 118.6   | 0.28                          | 0.84  | 1.33  |
| flat200-48.cnf         | 0.08    | 0.08   | 0.08   | 0.08   | 57.75                | 55.6    | 55.8     | 55.85   | 0.12                          | 0.28  | 0.47  |
| flat200-5.cnf          | 0.33    | 0.35   | 0.36   | 0.35   | 242.7                | 243.8   | 248.1    | 255.5   | 0.91                          | 1.98  | 3.44  |
| flat200-64.cnf         | 0.12    | 0.11   | 0.14   | 0.11   | 81.95                | 81.45   | 96       | 81.15   | 0.13                          | 0.27  | 0.38  |
| logistics.a.cnf        | 6.37    | 7.85   | 8.16   | 6.69   | 2180.2               | 2588.6  | 2588.6   | 2415.55 | 5.34                          | 8.84  | 11.85 |
| logistics.b.cnf        | 0.16    | 0.16   | 0.16   | 0.16   | 56.7                 | 56.7    | 56.7     | 56.7    | 0                             | 0     | 0     |
| logistics.c.cnf        | 1.83    | 1.66   | 1.54   | 1.76   | 371.75               | 342.1   | 306.4    | 388.65  | 3.08                          | 4.16  | 4.99  |
| logistics.d.cnf        | 0.24    | 0.24   | 0.24   | 0.24   | 8.65                 | 8.65    | 8.65     | 8.65    | 0                             | 0     | 0     |
| par16-1.cnf            | 0.14    | 0.15   | 0.26   | 0.17   | 174.2                | 178.65  | 280.35   | 223.65  | 1.36                          | 4.45  | 6.63  |
| par16-2.cnf            | 1.47    | 1.32   | 1.99   | 1.56   | 1626.75              | 1343.7  | 1822.3   | 1822.6  | 3.84                          | 12.33 | 19.34 |
| par16-3.cnf            | 1.15    | 0.88   | 0.95   | 0.82   | 1394.65              | 964.85  | 964.85   | 1047.6  | 3.26                          | 10.6  | 16.28 |
| par16-4.cnf            | 0.6     | 0.48   | 0.52   | 0.61   | 704.65               | 526.85  | 526.85   | 753.3   | 2.73                          | 8.05  | 12.43 |
| par16-5.cnf            | 1.55    | 3.24   | 3.52   | 2.15   | 1939.85              | 3664.2  | 3664.2   | 2884.95 | 4.25                          | 13.41 | 20.34 |
| qg1-08.cnf             | 43.97   | 47.55  | 52.09  | 36.22  | 1737.7               | 1901.45 | 2092     | 1363.2  | 5.25                          | 8.43  | 10.8  |
| qg2-08.cnf             | 285.79  | 308.05 | 365.88 | 323.7  | 19307.4              | 19355.1 | 21846.9  | 25036.7 | 10.62                         | 15.03 | 19.88 |
| qg3-08.cnf             | 0.07    | 0.07   | 0.07   | 0.07   | 5.9                  | 5.9     | 5.9      | 5.9     | 0                             | 0     | 0     |
| qg4-09.cnf             | 0.29    | 0.29   | 0.29   | 0.29   | 49.55                | 49.55   | 49.55    | 49.55   | 0                             | 0     | 0     |
| qg5-11.cnf             | 2.94    | 2.93   | 2.93   | 2.98   | 63.8                 | 65.2    | 63.6     | 64.75   | 0.23                          | 0.38  | 0.51  |
| qg7-13.cnf             | 124.53  | 138.47 | 156.72 | 185.67 | 3359.2               | 3629.1  | 3963.55  | 5439.05 | 5.92                          | 8.79  | 11.54 |
| uf250-054.cnf          | 2.59    | 4.4    | 4.53   | 2.15   | 2851                 | 4605    | 4605     | 2417.65 | 1.83                          | 4.83  | 7.74  |
| uf250-062.cnf          | 1.45    | 1.98   | 2.04   | 3      | 1586.45              | 2066.3  | 2066.3   | 3315.75 | 2.1                           | 5.32  | 8.6   |
| uf250-071.cnf          | 3.78    | 2.77   | 2.88   | 3.03   | 4181.55              | 2942.55 | 2942.55  | 3454.8  | 2.05                          | 5.4   | 8.35  |
| uf250-072.cnf          | 1.89    | 2.64   | 2.64   | 2.1    | 2004.65              | 2619.7  | 2619.7   | 2338.15 | 2.01                          | 4.95  | 8     |
| uf250-093.cnf          | 1.56    | 3.15   | 3.24   | 1.6    | 1691.95              | 3269.55 | 3269.55  | 1790.85 | 1.76                          | 4.98  | 7.73  |
| Winning solution       | 13      | 9      | 6      | 13     | 12                   | 9       | 14       | 5       |                               |       |       |

