
Exploring the Spectrum between CDCL and Local Search SAT Solvers

Tim van Heugten (1099213), Hamid Mushtaq (1542788),
Pieter Senster (1223577), and Ot ten Thije (1282859)

Delft University of Technology, The Netherlands

February 27, 2010

Abstract

In this paper we explore how far the conversion from a CDCL solver to a local search solver should be taken. To answer this question, we convert the CDCL solver MiniSat into a local search solver similar to UnitWalk, and benchmark the performance at three intermediate stages in the conversion process. We show that depending on the problem class at hand (application, crafted or random), one of the original MiniSat and UnitWalk solvers is always better. That is, no tested compromise of these approaches produces a solver that performs better than both original solvers on general benchmark instances. However, the two local-search-like solvers came very close to UnitWalk on randomized problem instances.

1 Introduction

A recent trend in the development of conflict-driven clause learning (CDCL) SAT solvers has been to adopt strategies used in local search solvers. In particular, current CDCL solvers incorporate phase-saving techniques [9, 12] and restart ever more frequently [3]. These features are combined with the traditional strong points of CDCL solvers: their use of heuristics to select decision variables and the use of lazy data structures for unit clause detection. Combining these approaches has yielded significant improvements in the performance of CDCL solvers [3, 9].

This trend raises the question what the optimum combination between the CDCL method and the local search approach is. To answer this question we adapt the CDCL solver MiniSat [12] to use a local search algorithm similar to UnitWalk [6]. We then benchmark the performance at three intermediate phases in the conversion process, to determine at what stage the combination obtains the best results.

The remainder of this section contains preliminary information on both conflict-driven clause learning SAT solvers and local search SAT solvers. In section 2 we describe the three phases in which adaptations have been made to MiniSat. The set-up of the benchmark procedure is described in 3. Section 4 presents the results of these benchmarks, which are further discussed in section 5. Finally, we state our conclusions in section 6 along with pointers for further research.

1.1 Conflict-Driven Clause Learning SAT solvers

One of the most popular techniques to solve the satisfiability problem is the use of a conflict-driven clause learning solver [1]. State-of-the-art solvers such as zChaff [8], MiniSat [11] and Rsat [10] have shown superior performance on industrial benchmarks (such as model checking and hardware verification).

All CDCL solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [2]. This procedure first performs *unit propagation* to simplify a formula and checks whether it is already satisfied. If it is not, heuristics are used to select a decision variable to branch on. A branch corresponds to assigning the propositional value 0 or 1 to the selected decision variable.

When a falsified clause is detected in one of the branches, a conflict clause is generated. This clause prohibits the set of variable assignments that imply the conflict. Then, the procedure backtracks to the highest decision level where this conflict clause is unit. This is called *backjumping*.

CDCL solvers will always find a satisfying assignment if there is one, and will return UNSATISFIABLE if no such assignment exists. In other words, CDCL solvers are *complete*.

A general overview of the main loop of a CDCL solver, in this case MiniSat, can be found in algorithm 1.

Algorithm 1 MiniSat main loop

```
1: while true do
2:   conflict ← PROPAGATE()
3:   if not conflict then
4:     if all variables assigned then
5:       return SATISFIABLE
6:     else
7:       DECIDE()
8:   else
9:     ANALYZE(conflict)
10:    if top-level conflict found then
11:      return UNSATISFIABLE
12:    else
13:      BACKJUMP(conflict)
```

Phase-saving When a CDCL solver backjumps, a satisfying assignment to a part of the problem that is not related to the current conflict could be lost. This results in the same work being done multiple times, decreasing the performance.

In [9], Pipatsrisawat and Darwiche introduced a “lightweight component caching scheme” that addresses this problem. They save the current assignment in an additional array before backjumping. Later on, when a decision variable needs to be given a truth value, the value from the saved assignment is used. This way the solution to a subproblem that is found earlier is retained. An empirical study of the performance gain related to MiniSat shows a significant improvement [9]. This caching approach is also referred to as “phase-saving” in literature.

The most recent version of MiniSat discussed in the literature [12] implements this phase-saving scheme. However, the most current version available online (MiniSat 2) does not.

Restart strategies Some subproblems require exponentially more computation time than other subproblems from the same problem [1]. This phenomenon is called *heavy-tailed behaviour* [3]. Restart strategies such as randomized restarts make sure that a solver does not spend too much computational effort on such a subproblem if there are other, simpler subproblems which may yield an answer more quickly. A simple strategy used by MiniSat 2 is to restart once the number of conflicts reaches a certain threshold. More recent solvers such as MiniSat 2.1 implement a restart strategy which is based on the sequence proposed by Luby et al. [7], which generally results in much faster restarts than MiniSat 2 does.

Activity heuristics Branching heuristics are a key aspect of conflict-driven SAT solvers. At every branch a decision must be made on what value to assign to which variable. The MiniSat algorithm uses a variant of the Variable State Independent Decaying Sum (VSIDS) heuristic [8]. However, instead of assigning a value to every literal as in the original paper, MiniSat only assigns a value to the variables. In other words, a variable and its negation are treated as equals [11]. Another difference is that the initial value of the variables is not determined using some heuristic, whereas in original VSIDS the literals are weighted by the number of clauses in which they appear [8].

Despite these changes the basic scheme remains the same:

1. All variables are initialised to 0
2. Branching is done on the variable with the highest value
3. When a conflict clause is created, all involved variables have their value increased
4. All values are decayed periodically (in MiniSat: each time a conflict clause is processed)

The underlying idea here is that variables that have frequently occurred in recent conflicts are likely to cause conflicts in the future. By

giving these variables a higher value, MiniSat is likely to select them when branching, increasing the chance of running into a conflict quickly.

Since MiniSat is a conflict-driven solver, it aims to encounter conflicts as early as possible. After all, the earlier a conflict is detected, the sooner a subtree of the solution space can be eliminated. Thus encountering conflicts earlier results in a speed-up when the VSIDS heuristic is used.

1.2 Local Search SAT solvers

A local search SAT solver uses a full assignment, initially picked at random, and then continues to make changes to this assignment until the given formula is satisfied. In this paper, UnitWalk [6] is used as a reference for the local search solvers. UnitWalk uses a full assignment with random values at the start of the algorithm. In each iteration the value of the most important – according to a random order of the variables – free variable of the full assignment is copied to a new assignment. Then all assignments that are made due to unit propagation are copied to the full assignment.

Notice the resemblance between the phase-saving technique employed in CDCL solvers and the UnitWalk algorithm [4]. They both save assignments made due to unit propagation and use these assignments when choosing a value for a decision variable.

One drawback of local search SAT solvers is that they are incomplete, i.e. they cannot conclude that a satisfying assignment for a problem does not exist. When an incomplete solver finds a solution it will return “satisfiable”, but when it does not find a solution it can only announce that the solution was “not found”. In the latter case the problem can still be either satisfiable or unsatisfiable. If a solution exists however, local search solvers usually find it quite quickly, particularly for random satisfiable problems.

