march

Towards a lookahead Sat solver for general purposes

master thesis M.J.H. Heule

Faculty of Electrical Engineering, Mathematics and Computer Science

Delft University of Technology

march

Towards a lookahead Sat solver for general purposes

master thesis M.J.H. Heule

February 2004

committee:

Dr. H. van Maaren

Prof. dr. ir. C. Roos

Dr. C. Witteveen

Acknowledgements

The process from my first introduction to the satisfiability problem to the development of the latest version of the solver march - described in this thesis - covers a period of more than two and a half years. During this time, I became more and more involved in the problem and its many approaches. Although march is a very personal project, it is also the result of teamwork, mutual inspiration and a careful allocation of skills. Therefore, I wish to thank the following people for their valuable contributions to the final result: Dr. Hans van Maaren, for his extensive insight in the satisfiability problem at large, his close personal guidance, and his mentorship and company during my first steps in the scientific world at the annual Sat competitions of 2002 and 2003. He made me understand that knowledge does not come from books, but from experience. I would like to thank Joris van Zwieten, for his friendship, his efficient, clean programming and his ongoing drive to order, stabilize and fine-tune the march solver. Many thanks to my friend Mark Dufour, for his manic attitude towards work, his lavish programming skills and his outstanding knowledge on optimizations of march for modern computer architecture. Finally, I would like to thank my dear friend Arjen van Lith for his textual suggestions, his editorial scrutiny and his personal support throughout the process of writing this thesis.

Abstract

The DPLL framework with a lookahead procedure that determines the branching variable is a powerful tandem to solve a wide range of satisfiability problems. It outperforms all other techniques on random CNF formulas. This thesis presents the solver march, which is based on this architecture and focuses on general fast performance. To achieve this goal, this thesis proposes a new kind of pre-selection heuristics which selects a small constant number of variables. Still, the most effective enhancements appeared to be a dramatic change of the data-structures which uses the reduction of the computational costs of the lookahead procedure as guideline. To increase the number of problems that could be solved by a lookahead solver, two forms of additional reasoning were integrated: Balanced addition of merely binary resolvents and equivalence reasoning, which merely costs some computational time in the pre-processor. The result is a version of march that solves most instances of the SAT 2003 competition benchmark set.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Ian Gent and Toby Walsh [11] quote the following diplomatic problem proposed by Brian Hayes as an introduction to the field of satisfiability:

> *"You are chief of protocol for the embassy ball. The crown prince instructs you either to invite Peru or to exclude Qatar. The queen asks you to invite either Qatar or Romania or both. The king, in a spiteful mood, wants to snub either Romania or Peru or both. Is there a guest list that will satisfy the whims of the entire royal family?"*

This problem can be represented as a propositional satifiability problem. We can express the three constraints by means of a propositional formula,

$$(P \vee \neg Q) \wedge (Q \vee R) \wedge (\neg R \vee \neg P)$$

where P, Q and R are Boolean variables which are true if and only if we invite, respectively, Peru, Qatar, and Romania. A solution to the problem is a satisfying assingment, an assignment of truth values to the Boolean variables that satisfies the propositional formula. In this case, there are just two satisfying assignments (out of the eight possible). These either assign P and Q to true and R to false, or assign P and Q to false and R to true. That is, we can either invite both Peru and Qatar and snub Romania, or we can invite just Romania and snub both Peru and Qatar.

Whilst propositional satifiability is a very simple problem, it is a cornerstone of the theory of computational complexity. Propositional satifiability was the first problem shown to be NP-complete [6]. Despite a quarter century of effort, it remains an open question whether problems in this class can be solved in polynomial time in the worst case. Many consider this question, the P=NP problem, to be one of the most important open problem facing theoretical computer science. As the best complete algorithms are exponential, NP-completeness is generally considerable to mark the boundary between tractability and intractability. Nevertheless, a large amount of research in recent years has shown that satisfiability problems can often be solved efficiently in practice.

This chapter continues by presenting the basic definitions of the satisfiability problem, followed by one elementary and two more sophisticated solving methods. Section 1.3 deals with two complexity related features. Finally, section 1.4 offers with the outline of this thesis.

1.1 The Sat problem

1.1.1 Basic definitions and terminology

A formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses ($\mathcal{F} = c_1 \wedge c_2 \wedge \ldots \wedge c_m$); each clause being a disjunction of literals ($c = l_1 \vee l_2 \vee \cdots \vee l_n$); and each literal is an atomic Boolean variable x_i or its negated form $\neg x_i$. The Sat problem deals with the question whether a CNF formula is satisfiable (has a Boolean solution) or not.

A literal x_i is satisfied if x_i is assigned to true, and unsatisfied if x_i is assigned to false. Likewise, literal $\neg x_i$ is satisfied if x_i if assigned to false, and unsatisfied if x_i is assigned to true. A clause c is satisfied if at least one of its literals is satisfied. Finally, Boolean formula $\mathcal{F}$ is satisfiable if there exists an assignment to the Boolean variables x_i, in such a way that all clauses in $\mathcal{F}$ are satisfied. Otherwise, $\mathcal{F}$ is called unsatisfiable. Despite this simple explanation of what the Sat problem deals with, actually solving large formulas efficiently is a rather complicated matter. Let us start with an example to explain the basic principles.