Table 3: Benchmark results(20 seeds) for Greedy(1), WalkSAT  $n/4$  (2), WalkSAT  $n^2/2$  (3), Original UnitMarch(4), all with preprocessing turned on.

WalkSAT versions are a bit slower. Reason for WalkSAT being slower is because it generally performs more iterations than greedy, and our implementation might not be as effective as possible.

One thing that stands out in our results is that solvers 2 and 3 (both versions of WalkSAT) have many equal results, for instance on the `uf_250` family. The solver has taken a different amount of time to solve these assignments, but has done precisely the same amount of periods on each instance. We are not sure what causes this, but a possibility is that WalkSAT is likely to enter a loop, so more steps in the WalkSAT part do not progress the solution.

We have also tested four versions with WalkSAT having an exit condition based on the hamming distance of the solution to the start of the WalkSAT period, see Section 3.2. We have tested them for a hamming distance of  $n/40$ ,  $n/20$ ,  $n/10$  and  $n/5$ . However, for the bigger problems (over 300 variables) these all ran extremely slow, so we could not complete the test. For smaller problems the number of periods in UnitMarch was comparable to the versions of WalkSAT discussed above, and the time used was significantly higher.

The last thing to note about the tests with WalkSAT is the performance on random 3-SAT instances. In UnitWalk, this switch to WalkSAT is done to improve performance on random 3-SAT problems and it is successful. It is therefore strange this method is not working for UnitMarch. One of the possibilities could be that UnitWalk counts the number of conflicts and then decides to switch to WalkSAT. We could not implement this test, because conflicts in UnitMarch are just ignored.

## 5 Conclusions and Future Work

Combining UnitMarch with other algorithms lead to some improvements in the number of periods used. Unfortunately the total time to solve an instance did not improve significantly. We can also conclude that UnitMarch and the other algorithms did not always help each other. Sometimes the other algorithms did not improve the current solution but made it worse for UnitMarch. Various parameters about how and when to use the alternative algorithm have a big influence on the different statistics. It was not easy to focus on an overall performance because of fluctuations.

Preprocessing using unit propagation costs very little time and shows a huge improvement for the assignments that contain unit clauses (see section 2.3).

We will end with some proposals for future work to try to improve the UnitMarch algorithm. To prevent running an alternative without it is necessary, we can think of defining enter conditions. UnitWalk,

for example, switches to WalkSAT if the number of conflicts will become above a certain threshold. Another proposal is to make different strategies for different bits of UnitMarch, which mean you run different alternative for different bits. Also combinations of alternatives could improve the algorithm. At last we mention the optimization of the preprocessing. We discovered preprocessing has a great influence, so it is worth to optimize this.

## References

- [1] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.
- [2] E.A. Hirsch and A. Kojevnikov. UnitWalk: A New SAT Solver that Uses Local Search Guided by Unit Clause Elimination. *Annals of Mathematics and Artificial Intelligence*, 43(1):91–111, 2005.
- [3] D. Le Berre and L. Simon. The essentials of the SAT 2003 competition. *Sixth International Conference on Theory and Applications of Satisfiability Testing*, 2919:452–467.
- [4] Marijn J.H. Heule and Hans van Maaren. From idempotent generalized boolean assignments to multi-bit search. *SAT 2007 Springer LNCS*, pages 134–147, 2007.
- [5] B. Selman and H. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. *Proceedings of the International Joint Conference on Artificial Intelligence*, 1:290–295, 1993.
- [6] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. *Proc. AAAI*, 94:337–343, 1994.
- [7] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.
- [8] Miroslav N. Velev. Formal verification of VLIW microprocessors with speculative execution. In *Computer Aided Verification*, pages 296–311, 2000.