CDCL solvers that incorporate phase-saving can be regarded as complete local search solvers: The phase-saving is similar to UnitWalk, but the solver remains complete due to the learning of new clauses and backtracking from implausible branches. In this paper we explore whether there are promising alternatives in the spectrum between CDCL and local search solvers.

2 Adaptations made to MiniSat

We now turn to determining what the optimal combination between the CDCL method and the local search approach is. This is done by considering three algorithms, each of which acts more like a local search solver and less like a CDCL solver than the previous one. From now on we will

refer to these three algorithms as phase one, two and three. Phase one is almost equal to a CDCL solver, while phase three is almost a local search solver, as illustrated in figure 1.

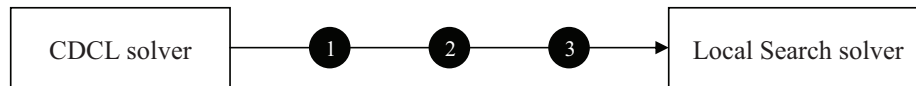


Figure 1: Schematic overview of the three phases in between CDCL and local search.

The phases are:

- (1) Backtrack instead of backjump
- (2) Continue after conflict
- (3) Ignore conflicts

We choose to start with MiniSat¹ because it is a well known solver that is designed to be easily extensible [11]. This provides a good starting point for the adaptations we need to make.

In summary, the adaptations are based on the following ideas. In phase one the intuition is tested that backjumping in a phase-saving solver is highly similar to backtracking. Phase two further exploits the phase-saving technique, by having MiniSat handle full assignments rather than partial ones. This requires the solver to continue after the first conflict has been found. Finally, phase three almost entirely ignores conflicts, using them only to update branching heuristics in an algorithm that otherwise follows the local search approach. The remainder of this section discusses these adaptations in more detail.

2.1 Phase one: Backtrack instead of backjump

When the original MiniSat approach detects a conflict, the solver jumps back to the highest decision level where the learned conflict cause is unit. The algorithm then continues downward again on a different path, using the updated heuristics to pick the values of decision variables.

When phase-saving is used however, most assignments will be identical even after a conflict has been detected. This means that a lot of double work is done by first undoing the assignment during the backjump, and especially during their re-assignment when the solver works its way down the decision tree again. This extra work can be prevented by simply backtracking a single level in the tree, rather than jumping back a lot of steps.

To replace backjumping with backtracking in the MiniSat code, a minor change was made to the `Solver::analyze()` method. This method

¹All adaptations made are based on MiniSat version 2, which does not implement phase-saving (section 1.1). We added phase-saving to MiniSat to be able to make a good comparison between the solvers presented in this section and other recent solvers.

```

1 // Find correct backtrack level:
2 if (out_learnt.size() == 1)
3   out_btlevel = 0;
4 else{
5   int max_i = 1;
6   for (int i = 2; i < out_learnt.size(); i++)
7     if (level[var(out_learnt[i])] >
8         level[var(out_learnt[max_i])])
9       max_i = i;
10  Lit p          = out_learnt[max_i];
11  out_learnt[max_i] = out_learnt[1];
12  out_learnt[1]    = p;
13  out_btlevel      = level[var(p)];
14 }

```

Figure 2: Solver::analyze() fragment administrating the backtrack level.

```

1 Lit p          = out_learnt[max_i];
2 out_learnt[max_i] = out_learnt[1];
3 out_learnt[1]    = p;
4 out_btlevel      = decisionLevel() - 1;

```

Figure 3: Code used to replace the last four lines of figure 2 in phase one.

determines the decision level which the solver should jump back to when a conflict has been found, storing it in the variable `out_btlevel`. In normal MiniSat, this variable is set to the level at which the conflict clause became unit. Our change consists of changing the value of `out_btlevel` to the decision level one below the current one. Before the change `out_btlevel` was assigned as shown in figure 2. In the adapted version, it is set to the current decision level minus one. The exact code used, to replace lines 10 to 13, is shown in figure 3.

2.2 Phase two: Continue after conflict

The implementation of MiniSat enforces the invariant that the solver is never in a conflicting state [12]. This invariant is maintained by resolving the conflict as soon as it is detected. UnitWalk has no such invariant but continues unit propagation even when conflicts occur; in fact it is very likely to reach several full assignments that contain conflicts. In our approach from CDCL solver towards a local search solver we will have to let go of this restriction too. In this phase no action is taken to revert a conflict when it is encountered in the current assignment.

Still similar to MiniSat, this phase uses heuristics for picking a branch variable. The variant of the VSIDS heuristic used in this phase is slightly

different from the original MiniSat implementation. Our tests showed that reversing the heuristics provides a great improvement of the performance of the algorithm. Branch variables are not selected when they have the highest activity value, but instead if they have the lowest. So, with the inverted heuristics, the algorithm is branching on variables that were involved the least in recent conflicts. In this phase we want to find conflicts as late as possible, because then most of the unit propagation is done on an assignment that does not contain a conflict, and thus gives assignments closer to a correct solution. This inversion of heuristics was achieved by flipping the comparison operator in the `VarOrderLt` definition.

Another aspect that remains identical to the CDCL approach is the learning of conflict clauses. The reason for this is that conflict clauses do contain valuable information about restrictions to the correct assignment, when such an assignment exists. Although they are not used to trigger backtracking when they lead to a conflict in a next iteration, they can be useful when propagating unit information and lead the solver away from possible conflicts. Therefore, after a conflict is found this algorithm still analyses the conflict and constructs a conflict clause from it. The new clause is then added to the conflict database, if it contains more than one literal.

Conflict clauses that consist of only one literal are ignored, because the MiniSat implementation does not handle single literal clauses. On the other hand, the (unit) clause can also not be propagated right away. The reason for this is that no backtracking is performed and the (unit) literal is therefore still assigned. For similar reasons it is not possible to re-branch on the involved variables for the larger conflict clauses, since all those variables stay assigned.

In algorithm 2 we show the pseudocode for this phase. Despite the fact that conflict clauses are learned in this phase, it is unable to conclude that a problem is unsatisfiable: the solver has become incomplete. The previous phases were able to conclude unsatisfiability on the basis of a conflict on decision level zero. Because this phase never backtracks however, it will never encounter a conflict on decision level zero. Only unit propagation, as a result of branching, can lead to a conflict, but that implies that the decision level is no longer zero. The difference arises also from the fact that unit conflict clauses that were previously enqueued are ignored in this phase.

To allow the algorithm to terminate while searching for a solution on unsatisfiable problems, the number of solver iterations is bounded by `MAX_PERIODS`. When this value is exceeded, the algorithm breaks off its search and terminates with an internal timeout. In such cases the solver returns the result "unknown". The value of `MAX_PERIODS` is set to 500.000,

which was expected to be sufficiently high to find the solution to any solvable problem.