1.1.2 An example

Consider the example formula $\mathcal{F}_{\text{example}}$ below. Let us first try to satisfy this formula with some guessing.

$$\mathcal{F}_{\text{example}} = clause_1 \wedge clause_2 \wedge clause_3 \wedge clause_4$$

$$\begin{array}{rcc} clause_1 & = & a \vee b \vee c \\ clause_2 & = & a \vee \neg c \\ clause_3 & = & b \vee c \vee d \\ clause_4 & = & \neg a \vee \neg b \vee \neg d \end{array}$$

The first guess is that d is true. After assigning d to true, $clause_3$ gets satisfied and $clause_4$ is reduced to a binary clause, since it contains a literal with a negative reference to d. We refer to the simplification of the formula as a result of an assignment as *propagation*.

$$\mathcal{F}_{\text{example}} \cup \{d\} = (a \vee b \vee c) \wedge (a \vee \neg c) \wedge (\neg a \vee \neg b)$$

Since the satisfiability of $\mathcal{F}_{\text{example}}$ is still undetermined after propagating d, another guess is required. The second guess is setting c to false. Since variable c occurs once positive and once negative in $\mathcal{F}_{\text{example}} \cup \{d\}$, one clause is reduced in length, another becomes satisfied. Now, two clauses remain, both containing the variables a and b:

$$\mathcal{F}_{\text{example}} \cup \{d, \neg c\} = (a \vee b) \wedge (\neg a \vee \neg b)$$

Fixing a to true or false, results in forcing b to the opposite truth value. For example, assigning a to true forces b to false, which solves the formula:

$$\mathcal{F}_{\text{example}} \cup \{d, \neg c, a, \neg b\} = \varnothing$$

In general though, solutions will not reveal themselves by the simple guessing procedure above. The following paragraphs will explain how to extend this procedure to a complete solving method.

1.1.3 Effective branch variables

We refer to an effective branching variable as a variable that would, in respect to the other variables, cause a relatively large reduction if one would fix this variable and simplify the formula. Selecting an effective branching variable is an important aspect of *solving* (determining whether there exist a solution for) a formula. Recall the example of above. Here, d was selected first as a branching variable. Assigning d to true resulted in one satisfied clause and the reduction of another. If d were assigned to false, it would cause the same number of reductions.

It appears that d is not a very effective branching variable; the selection of all other variables results in a larger reduction. The most effective branching variable in $\mathcal{F}_{\text{example}}$ is a. Assigning it to true will satisfy two clauses and reduce one. Assigning it to false will even directly result in a solution! This is the case, due to the occurrences of *unit-clauses* (clauses of length one) in the latter reduction: The occurrence of a unit-clause in the formula, forces the only literal in that clause to be true. Consider $\mathcal{F}_{\text{example}}$ with a assigned to false:

$$\mathcal{F}_{\text{example}} \cup \{\neg a\} = (b \vee c) \wedge (\neg c) \wedge (b \vee c \vee d)$$

In $\mathcal{F}_{\text{example}} \cup \{\neg a\}$, $clause_2$ has been reduced to unit-clause $\neg c$. This forces c to be assigned to false, which results in :

$$\mathcal{F}_{\text{example}} \cup \{\neg a, \neg c\} = (b) \wedge (b \vee d)$$

The resulted formula also contains a unit-clause, namely b. Fixing b on its forced value true, results in a solution because it satisfies both remaining clauses:

$$\mathcal{F}_{\text{example}} \cup \{\neg a, \neg c, b\} = \varnothing$$

This iterative propagation as a result of eliminating unit-clauses is described in algorithm 1.1 IterativeUnitPropagation (IUP). We refer to branching on literal l_i as IUP($\mathcal{F} \cup \{l_i\}$). The fact that branching on $\neg a$ results in a solution, is denoted as IUP($\mathcal{F}_{\text{example}} \cup \{\neg a\}) = \varnothing$

Algorithm 1.1 IterativeUnitPropagation($\mathcal{F}$)

```
while F does not contain an empty clause and unit clause y exists do
    satisfy y and simplify F
end while
return F
```

1.2 Sat Solvers

A powerful framework to solve a Sat problem is the Davis-Putman-Logemann-Loveland procedure (DPLL) [8]. This framework is used by practically all Sat solvers (a program that solves Sat problems). This section explains the principles of the DPLL procedure, followed by the two most important modifications of the "pure" procedure.

The pseudo-code of DPLL as presented in algorithm 1.2 is the most commonly used abstract model of the framework: A recursive and depth-first search-tree procedure. Each node, the formula is simplified by calling the Iterative-UnitPropagation procedure. Note that after selecting a branch variable, at least one unit-clause is present. The pseudo code is presented in algorithm 1.1. This procedure fixes all unit-clauses in $\mathcal{F}$ on their truth value and then simplifies the formula: All clauses that become satisfied by these truth assignments are removed from the formula, and all complement occurrences of the literal involved in the other clauses are removed. This removal could result in an *empty clause* (i.e. every literal from a clause is removed, so it can never be satisfied). If this is the case, a backtrack operation is performed, restoring the formula to the one of its parent node.

If the satisfiability of the resulting formula is still undetermined after simplification, a branching variable is selected by the GetBranchVariable procedure. Here, no implementation of this procedure is provided, since the number of possible implementations is vast. A simple but not very effective example of this procedure returns the variable that occurs most frequently in $\mathcal{F}$. The selected branching variable is then added as a unit-clause to $\mathcal{F}$, first positive and then negative, followed by a recursive call of DPLL($\mathcal{F}$).

This procedure of simplifying, variable selection and unit clause addition is repeated until one of the following two conditions is reached: (1) All clauses are satisfied, meaning that the formula is satisfiable or (2) every possible truth assignment results in an empty clause, which qualifies the formula as unsatisfiable.

Algorithm 1.2 DPLL($\mathcal{F}$)

$\mathcal{F}$:= IterativeUnitPropagation($\mathcal{F}$)
if $\mathcal{F} = \varnothing$ **then**
  return "satisfiable"
else if $\mathcal{F}$ contains empty clause **then**
  return "unsatisfiable"
end if
x := GetBranchVariable()
if DPLL($\mathcal{F} \cup \{x\}$) = "satisfiable" **then**
  return "satisfiable"
else
  return DPLL($\mathcal{F} \cup \{\neg x\}$)
end if

1.2.1 Lookahead solvers

Generally, a lookahead solver does not differ much from the "pure" DPLL framework: It solely implements the GetBranchVariable procedure in a relatively complex but effective way: The Lookahead procedure. The pseudo-code of this procedure is found in algorithm 1.3. Many alternative branch variable selection procedures deal with counting occurrences of variables in the formula in a more or less "educated" manner. These are generally much cheaper than the Lookahead procedure. It appears that for many problems, an accurate prediction of the eventual propagation is required to select an effective branching variable. However, these counting procedures could hardly predict the iterative behaviour of propagating a variable. The the most reliable way of obtaining a prediction is to actually perform the propagation.

For each free variable x_i, the Lookahead procedure calls both IUP($\mathcal{F} \cup \{x_i\}$) and IUP($\mathcal{F} \cup \{\neg x_i\}$). We refer to IUP($\mathcal{F} \cup \{x_i\}$) during the Lookahead procedure as the *lookahead* on x_i. In contrast, the fixing of variables on the DPLL level is referred to as fixing on the *chosen path*. If the lookahead on either the positive or the negative literal results in a *conflict* (an empty clause after IUP), this literal is called a *failed literal*. If a solution exists for $\mathcal{F}$, the complement of the failed literal is fixed on true in this solution. For example, if $\neg x_i$ appears to be a failed literal, variable x_i is assigned to true in all possible solutions from this point onward. When the lookahead on both x_i and $\neg x_i$ result in conflict, the formula under consideration is unsatisfiable.

When both lookaheads on a variable do not result in a conflict, the lookahead will be evaluated. A lookahead evaluation function measures the difference between $\mathcal{F}$ and its reduction IUP($\mathcal{F} \cup \{x_i\}$). Such an evaluation function is referred to as Diff. A Diff could be based on all kinds of entities. Examples of such entities are the number of clause reductions, the number of satisfied clauses and the reduction of free variables. A widely used Diff is one that counts the newly created binary clauses in the resulting formula. This DIFF is proposed by Li [17]. The final heuristic function $\mathrm{H}(x_i)$, calculated as in line 12 of algorithm 1.3, determines which variable is returned. The MIXDIFF usually multipies both obtained DIFF's, but other functions are possible as well. The variable x_i with the highest $\mathrm{H}(x_i)$ is selected for branching.

A lookahead procedure has two characteristic advantages: (1) the detection of failed literals and (2) the selection of an effective branching variable - depending on the DIFF. The disadvantage of this procedure is that it is very expensive. Examples of fast lookahead solvers are satz [17], OKsolver [15] and kcnfs [9].

Algorithm 1.3 LOOKAHEAD( )

1: **repeat**
2:   **for** all free variables in $\mathcal{F}$ **do**
3:     $\mathcal{F}'$:= ITERATIVEUNITPROPAGATION($\mathcal{F} \cup \{x_i\}$)
4:     $\mathcal{F}''$:= ITERATIVEUNITPROPAGATION($\mathcal{F} \cup \{\neg x_i\}$)
5:     **if** $\mathcal{F}'$ contains empty clause **and** $\mathcal{F}''$ contains empty clause **then**
6:       **return** "unsatisfiable"
7:     **else if** $\mathcal{F}'$ contains empty clause **then**
8:       $\mathcal{F} := \mathcal{F}''$
9:     **else if** $\mathcal{F}''$ contains empty clause **then**
10:       $\mathcal{F} := \mathcal{F}'$
11:     **else**
12:       $\mathrm{H}(x_i)$:= MIXDIFF(DIFF($\mathcal{F}$, $\mathcal{F}'$) , DIFF($\mathcal{F}$, $\mathcal{F}''$))
13:     **end if**
14:   **end for**
15: **until** NONEWFAILEDLITERALSDETECTED( )
16: **return** x_i with highest $\mathrm{H}(x_i)$

1.2.2 Conflict driven solvers

A conflict driven solver uses a somewhat different DPLL framework. Its pseudo code is presented in algorithm 1.4. The most significant change with respect to algorithm 1.2 is that no chronological backtracking is performed: Whenever a conflict occurs during the solving phase, it is analysed using the ANALYSECONFLICTS procedure. This procedure determines the depth of the search-tree to which one should backtrack in order to resolve the conflict. Besides this, ANALYSECONFLICTS also adds one or more conflict clauses describing the resolved conflict. When backtracking beyond the root of the tree is called for, a conflict exists even without any branching. This indicates that the problem is unsatisfiable.