Algorithm 2 MiniMarch main loop, phase two

```

1: for  $i = 1$  to MAX_PERIODS do
2:   conflict_found  $\leftarrow$  false
3:   while not all variables assigned do
4:     conflict  $\leftarrow$  PROPAGATE()
5:     if conflict then
6:       conflict_found  $\leftarrow$  true
7:       conflict_clause  $\leftarrow$  ANALYZE(conflict)
8:       ADD_CLAUSE(conflict_clause)
9:       DECAY_VAR_ACTIVITY()
10:    DECIDE()
11:   if conflict_found then
12:     BACKTRACK(0)
13:   else
14:     return SATISFIABLE
15: return UNKNOWN

```

The implementation of phase two consists of two changes to the MiniSat code. First, in order to continue after a conflict has been found, we remove the backjump call and thus simply continue assigning. We do however record that a conflict has occurred, for otherwise MiniSat would conclude it has found a satisfying assignment from the fact that it reached a full assignment. Figure 4 shows the relevant fragment from the original `Solver::search()` method in MiniSat. The fragment that replaces this part in the implementation of phase two is shown in figure 5. Because we no longer backtrack, we also removed the part of the code of `Solver::analyze()` that determines to which level backtracking has to be performed. That is, we removed all code shown in figure 2.

The second change is that once a full assignment is reached, we check our conflict flag to see if there was a conflict. If a conflict did occur, we restart the solver by backtracking to decision level 0. This effectively clears the entire current assignment and starts a new period, while of course retaining the information by phase-saving. The relevant code was inserted in the `Solver::search()` method, and is reproduced here in figure 7.

2.3 Phase three: Ignore conflicts

In this phase, all conflict-driven features of the MiniSat algorithm have been removed. The algorithm has become like an implementation of the

```
1 learnt_clause.clear();
2 analyze(confl, learnt_clause, backtrack_level);
3 cancelUntil(backtrack_level);
4 assert(value(learnt_clause[0]) == l_Undef);
5
6 if (learnt_clause.size() == 1){
7   uncheckedEnqueue(learnt_clause[0]);
8 } else {
9   Clause* c = Clause_new(learnt_clause, true);
10  learnts.push(c);
11  attachClause(*c);
12  claBumpActivity(*c);
13  uncheckedEnqueue(learnt_clause[0], c);
14 }
15
16 varDecayActivity();
17 claDecayActivity();
```

Figure 4: Original Solver::search() fragment handling conflicts in MiniSat 2.

```
1 learnt_clause.clear();
2 analyze(confl, learnt_clause, backtrack_level);
3
4 conflictFound = true;
5
6 if (learnt_clause.size() != 1){
7   Clause* c = Clause_new(learnt_clause, true);
8   learnts.push(c);
9   attachClause(*c);
10  claBumpActivity(*c);
11 }
12
13 varDecayActivity();
14 claDecayActivity();
```

Figure 5: The fragment that replaces the code shown in figure 4 in phase two.

```
1 learnt_clause.clear();
2 analyze(confl, learnt_clause, backtrack_level);
3
4 conflictFound = true;
5
6 varDecayActivity();
```

Figure 6: The fragment that replaces the code shown in figure 4 in phase three.

```
1 escape--; // number of remaining iterations
2 if(status != l_False && conflictFound && escape > 0)
3 {
4     polarity_mode = polarity_user;
5     for(int i = 0; i < nVars(); i++) {
6         polarity[i] = (toLbool(assigns[i]) != l_True);
7     }
8
9     cancelUntil(0);
10    status = l_Undef;
11    conflictFound = false;
12 }
```

Figure 7: Code to restart the phase two solver when a conflicting full assignment is found. This fragment was inserted in the `Solver::solve()` method, at the end of the “Search” loop.

local search algorithm used in `UnitWalk`. No backtracking is performed and no new clauses are learned. Unlike `UnitWalk` however, variable activity is recorded, so the `analyze` method must still be executed. The changes to the conflict handling part of the `Solver::search()` method are illustrated in figure 6. Similar to phase two, the determination of the backtrack level in the `Solver::analyze()` method is omitted from the implementation, because no backtracking is performed and the resulting value is of no meaning.

The implementations of phase two and three are very similar. The only essential difference is that phase two learns conflict clauses, which is no longer done in phase three. Apart from this, the changes from phase two carry over to phase three. Therefore the heuristics are reversed based on the same rationale given earlier, in order to improve performance.

Because this phase is a local search algorithm, it is not a complete solver. As in phase two, the number of iterations is capped to enforce an end to the search in unsatisfiable problems. The value of `MAX_PERIODS` remains set to 500.000. The pseudocode for this phase can be found in algorithm 3.

3 Method

To determine the merits of our adaptations, we ran them on a set of benchmarks and compared their performance to that of current CDCL and local search solvers. First we present the method used to select the benchmarks in section 3.1. We then discuss the solvers used as reference in section 3.2. Finally, the technical details about the execution of the tests is explained in 3.3.

Algorithm 3 MiniMarch main loop, phase three

```
1: for  $i = 1$  to MAX_PERIODS do
2:   conflict_found  $\leftarrow$  false
3:   while not all variables assigned do
4:     conflict  $\leftarrow$  PROPAGATE()
5:     if conflict then
6:       conflict_found  $\leftarrow$  true
7:       ANALYZE(conflict)           // bumps variable activity
8:       DECAY_VAR_ACTIVITY()
9:     DECIDE()
10:  if conflict_found then
11:    BACKTRACK(0)
12:  else
13:    return SATISFIABLE
14: return UNKNOWN
```

3.1 Benchmark selection

To verify the correctness of our solvers and determine their efficiency, their performance was recorded for various benchmark problems. These problems are taken from the 2009 SAT competition database².

The performance of MiniSat 2.1 in the 2009 SAT competition was used as guidance while assembling the test set. For each of the categories (application, crafted and random), only instances solved by MiniSat 2.1 were considered for inclusion. These were ordered by the time MiniSat needed to solve them, after which we selected twenty satisfiable problems from each category. The selected problems were equally distributed over the ranking. For example, we took the fastest, then the third fastest, the fifth fastest and so on. This makes sure that there are both relatively easy and relatively hard problems in our set.

The same procedure was used to select the unsatisfiable benchmarks. However, since some algorithms presented in this paper are incomplete, we selected a smaller set of such problems: ten from each category. Because MiniSat 2.1 did not solve ten unsatisfiable random instances, we only included the four it did solve. A complete overview of all used benchmarks can be found in appendix A.

Note that by taking MiniSat 2.1 as reference, the size of the problem instances is determined by how well MiniSat handles problem instances. Therefore the size of our random benchmarks in terms of variables and clauses is rather small compared to the size of the application and crafted benchmarks.

²<http://www.satcompetition.org>

3.2 Reference solvers

In order to compare the performance of our three adaptations to current solvers, we also ran the benchmarks using representatives of the two “extremes” of the CDCL – local search spectrum: MiniSat and UnitWalk.

Unfortunately the source code of the most recent MiniSat 2.1 solver has not been made available to the public, so we had to use MiniSat 2 instead. However, in order to reflect at least some of the improvements made in MiniSat 2.1, we modified MiniSat 2 to incorporate the phase-saving technique used by our adaptations. Both the original MiniSat 2 solver and our modified version were run on all benchmarks.