Since AnalyseConflicts adds conflict clauses to the formula, it could happen that after backtracking to *blevel*, an empty clause is derived again after $\mathcal{F} := \mathrm{IUP}(\mathcal{F})$. Therefore, a second while-loop is necessary to backtrack to the lowest possible depth. Examples of fast conflict driven solvers are zchaff [31], berkmin [13] and satzoo [26].

Algorithm 1.4 ConflictDrivenDPLL($\mathcal{F}$)

```
while TRUE do
  l_i := GetBranchLiteral( )
  if an l_i is selected then
    F := IterativeUnitPropagation(F ∪ {l_i})
    while F contains empty clause do
      blevel := AnalyzeConflicts( )
      if blevel = 0 then
        return "unsatisfiable"
      else
        BackTrack( blevel )
        F := IterativeUnitPropagation(F)
      end if
    end while
  else
    return "satisfiable"
  end if
end while
```

1.3 Complexity related features of Sat

Problems for which a possible solution can be verified within polynomial time have *nondeterministic polynomial time* (NP) complexity. Cook proved [6] that the Sat problem is among the most difficult NP-problems and is therefore called *NP-complete*. Many problems are NP-complete and for none of them an efficient solving procedure is known. If an efficient solving procedure could be developed for one of them, then all could be solved efficiently.

1.3.1 Transformations

Every NP-complete problem could be transformed to another in polynomial time. Some of these transformations are natural, especially if the problems are closely related, but other transformations are very artificial. The Sat problem is of great interest, since for many problems a natural transformation to Sat is known. Examples of these problems are graph colouring problems, bounded model checking [4], the stable marriage problems [12] and factoring problems [24].

SAT solvers can be used to solve these problems: Transform the problem into SAT, solve the result using a SAT solver and retransform the solution to the original problem space. Because of these wide applications, SAT "world championships" have been organised in recent years. During these competitions SAT problems are divided into three categories: Industrial, handmade and random. The first category consists of benchmarks that are used for industrial applications. Currently, it involves predominantly bounded model checking, formal verification and planning problems. The second category deals with benchmarks that are designed to obstruct the performance of current SAT solvers. These difficult, but often small instances require specific reasoning to be solved in sensible period of time. The last category consists of random instances. These are discussed in the following paragraph.

The fact that many combinatorial problems can be naturally formulated as a SAT problem, does not necessarily imply that SAT solvers provide the fastest methods for solving these problems. Most likely they do not: Problem specific information is obscured during the transformation to SAT. A program with access to this information could theoretically be faster using problem specific solving methods. In practice however, problem type specific solvers often require significant investments. Recent optimizations performed on SAT solvers have resulted in substantial performance gains and new developments arise with increasing world wide effort. Therefore, outperforming SAT solvers will become even more difficult in the future.

1.3.2 Uniform random CNF

A special kind of problem is the *uniform random 3*-SAT (in short random 3-SAT), in which 3-SAT denotes that all clauses initially have length 3. These formulas are constructed in such a way that all variables have an equal chance of being selected for a literal. The chance of a positive or negative literal is 50%. A useful random 3-SAT generator is the OKgenerator [16]. Random formulas are often used to evaluate the performance of algorithms. However, it appears that due to some properties of random 3-SAT, only formulas of a specific clause-to-variable ratio are useful for evaluation purposes.

The *density* of a formula is the ratio of the number of clauses to the number of variables. The density plays an important role in the properties of random 3-SAT: Experimental results [27] show that random 3-SAT benchmarks with a density lower than approximately 4.26 are almost all satisfiable, while benchmarks above this density are nearly all unsatisfiable. The higher the number of variables, the steeper this threshold appears to be. Figure 1.1 shows this threshold for small formulas.

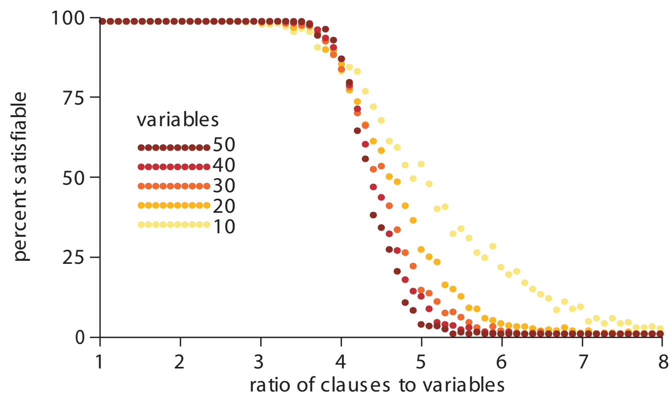


Figure 1.1 — Random 3-Sat threshold for small formulas

The density of a formula also has a profound influence on the computational complexity for solving random 3-Sat formulas. Experimental results [27] show that using a DPLL Sat solver, high peaks in running time are noticed when solving formulas close to the threshold. In figure 1.2, this behaviour is shown for small random 3-Sat formulas. Recent studies [5] show that the location of these peak running times depends on the solving method (eventually not DPLL based methods) and that it is therefore not necessarily close to the threshold.

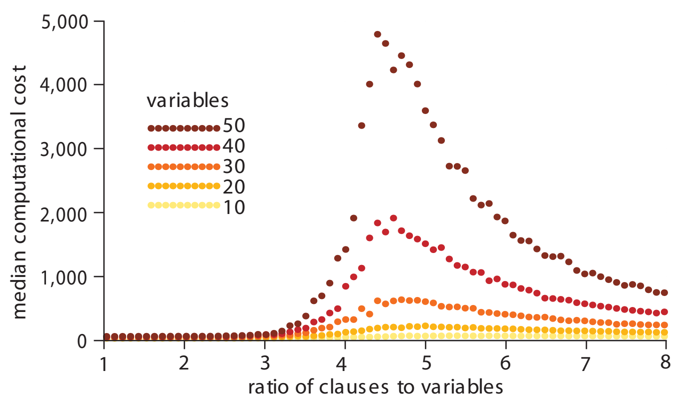


Figure 1.2 — Computational time required to solve small random 3-Sat formulas

Because hard benchmarks of various sizes can be easily generated, the study of random 3-Sat formulas seems useful for developing effective solving procedures. In recent years, several Sat solvers have been optimized towards solving these hard instances. The two most successful examples of these solvers are satz [17] and kcnfs [9], both DPLL lookahead solvers.

1.4 Outline

The further outline of this thesis is as follows: Chapter 2 offers an historical overview of the march solver, along with our focus and the motivations of the improvements made during its development. The next four chapters describe the features of the current version of march in detail: Chapter 3 deals with the pre-processing features of our solver. Pre-processing the original formula is essential in fast Sat solvers, because it can significantly reduce its size and could in some cases even solve it. Heuristics to perform two important procedures of march only partially, are described in chapter 4. The largest performance gains are achieved by enhancements to the data-structures during the development of march. The current data-structures are described in chapter 5. Chapter 6 offers an explanation of the applied equivalence reasoning in march. Results of march 's performance in general and of equivalence reasoning in particular are presented in chapter 9. Finally, chapter 10 offers conclusions and some ideas for future research.

Chapter 2

March

The development of the march solver was first and foremost a fascinating project. It originated as a practical examination of the 'satisfiability' course at TU Delft. In developing the solver march, the decision to participate at the SAT 2002 competition in Cincinnati added an extra competitive drive: march finished second in the random category, yet it did a rather poor job in both others. Regardless of - or perhaps due to - the good results in the 2002 competition (see section 7.1), we changed and enhanced march dramatically in order to perform even better the next year and beyond.

Afterwards, it was easy to find three explanations why march was only competitive on random benchmarks: First, the solver was merely optimised for these particular instances. Second, march's data-structures suffered under massive numbers of clauses and variables. Third, additional reasoning that could speed up non-random instances was not implemented. However, before our participation in the competition, we expected that optimizations on random formulas would also raise our opportunities in both other categories of the field. Since we turned out to be wrong, we considered that the future focus would be to develop a solver with an fast performance in general.

This decision had many consequences for the design of march. We kept the lookahead architecture, considering it essential for a fast general solver, but this also posed a challenge. Within the lookahead environment we made two major changes regarding the branching strategy: Both the pre-selection heuristics and the lookahead evaluation function were replaced. Also, the data-structures required several reforms. Here, efficient use of memory during the lookahead phase was crucial. Finally, some additional reasoning was integrated. This may not have resulted in a performance boost on random instances, but was clearly the case for many problems outside of this domain.

2.1 Focus

Why engineering a satisfiability solver? What would be its purpose? As mentioned above, we tacitly assumed that optimizations on performance on random CNF benchmarks would eventually result in a solver that performs well all over the benchmark spectrum. Since our assumptions seemed not very well grounded, it forced us to take a position regarding this question: where do we want to excel?

Should we concentrate on random CNF benchmarks, or focus on general performance? The last option would probably keep us from success in any of the categories of the competition. The addition of new techniques and modifications to speed up the lookahead would probably favour only one of both options.

It seemed to be a choice between fundamental research and the practical application of it: As described above, the random CNF domain in scientifically interesting, because its sheds light on the complexity of satisfiability. The complexity of an solver can be calculated using the average time it takes to find a solution. However, the actual solution of a random CNF benchmark is of no interest. On the other hand, problems outside this domain often have a specific application. Therefore, the search is pre-eminently focused on a solution. The solution time is not primarily important, as long as it is "sensible". A notion of the complexity is hard to grasp, because general performance is hard to measure: How to objectively define a representative group for general purpose?

The satisfiability problem holds a wide range of applications, so research on a general Sat solver is fruitful. If one compares a solver which is specialized in random CNF instances with a solver optimized for general use, the latter would be more suitable for practical use. However, for only a small number of applications it would be commercially interesting to develop a program that could outperform a state-of-the-art Sat solver. In many cases, the choice for a general purpose solver would be justified, since many problems could be translated into a Sat problem.

Also if one favours fundamental research of the satisfiability problem, the choice for a general purpose solver can be defended: By optimizing solving techniques for general use, one could dispose of optimizations focused on particular benchmarks families, since these serve no general purpose. One could almost speak of a falsification of the techniques within the CNF domain. Techniques that perform well on almost all possible benchmarks provide insight in the principles of effectively solving Sat problems.

An apparent reduction in computational complexity on random CNF benchmarks is not directly expected, but certainly possible. However, the most important aims of the improvements to our solver our solver consist of a faster performance on practical applications and a better understanding of effective solutions.