3.3 Technical setup

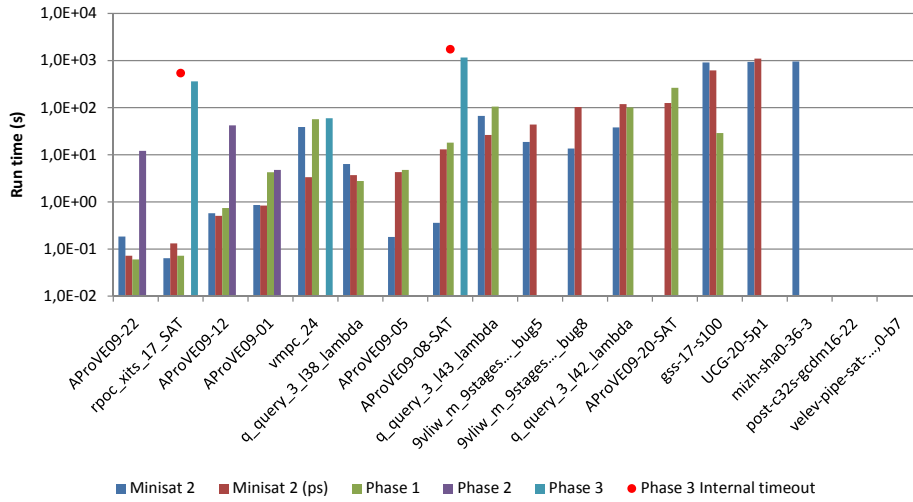
Because we ended up with eight different SAT solvers, the tests had to be run in parallel. Therefore, several PC’s with a similar configuration have been used to run the tests. All benchmarks have been executed on a PC running SUSE Linux 11, equipped with 2GB of memory and an Intel Core2 Duo E6850 processor clocked at 3.00GHz.

Still, allowing all solvers to take the time they needed to solve a problem would take too much time. To circumvent this problem, every run of a solver was limited by a maximum execution time of 20 minutes. Besides this (external) timeout, which is used to end solvers on big problems, the incomplete solvers use an internal timeout to end on unsatisfiable problems. For phase two and three this is the MAX_PERIODS limit of iterations, as mentioned before, set to 500.000. When a solver reaches an internal timeout, this has been explicitly noted in the results for satisfiable benchmarks. However, for unsatisfiable this will not be noted, since it is the expected behaviour of these solvers.

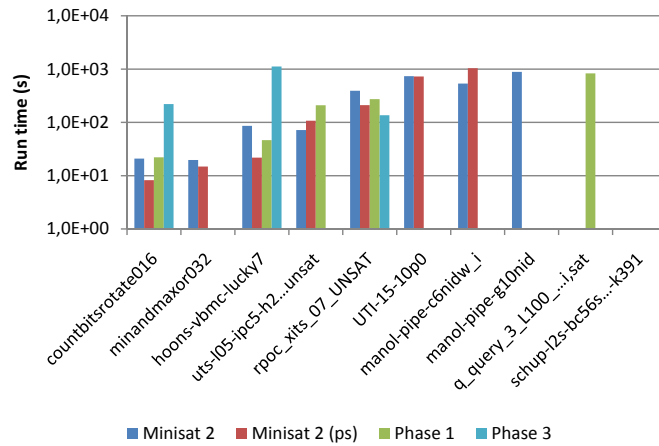
4 Results

Figures 8–10 show the results of running the various solvers on the application, crafted and random benchmarks respectively. The benchmarks in these charts are sorted in increasing order of time required by MiniSat 2 with phase-saving.

As can be seen in figure 8(a), phase-saving (brown bars) usually results in faster run times. Phase one (green) performs comparable to the original MiniSat 2 solver (blue) and MiniSat 2 phase-saving solver (red), on three benchmarks it is even the best performing solver. It does however also solve three problems less within the 20 minute time limit. The results for phase two on this benchmark set (purple) show that it performs far worse than the original MiniSat solvers. It manages to solve only three benchmarks and needs orders of magnitude more time to do so. Phase



(a) Satisfiable benchmarks



(b) Unsatisfiable benchmarks

Figure 8: Results on Application benchmarks. Note that the vertical (time) axis is logarithmic. “MiniSat 2 (ps)” is MiniSat 2 with our implementation of phase-saving. If a solver timed out on a benchmark, that bar is not shown in the figures.

three (light blue) performs even worse: it solves only one benchmark. For two benchmarks it ran into an internal timeout (marked with a red dot), and for the other benchmarks it does not complete its execution within the time limit.

The results for unsatisfiable benchmarks (figure 8(b)) are similar to those on the satisfiable benchmarks. Phase one is able to solve less of the problems than the original solvers, and phase three is able to complete only a few tests. All three results of phase three are internal timeouts on the benchmarks, which must be interpreted as “unknown”. Phase two did not complete any of these benchmarks and was therefore omitted from the chart.

The incompleteness of phase three seen in the application benchmarks becomes even more apparent in the results on the satisfiable crafted benchmarks, shown in figure 9(a). Again, benchmarks where phase three encountered an internal timeout are marked with a red dot.

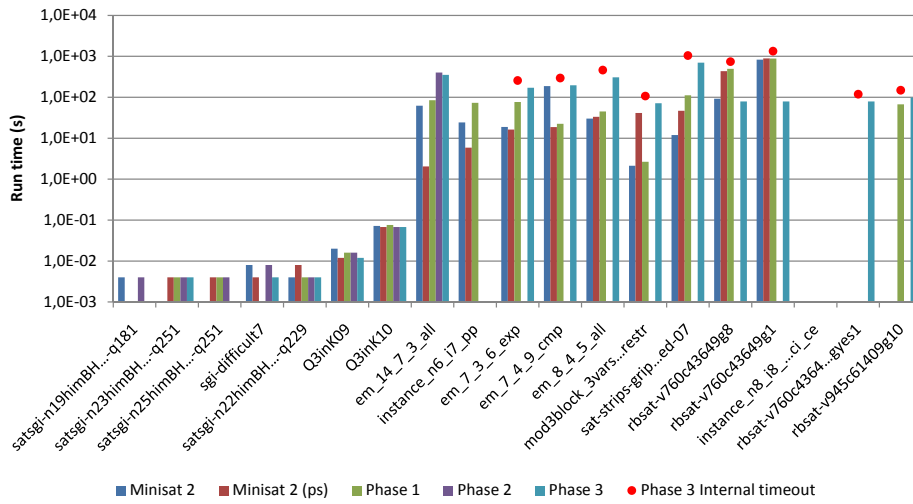
However, the adaptations are more competitive on this set than on the application benchmarks. Especially on the smaller sets the performance of all adaptations is on par with MiniSat 2, both original and with added phase-saving. Considering the entire set the performance of phase one is comparable to the results of MiniSat 2 using phase-saving, sometimes performing better, sometimes worse. Phases two and three perform considerably worse on the larger sets, timing out either internally or externally.

Figure 9(b) shows that phase one performs quite well on the unsatisfiable crafted benchmarks. Phase one is the fastest solver on most of these benchmarks, and is outperformed by the reference solvers on two problems only. Also, the variance in performance appears to be slightly less than on the satisfiable benchmarks. As with the application benchmarks, phase two was unable to complete these unsatisfiable instances within the time limit.

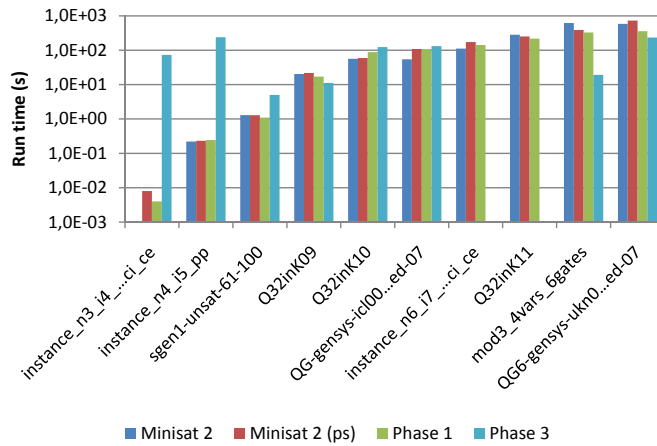
Finally, the results for the random benchmarks, shown in figure 10, are the most promising. What we see here is that phases two and three consistently outperform MiniSat 2, both with and without phase-saving. For phase three, this happens on both satisfiable and unsatisfiable benchmarks.

On average the run times for the satisfiable sets are also quite close to those obtained using UnitWalk (shown in orange), but the scores on individual benchmarks show larger variance. While the performance of phases two and three approaches UnitWalk, they do not consistently beat it: phase two scores better on seven instances, phase three on six. Comparing phase two and three to each other shows that two outperforms three on twelve of the twenty sets.

Compared to the original MiniSat 2 solver, phase one performs better on average, although the difference is less spectacular than for phase three. Another interesting aspect is that MiniSat 2 without phase-saving

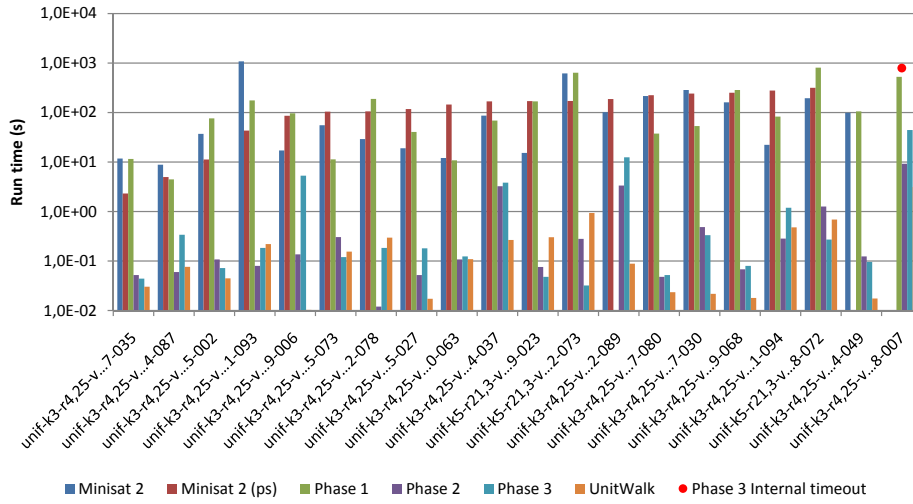


(a) Satisfiable benchmarks

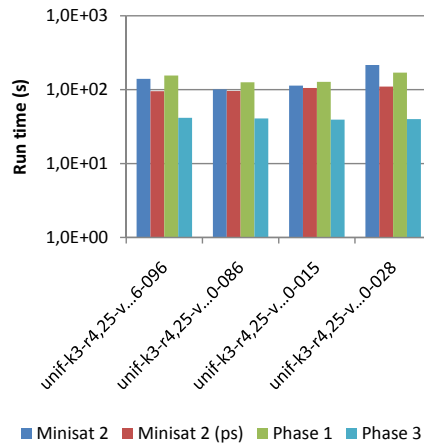


(b) Unsatisfiable benchmarks

Figure 9: Results on Crafted benchmarks. Note that the vertical (time) axis is logarithmic. “MiniSat 2 (ps)” is MiniSat 2 with our implementation of phase-saving. If a solver timed out on a benchmark, that bar is not shown in the figures.



(a) Satisfiable benchmarks



(b) Unsatisfiable benchmarks

Figure 10: Results on Random benchmarks. “MiniSat 2 (ps)” is MiniSat 2 with our implementation of phase-saving. Note that the vertical (time) axes are logarithmic. If a solver timed out on a benchmark, that bar is not shown in the figures.

regularly outperforms MiniSat 2 with phase-saving.

5 Discussion

The difference in performance between the original MiniSat solver and phase one is relatively small. This is in line with our expectations, since backjumping and backtracking should exhibit similar behaviour in the presence of phase-saving.

The performance of phase two significantly differs from the original MiniSat solver. On application and crafted benchmarks phase two is slower than both MiniSat and phase one, and it gives a timeout on more than half of the benchmarks. However, on random benchmarks the performance is about two orders of magnitude better.

The version that is most like UnitWalk, phase three, shows similar performance to UnitWalk on the three problem classes. While the performance is worse than MiniSat on application and crafted benchmarks, phase three performs significantly better on the random benchmarks than both MiniSat solvers, and is on par with UnitWalk.

In order to test the decision to invert the heuristics for phase two and three, we executed all benchmarks with two versions of the two phases: one with original and one with reversed heuristics. For the application and crafted benchmarks the results whether or not the heuristics were inverted were about the same: either way the solvers timed out on most benchmarks. On the random benchmark set however, a clear victor emerged. While the solvers with normal heuristics were still unable to solve even a single benchmark, the inverted versions came to a conclusion quite efficiently. We therefore conclude that using inverted heuristics was the correct decision.

Based on the results for the random benchmarks alone, it is difficult to judge whether adding the conflict clauses (as in phase two) is better than ignoring them altogether (as in phase three). The results are influenced by the decision which conflicts to add, and new insights about these decisions might lead to an advantage for phase two. Since the current results are highly comparable, phase three would be the preferred choice in case memory is scarce.

Finally, we note that it is relatively easy to change a CDCL solver into a solver that uses local search. It is only the main algorithm that changes, while most data structures and administrative procedures remain unchanged. This gives rise to the idea for a hybrid solver that intelligently switches between two algorithms. Since both are implemented in the same solver, the two algorithms can share their progress.

6 Conclusions and future work

In this paper we investigated where the optimal combination between the CDCL method and the local search approach lies. We have shown that depending on the problem class at hand (application, crafted or random), the original solvers, MiniSat and UnitWalk, perform best. That is, a compromise of both approaches does not give us a solver that performs generally better than either of the original solvers.