To increase the general purpose performance of march, we focused on a reduction of the number of parameters and on a design that serves the average performance, even if this means a minor slowdown on random benchmarks. The latter is more important than the first: On average, a multiple parameter technique could outperform a single parameter technique. Still, there must be convincing experimental evidence supporting this better performance.

2.2 Branching strategies

Three different heuristics were applied in march to obtain an effective branch variable: The *MAR-heuristics* developed by Van Maaren and Warners [20] and two lookahead evaluation functions. Since the first is not further discussed in this thesis, a small description is offered later in this section. Of the two lookahead evaluation functions, the first was developed by Li [17]. It counts the newly created binary clauses in the formula during a lookahead. Throughout this thesis, this evaluation function is referred to as *the elementary* DIFF. The difference with the second evaluation function, developed by Dubois and Dequen [9], is that the latter weighs the newly created binary clauses. This evaluation function is referred to as *the complex* DIFF. Both are discussed more in detail in section 4.1. All three heuristics were integrated in the PARTIALLOOKAHEAD procedure. This procedure is first presented in general, followed by a paragraph with a timeline showing when these heuristics were embedded in march.

2.2.1 MAR heuristics

The first stable version of march followed a branching stategy that selected the variable with the highest MAR-value. This value is based on heuristics developed by Warners and Van Maaren. These heuristics are described in such a way, that march's computations could be repeated by other people. For details about the geometrical motivations we refer to [20].

The calculations of the MAR-values are based on the relative occurrences between variables. These relations are stored in two matrices $\mathcal{Q}_2$ and $\mathcal{Q}_3$ for the binary and ternary clauses, respectively. Both matrices are 2-dimensional and both row- and column-length equal the number of variables.

$\mathcal{Q}_2$ is filled with all the binary clauses. For every clause $(\neg)x_i \vee (\neg)x_j$ a number is added to $\mathcal{Q}_{2,ij}$. If the literals have the same sign, then 1 is added, otherwise -1. The diagonal elements $\mathcal{Q}_{2,ii}$ are equal to the total number of occurrences of variable x_i, positive and negative, in binary clauses. The filling of $\mathcal{Q}_3$ takes place almost identically, but here, the ternary clauses are used. Since three distinct pairs of variables are possible in a ternary clauses, all three pairs are added to this matrix.

For the actual calculations of the MAR-values matrix $\mathcal{Q}$ is used. This matrix is the sum of $\mathcal{Q}_3$ and a weight times $\mathcal{Q}_2$. The weight indicates the importance of the binary clauses in regard to the ternary clauses. Experimental results show that a weight equal to 11 results in optimal performance on random 3-SAT benchmarks near the threshold [20]. Below, examples of $\mathcal{Q}$, $\mathcal{Q}_2$ and $\mathcal{Q}_3$ are offered. These are filled with formula $\mathcal{F}_{\mathcal{Q}}$. The final calculation of the MAR-values is obtained by using function f_{mar}.

$$\mathcal{F}_{\mathcal{Q}} = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3)$$

$$\mathcal{Q}_2 = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathcal{Q}_3 = \begin{bmatrix} 2 & 0 & -2 \\ 0 & 2 & 0 \\ -2 & 0 & 2 \end{bmatrix}$$

$$\mathcal{Q} = w * \mathcal{Q}_2 + \mathcal{Q}_3 = \begin{bmatrix} w+2 & w & -2 \\ w & w+2 & 0 \\ -2 & 0 & 2 \end{bmatrix}$$

$$f_{mar}(x_i) = \mathcal{Q}_{ii} - \sum_{j \neq i} \frac{(\mathcal{Q}_{ij})^2}{\mathcal{Q}_{jj}}$$

2.2.2 Partial lookahead

The LOOKAHEAD procedure is rather expensive. A possibility to reduce computational costs is to restrict the number of variables that enter this procedure. However, this will generally result in a decreased detection of failed literals, which makes the LOOKAHEAD procedure less powerful. Performing lookahead on only a subset of the free variables is referred to as *partial lookahead*. This set contains the variables that enter the LOOKAHEAD procedure, and is denoted as $\mathcal{P}$.

Pre-selection heuristcs (heuristics that select the variables to enter the lookahead) are useful to optimise the performance gained by partial application of the LOOKAHEAD procedure. Like the quality of the DIFF, we determine the quality of pre-selection heuristics to their ability to select an effective branching variable. Both the quality of pre-selection heuristics and of a DIFF are hard to measure from a general point of view. They may be optimized for a certain family of problems, but outside that family their quality is unknown.

When the quality of a DIFF is low, it may be compensated by a high quality of the pre-selection heuristics: When only a few variables are selected for the lookahead and they are all effective branching candidates, the “bad” performing DIFF could choose, in the worst case, the least effective candidate. A smaller set of lookahead variables could, in this case, increase the effectivity of the branching variable.

On the other hand, if the quality of the DIFF is high, it may compensate the lack of quality of the pre-selection heuristics: When the selected set of lookahead variables is large enough, the DIFF could still select an effective candidate for branching.