In our research we have been unable to cover all the aspects that showed up while exploring the intermediate stages of satisfiability solvers. Here we suggest several directions that could be explored in order to improve the performance of the solvers that are mentioned in this report.

- We have seen that MiniSat can be changed into a local search solver while maintaining its data structures and helper methods. A version of MiniSat with a switching strategy between CDCL and local search might therefore have potential.
- We showed that inverting the heuristics, compared to MiniSat, provided a big performance improvement for phases two and three. This was done by only reversing the ordering of the priority of the decision variables, while maintaining the same activity valuation.

Although the activity measurement worked well for CDCL, a different measure might work even better for the conflict avoiding solvers that phase two and three have become.

- In section 2.2 it was shown that phase two has become an incomplete solver because no level zero conflicts could be encountered. To make phase two complete, some propagation should be done on level zero. One obvious candidate approach is to enlist any unit conflict clauses that the solver may encounter, and propagate these literals at level zero, every following iteration. Also the larger conflict clauses contain at least one literal that is assigned the wrong value. But since it is not immediately clear which of these literals is wrong, some more care is needed before it can be decided that they can help to make phase two complete.

References

- [1] A. Biere, M. Heule, H. van Maaren, and T. Walsh. Handbook of Satisfiability, Chapter 4, 2009.
- [2] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):397, 1962.
- [3] C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, 24(1):67–100, 2000.
- [4] S. Haim and M. Heule. Towards Ultra Rapid Restarts. *Unpublished*.
- [5] M. Heule and H. van Maaren. Parallel SAT Solving using Bit-level Operations. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:99–116, 2008.
- [6] E. 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.
- [7] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.
- [8] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th conference on Design automation*, pages 530–535. ACM New York, NY, USA, 2001.
- [9] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. *Lecture Notes in Computer Science*, 4501:294, 2007.
- [10] K. Pipatsrisawat and A. Darwiche. Rsat 2.0: Sat solver description. *SAT competition*, 7, 2007.
- [11] N. Sörensson and N. Eén. An extensible SAT-solver. *Lecture notes in computer science*, 2919:502–518, 2004.
- [12] N. Sörensson and N. Eén. MiniSat 2.1 and MiniSat++ 1.0-SAT Race 2008 Editions. *SAT 2009 competitive events booklet: preliminary version*, page 31, 2009.
- [13] L. Zheng and P. J. Stuckey. Improving sat using 2sat. *Aust. Comput. Sci. Commun.*, 24(1):331–340, 2002.

A Benchmarks

Application	Crafted	Random
9vliw_m_9stages_iq3_C1_bug5 AProVE09-12 gss-17-s100 q_query_3_l43_lambda 9vliw_m_9stages_iq3_C1_bug8	Q3inK09 rbsat-v760c43649g1 Q3inK10 rbsat-v760c43649g8 em_14_7_3_all	unif-k3-r4.25-v360-c1530-S144043535-002 unif-k3-r4.25-v380-c1615-S326477539-068 unif-k3-r4.25-v360-c1530-S1684547485-073 unif-k3-r4.25-v380-c1615-S486002104-037 unif-k3-r4.25-v360-c1530-S1826927554-087
AProVE09-20-SAT mizh-sha0-36-3 rbcl_xits_14_SAT AProVE09-01 AProVE09-22	rbsat-v760c43649gyes1 em_7_3_6_exp rbsat-v945c61409g10 em_7_4_9_cmp sat-strips-gripper-12t23.sat05-1144.reshuffled-07	unif-k3-r4.25-v400-c1700-S1326039470-063 unif-k3-r4.25-v360-c1530-S2032263657-035 unif-k3-r4.25-v400-c1700-S419666888-007 unif-k3-r4.25-v360-c1530-S722433227-030 unif-k3-r4.25-v420-c1785-S180439591-093
post-c32s-gcdm16-22 rpoc_xits_17_SAT AProVE09-05 UCG-20-5p1 q_query_3_l38_lambda	em_8_4_5_all satsgi-n19himBHm22-p0-q181 instance_n6_i7_pp satsgi-n22himBHm25-p0-q229 instance_n8_i8_pp_ci_ce	unif-k3-r4.25-v380-c1615-S1036996799-006 unif-k3-r4.25-v420-c1785-S1940784272-089 unif-k3-r4.25-v380-c1615-S1041106011-094 unif-k3-r4.25-v440-c1870-S208461254-049 unif-k3-r4.25-v380-c1615-S1516934767-080
velev-pipe-sat-1.0-b7 AProVE09-08-SAT	satsgi-n23himBHm27-p0-q251 mod2-rand3bip-sat-220-1.sat05-2173.reshuffled-07	unif-k5-r21.3-v90-c1917-S1040456712-073 unif-k3-r4.25-v380-c1615-S1674394075-027
UTI-15-10p1 q_query_3_l42_lambda vmpc_24	satsgi-n25himBHm27-p0-q251 mod3block_3vars_9gates_restr sgi-difficult7	unif-k5-r21.3-v90-c1917-S744068878-072 unif-k3-r4.25-v380-c1615-S1841979702-078 unif-k5-r21.3-v90-c1917-S852794659-023

Table 1: Satisfiable benchmarks in test set.

Application	Crafted	Random
UTI-15-10p0.cnf.gz	Q32inK09	unif-k3-r4.25-v360-c1530-S1028159446-096
manol-pipe-c6nidw_i.cnf.gz	instance_n3_i4_pp_ci_ce	unif-k3-r4.25-v360-c1530-S23373420-028
q_query_3_L100_coli.sat.cnf.gz	Q32inK10	unif-k3-r4.25-v360-c1530-S1369720750-015
uts-105-ipc5-h26-unsat.cnf.gz	instance_n4_i5_pp	unif-k3-r4.25-v360-c1530-S253750560-086
countbitsrotate016.cnf.gz	Q32inK11	
manol-pipe-g10nid.cnf.gz	instance_n6_i7_pp_ci_ce	
rpoc_xits_07_UNSAT.cnf.gz	QG-gensys-icl003.sat05-2715.reshuffled-07	
hoons-vbmc-lucky7.cnf.gz	mod3_4vars_6gates	
minandmaxor032.cnf.gz	QG6-gensys-ukn003.sat05-2726.reshuffled-07	
schup-l2s-bc56s-1-k391.cnf.gz	sgen1-unsat-61-100	

Table 2: Unsatisfiable benchmarks in test set.

AProVE09-01-SAT	AProVE09-13-SAT	cmu-bmc-longmult15-UNSAT	minor032-UNSAT
AProVE09-03-SAT	AProVE09-17-SAT	een-tip-sat-texas-tp-5e-SAT	q_query_3_l37_lambda-SAT
AProVE09-05-SAT	AProVE09-19-SAT	een-tip-uns-nusmv-t5.B-UNSAT	q_query_3_l38_lambda-SAT
AProVE09-07-SAT	AProVE09-21-SAT	hsat_vc11803-UNSAT	q_query_3_l39_lambda-SAT
AProVE09-10-SAT	AProVE09-22-SAT	hsat_vc11813-UNSAT	rpoc_xits_17_SAT
AProVE09-11-SAT	AProVE09-24-SAT	manol-pipe-c9-UNSAT	schup-l2s-abp4-1-k31-UNSAT
AProVE09-12-SAT	AProVE09-25-SAT	manol-pipe-g6bi-UNSAT	xinetd_vc56703-UNSAT

Table 3: Benchmarks in the Debug set.

B An Extensible SAT-solver

The writers of the article noticed that there is a lot of information on the techniques used in SAT solvers that implement a conflict-driven learning style, but that this information is insufficient to be used to create an implementation of such a solver. In the article they attempt to describe a solver to such an extent that it is possible for the reader to implement a solver themselves or to extend the minimal solver MiniSat. The authors do this based on the experience they gained from constructing the solvers Satzoo and Satnik and use MiniSat as an example.

This summary will not contain all the details required for implementing a SAT solver. Instead the concepts that are covered in the article are mentioned here, so an interested reader will know what there is to find in the original article.

Search The search is the process of exploring the search tree by picking a variable and assigning a value to it. Then the consequences of that assignment are propagated, possibly resulting in more variables to be assigned, and the propagation of these variables. Every variable that is assigned as a consequence of the first assignment is said to have happened on the same decision level.

All the assignments are recorded in a stack. The stack is divided in decision levels and is known as the trail. This procedure continues until all variables are assigned and a solution is found, or until a conflict is found. Then the analyze method is called to construct a learned clause that contains the literals that lead to the conflict. The variables are returned to an unassigned state, one decision level at a time, until one literal from the conflict clause becomes unit. That literal is then propagated, and the process continues. When the search encounters a conflict at level 0 it has determined that the problem is unsatisfiable.

Propagation For the concept of unit propagation MiniSat employs the data structure of watched literals. The advantages are that when a variable is assigned, not all clauses that contain the corresponding literals need to be checked, and also, when backtracking, the watched literals need not to be updated, so backtracking is cheap.

When propagating a variable, all clauses for which it is a watched literal are visited. If the clause was already satisfied, the change to the propagated variable is ignored. If, however, the literal is falsified, a new unassigned literal is looked for to become the new watched literal. When no such literal is found the other watched literal becomes unit and is enqueued to be propagated.

Learning When a conflict is encountered the algorithm will learn from it before it backtracks to a non conflicting state. Each of the assigned variables involved is either the result of a decision by the algorithm or it was propagated as the result of some other decisions. In the latter case the algorithm searches back to get a list of the decision variables of which the conflict is a direct consequence. The values of these variables are taken together in a new learned clause.

Activity Heuristics To choose which variable to use for branching, MiniSat uses a variable based heuristic. When a variable is involved in a conflict its activity value is increased. At the same time the activity value of all variables is decreased, to reflect the higher importance of variables involved in more recent conflicts.

A similar approach is used for clauses. Because learning too many clauses can increase the problem size too much, such that it deteriorates the performance of the algorithm. Before such a point is reached the algorithm can remove some of the learned clauses, and it will do so based on the activity heuristics of the clauses.

Solve The general procedure of the MiniSat algorithm is the solve method. It is called after the problem description has been transported to the solver. From this method, the search method is called to start looking for a solution. The solve method determines the maximum number of clauses to be learned, before some have to be discarded, and how many conflicts may be encountered. When too many conflicts are encountered, the search is restarted to prevent the solver from pursuing a too deep search path. If the search found a solution, or a decision level 0 conflict, before the limit of conflicts was encountered, the solve method will pass this value to its caller.

C Conflict-Driven Clause Learning SAT Solvers

One of the most popular techniques to solve the satisfiability problem is the use of a conflict-driven clause learning (CDCL) solver. CDCL solvers are based on the Davis-Putnam-Logemann-Loveland procedure [2]. This procedure first performs *unit propagation* to simplify a formula and checks whether it is already satisfied. Otherwise, heuristics are used to select a decision variable to branch on. A branch corresponds to assigning the propositional value 0 or 1 to the selected decision variable. When a falsified clause is detected in one of the branches, the procedure backtracks to the highest decision level where the conflict clause is unit. This is called *backjumping*.

Modern CDCL solvers have several important features that are not present in the standard DPLL procedure, including lazy data structures, search restarts and branching heuristics.

Ordinary backtracking SAT solvers represent a clause as a list of literals. Furthermore, each variable is associated with a list of clauses in which it is contained. Since the number of clauses increases in a CDCL solver, this leads to an efficiency problem.

Lazy data structures, such as watched literals, solve this problem using the following insight: a clause can only propagate unit information when all but one of its literals have become false. To utilize this knowledge, the watched literals structure maintains pointers to two distinct literals from each clause, called *watch pointers*. These pointers obey the invariant that they will never point at a falsified literal, if they can point at true or unassigned literals instead. As the solver proceeds and more variables become assigned, these watch pointers are updated to point at either unknown or true literals, maintaining this invariant.

When one of the watched literals is true, the clause is satisfied and further updating is no longer necessary. But when one of the watched literals is false, the invariant implies that all unwatched literals must also be false. This means that the clause can be declared unit on the literal watched by the other pointer, without having to inspect any other literal.

Restart strategies such as rapid randomized restarts make sure that a solver does not spend too much computational effort on such a subproblem if there are other, simpler subproblems which may yield an answer more quickly (the so called heavy-tailed behaviour).

Finally, the introduction of lazy data structures has given rise to new branching heuristics such as VSIDS. These new heuristics are needed since no accurate information about the number of satisfied literals in a clause is available in such a data structure. VSIDS instead chooses the literal that appears most often in all clauses. This very simple heuristic was proposed mostly because it is very fast, even though it is not very accurate.

D Parallel SAT solving

A natural way to encode the assignments used in SAT solvers is to use a one when a variable is true and a zero when it is false. This approach is used by most SAT-solvers, but these will only evaluate a single assignment during each run. The paper “Parallel SAT Solving using Bit-level Operations” [5] presents a way to exploit the fact that modern processors operate on numbers of 32 or 64 bits in size. This is done by evaluating assignments in parallel, using bit-level operations such as AND and OR implemented in the hardware of the processors themselves. The rest of this appendix gives a summary of the approach discussed in the paper just mentioned.

The basic idea is to encode multiple assignments for a variable in a single k -bit integer, called a *bit vector*. In this vector, each bit represents a truth value in a separate assignment. For example, consider the following 2-bit integers:

$$x_1 = 10, x_2 = 01$$

Here, the leftmost bits represent the assignment where x_1 is true and x_2 is false. Using the rightmost bits on the other hand, the assignment is x_1 false and x_2 true.