Algorithm 2.1 PARTIALLOOKAHEAD( )

```
Compute 𝒫
repeat
  for each variable x_i in 𝒫 do
    ℱ' := ITERATIVEUNITPROPAGATION(ℱ ∪ {x_i})
    ℱ'' := ITERATIVEUNITPROPAGATION(ℱ ∪ {¬x_i})
    if  ℱ' contains empty clause and ℱ'' contains empty clause  then
      return  "unsatisfiable"
    else if ℱ' contains empty clause  then
      ℱ := ℱ''
    else if ℱ'' contains empty clause  then
      ℱ := ℱ'
    else
      H(x_i) := DIFF(ℱ, ℱ') × DIFF(ℱ, ℱ'')
    end if
  end for
until NoNewFailedLiteralsDetected
return  x_i with highest H(x_i)
```

Eversince the earliest version of march that uses pre-selection heuristics, we have chosen to fix the size of $\mathcal{P}$ to 10% of the initial number of variables. This was not the result of an actual optimazation, but merely an educated guess based on the following considerations:

- the smaller $\mathcal{P}$, the lower the computational costs of the lookahead phase.
- the larger $\mathcal{P}$, the smaller the tree-size because of detected failed literals.
- if $\mathcal{P}$ is small enough,
  "good" pre-selection heuristics may compensate a inadequate DIFF.
- if $\mathcal{P}$ is large enough,
  a "good" DIFF may compensate inadequate pre-selection heuristics.

2.2.3 Timeline

The first version of march was not a lookahead solver. Instead, it simply selected for each node the variable with the highest MAR-value to branch on. Experiments with this version showed that using the MAR-heuristics on random 3-SAT formulas resulted in a small *node-count* (total number of nodes visited during solving a formula) compared to other non-lookahead techniques. However, the lookahead-techniques outperformed this version by far. Therefore, march was enhanced with lookahead: It used an implementation of the PartialLookahead procedure that applied the MAR-heuristics as pre-selection heuristics and the elementary Diff for the lookahead evaluation function. This version participated at the Sat 2002 competition in Cincinnati.

Halfway between the Sat 2002 and Sat 2003 competition, we decided to replace the MAR-heuristics. This was motivated by three observations: First, after many data-structure optimizations, calculation of the MAR-heuristics appeared to be the bottleneck of computation on many benchmarks, making a full lookahead (not using pre-selection heuristics) on several benchmarks even faster. Second, the MAR-heuristics prevented us from developing a solver with a general good performance: The weight used to construct $\mathcal{Q}$ appeared to have a major influence on the eventual speed, but its optimal value varied heavily with respect to the different benchmarks families. Although various models have been examined, none of them resulted in satisfying results. However, recently a procedure is developed to obtain an effective weight [21]. Third, additional reasoning appeared to result in favourable effects on many benchmarks. Yet, our optimized data-structure to calculate the MAR-values was not able to integrate this additional reasoning into the MAR-heuristics. In theory, pre-selection heuristics that know how to use this additional reasoning are more suited, since they have more "knowledge" about the formula.

The MAR-heuristics were replaced by an approximation function of the elementary Diff, which is described in detail in paragraph 4.1.2. These preselection heuristics resulted in a comparable node-count on random 3-Sat formulas, while in general it performs significantly better than our best implementation of MAR-heuristics. Besides, this feature it is not troubled with the three problems that bothered the MAR-heuristics: (1) It has relatively low computational cost; (2) it does not contain any parameters; and (3) it puts additional reasoning to good use. But most importantly: it boosts performance considerably.

After the Sat 2003 competition, we experimented to obtain a more effective lookahead evaluation function. It appeared that the evaluation function proposed by Dubois and Dequen [9] resulted in the fastest performance. The most significant gains were achieved on random 3-Sat formulas, but on many other benchmarks a clear speed-up was also noted. This feature makes this Diff suitable for a general purpose Sat solver. Since march applies some changes to the formula during solving, this Diff is implemented slightly differently. Details about this implementation are described in paragraph 4.1.3.

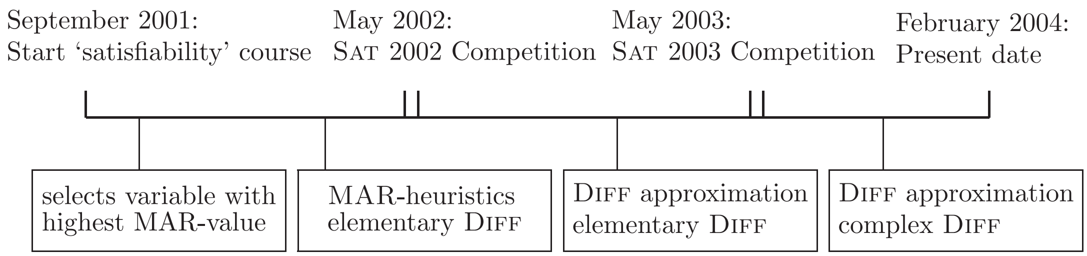


Figure 2.1 — Applied pre-selection heuristics and lookahead evaluation functions over the course of time.

2.3 Data-structure optimisations

As soon as the first version of march was stable, much effort was spent on optimization of the data-structures. Although this does not change the complexity of the eventual solving algorithm, optimized data-structures could account for a considerable constant speed-up factor. All data-structures in the current version of march are described in chapter 5. The section below deals with the most important aim behind the optimizations, efficient memory and cache usage, and with the optimizations for the data-structure to calculate the MAR-values. This data-structure was removed as a result of the replacement of the pre-selection heuristics (see paragraph 2.2.3).