To demonstrate that this approach can be used to solve SAT problems more efficiently, the existing UnitWalk solver was adjusted to incorporate bit vectors rather than simple assignments. The UnitWalk algorithms essentially relies on two operations: Detecting when clauses are unit under a given assignment, and propagating such unit information once a unit clause has been detected. We will now discuss how these operations can be implemented to work on bit vectors.

Representation In order to efficiently detect unit clauses, the representation of assignments needs to be slightly adjusted. Rather than using just one bit to distinguish true and false assignments, we reserve a second bit which allows us to also indicate when variables are unassigned or in conflict.

By encoding true as 01 and false as 10, the NOT operator retains its semantics. This also has the benefit that if a variable occurs as negated literal, we merely need to interpret the variables’ value from right to left in order to get its negated value. If for example x_1 true, then it is encoded as 10, so when $\neg x_1$ occurs as a literal we read right to left, obtaining the value 01, which is indeed the encoding for false. The values “unassigned” and “bit conflict” are encoded as 00 and 11 respectively. For ease of implementation, the two bits for each assignment are stored in separate arrays, F and F_{\neg} .

Unit clause detection Now remember that a clause is unit when all but one of its literals are false, and when that final literal is unassigned. This can be detected using the arrays F and F_{\neg} just described. A literal x_i is unassigned exactly when $\text{NOT}(F[x_i] \text{ OR } F_{\neg}[x_i])$, it is false when $F_{\neg}[x_i]$ is true and true when $F[x_i]$ is true.

For example, to determine whether the clause $x_1 \wedge \neg x_2 \wedge x_3$ is unit on literal x_3 , we evaluate:

$$F_{\neg}[x_1] \text{ AND } F[x_2] \text{ AND NOT } (F[x_3] \text{ OR } F_{\neg}[x_3])$$

Since the logical operators all work on bit vectors, this technique can be used to detect unit clauses in multiple assignments simultaneously.

Unit propagation Once unit clauses have been detected, they need to be propagated. This is done using an adaptation of the UnitWalk algorithm. As in UnitWalk, a random permutation of variables is selected. The value of the first variable in the permutation is then copied from the previous assignment. If this leads to unit clauses on some bit positions, the unit information is propagated on those bit positions only. This may in turn lead to new unit clauses, etcetera. When the propagation stops, the next variable with unassigned bits is selected from the permutation, and its unassigned bit positions are copied from values in the previous iteration. If at some point a fully satisfying assignment is generated at any bit position, the algorithm stops and returns the values at that bit position as a found satisfying assignment.

Optimizations The performance of the original bit-parallel algorithm can be further improved by applying two optimizations: Removing duplicate assignments and detecting autarkies.

Duplicate assignments occur when different bit positions in the bit-vectors converge to represent the same assignments. Once this has happened, the algorithm is effectively run twice on the same input, which of course wastes resources. Fortunately, such duplications can be detected using a sequence of clever matrix multiplications. Once detected, a duplicate assignment is replaced by a randomly chosen new assignment.

Autarkies are subsets of clauses in the formula, with the following property: The set of all variables used in these clauses is disjoint from the variables used elsewhere in the formula. Therefore, once a satisfying assignment for an autarky has been found, it can be removed from the formula: if an assignment for the remaining variables is found that satisfies the remaining clauses, this assignment can be concatenated to the autarky assignment to yield a solution to the original problem.

The paper describes an algorithm for detecting these autarkies in parallel. In bit-parallel solving, an additional advantage of detecting autarkies

is that once an autarky has been revealed at a single bit position, it can be used at all other bit-positions as well.

Performance The paper presents a comparison between the original UnitWalk algorithm and UnitMarch. The results indicate a marked speed-up when UnitMarch is used rather than UnitWalk. Interestingly, for many benchmarks this holds already when March is run with single-bit vectors, i.e. with a single assignment. When March is used in a 32-bit configuration the speed-up becomes even more apparent.

E UnitWalk

Introduction Unit clause elimination is a vital aspect of complete solvers, but on the other hand, incomplete solvers are mainly based on local search and do not use unit clause elimination. UnitWalk is unique in the sense that it is an incomplete solver that combines local search with unit clause elimination. It propagates unit clauses as soon as they are encountered. Experimental results have shown that UnitWalk performs substantially less flips than WalkSat/TABU in almost all of the cases.

Algorithm The UnitWalk algorithm works as follows. Initially a vector A of size n (total number of variables) is assigned random truth assignments. Then inside a period, vector B is assigned random permutations of numbers 1 to n . Secondly, G is assigned the initial formula. Then for n iterations, the following steps are performed. While G contain a unit clause, a unit clause $\{x_j\}$ or $\{\neg x_j\}$ is randomly chosen and $A[j]$ is flipped if it did not contain the satisfied value and there was no conflicting unit clause. G is modified accordingly. If no unit clause was found in an iteration, then G is modified by reading the value from A indexed by $B[i]$. If after n iterations G is *TRUE* then the problem is satisfiable and A contains the solution. If no unit clause was encountered in G in any one of the n iterations, one random value in A is forcefully flipped. If at the end of a period, G is not *TRUE*, another period is repeated. If after a certain number of periods, G is still not *TRUE*, then A is once again assigned random values and the procedure restarted again. This restart of the whole procedure, with a new random assignment, is known as a try and is done because UnitWalk (like other local search methods) can get stuck into a local optimum. If after a certain number of tries, G is still not *TRUE*, then either the problem is unsatisfiable or it was too hard for UnitWalk to solve the problem. This is the reason why UnitWalk is probabilistically incomplete.

Some of the incomplete solvers are essentially incomplete, which means that for some initial assignments, the probability of hitting a satisfying assignment is strictly less than one. Due to this reason, such solvers require restarts. But in case of UnitWalk, the probability of hitting a satisfying assignment is one for any initial assignment and hence it is probabilistically approximately complete. Therefore, from a theoretical point of view, no restarts are required for UnitWalk and a solution for a satisfying formula can be obtained with only one try. In fact the authors of this paper, did use only one try for some of their experiments. However, for some formulas, progress towards the solution might take a very long time and therefore restarting with a new initial assignment is a better option.

Implementation A CNF formula is implemented as a list of clauses, with each clause containing its size and a link to an array of its literals. At each substitution, the size of the clause is reduced, with the corresponding literal being replaced by the last literal of the clause. Furthermore, for each variable, a list of clauses using that variable is maintained. Also, to allow quick finding of unit clauses, a list of indexes of unit clauses is maintained.

Optimizations Performance of UnitWalk can be improved by adding resolvents. Since large resolvents can slow down the performance, therefore only resolvents of size two have been used by the authors in their extension of UnitWalk. They add 2-resolvents after each substitution using the fast IncBinSat [13] method. Experimental results prove that the solver indeed performs better by adding the 2-resolvents.

Experiments have also shown that UnitWalk solves formulas that WalkSAT does not solve and vice versa. Therefore they are in a sense, complementary. Due to this reason, UnitWalk and WalkSAT are combined to solve a wider range of problems. The solver implemented by the authors alternates between UnitWalk and WalkSAT. While this implementation improves solution time for some formulas, it also slows down performance for others.