2.3.1 Cache optimisations

The impact of a chosen data-structure on the performance of a SAT solver, is greatly influenced by the architecture of modern computers, which consists of a small, fast cache and large, relatively slow memory. Whenever the solver needs data that is not available in the cache - a socalled *cache miss* - it needs to fetch it from the memory. This takes valuable time. Zhang estimates that even for his fast solver zchaff, the retrieval of data costs about 80% of the total time needed to solve a problem [33]. Subsequently, optimizations to reduce the amount of cache misses could increase the performance of a solver considerably.

Zhang and Malik also describe various optimizations for a conflict driven solver [33]. However, the differences between conflict driven solvers and lookahead solvers are enormous. Therefore, other changes to the data-structure are required to reduce the number of cache misses.

All procedures suffer more or less from cache misses. It seems reasonable to reduce this performance drain for those procedures that are responsible for the highest computational costs, even if this might slowdown some (cheap) others. To get an impression which procedures cost most, the performance of march was

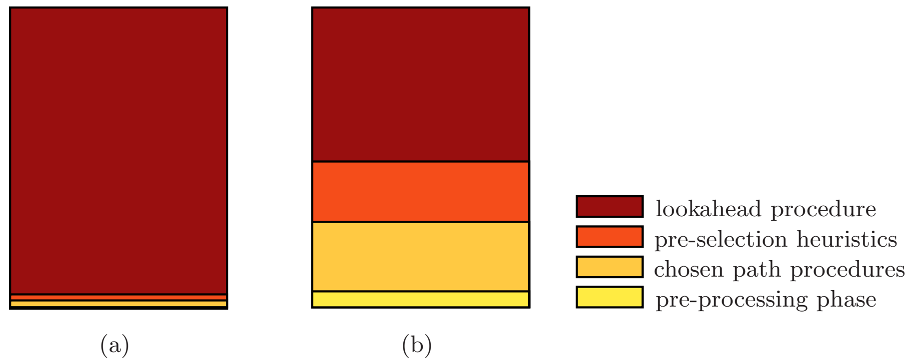


Figure 2.2 — Profiles of two march versions on qg07-12.cnf. (a) Version which participated at SAT 2002 and spends 941 seconds, and (b) the current version, which spends 8 seconds.

profiled.

Profiles describe the relative computational cost of the different procedures a solver executes. An unbalanced profile points to the procedure where optimisations could achieve large gains. We profiled march on the qg7-12 benchmark. This problem, one of the most difficult of the quasigroup [34], grasped our attention since it was solved in 200 nodes which took 941 seconds: Almost 5 seconds for each node! For comparison, march handles almost 1000 nodes a second on random 3-SAT formulas. The result of this profile is showed in Figure 2.2 (a): Over 95% of the total computational time was spent during the lookahead phase.

Judging from these and other profiles, it became clear that the costs of the lookahead procedure were the main reason for the lack of performance on both hand-made and industrial benchmarks. Reducing these costs seemed crucial for a faster general performance. All changes and enhancements of the data-structures were designed to reduce these costs. To this extent, we aimed to reduce both the total memory used during the lookahead phase, as well as the required memory for specific sub-procedures.

The resulted solver (the current version of march), showed enormous performance gains on those problems with similar unbalanced profiles. One of the benchmarks that appeared to profit most was qg7-12. The computational time required to solve this instance dropped from 941 to 8 seconds, while the number of nodes was still around 200. This version's profile on this benchmark looks also much more balanced (see Figure 2.2 (b)).

2.3.2 MAR data-structure

To apply the MAR-heuristics successfully on larger structured problems, it was crucial to replace the straightforward data-structure used to calculate the MAR-values: On problems outside the random CNF domain, the number of variables can be large. The necessary variable-variable matrix $\mathcal{Q}$ to obtain the MAR-values, could grow larger than the available memory. If, for instance, the number of variables is 10,000, the matrix would consist of 100,000,000 entries. Even if it is possible to allocate such an elaborate matrix, than the costs of updating it will be overwhelming.

The number of possible non-zero elements in the matrix is relatively small. The length of clauses determine the possible non-zero elements: Each pair of two literals that could be made in a clause is a possible non-zero entry. In the worst case, the total number of non-zero elements could be calculated according to equation (2.1). Here, $Clength_i$ refers to the length of clause i. For random 3-Sat formulas on the threshold, this means that only $\frac{13}{\#variables}$ of the entries is possible non-zero. Storing the matrix in lists containing only the possible non-zero-elements will substantially reduce memory spent. The costs to update the matrix will slightly increase; loops through the list are required since a direct indexation is no longer possible.

To reduce the number of possible non-zero elements, a 3-Sat translation is performed on the formula after initialisation. The current translator is described in section 3.3. Due to the use of the 3-Sat translator, the number of possible non-zero items decreases: e.g. Ninety distinct pairs could be made with a clause containing ten literals. Yet, only 24 pairs could be made with the eight clauses that are the result of the translation. Now, the worst case number of possible non-zero entries is equal to equation 2.2. In this equation, $Clength_i$ refers to the original length of clause i.

$$\sum_{i=0}^{\#clauses} \frac{(Clength_i - 1)(Clength_i)}{2} \tag{2.1}$$

$$\#BinaryClauses + \sum_{i=0}^{\#clauses} 3 \times (Clength_i - 2) \tag{2.2}$$

The lists and the 3-Sat translator made the calculation of the MAR-values significantly faster on larger problems. Still, it was not fast enough: Recall the performance of `march` on `qg7-12`. It seems clear in the profile that the MAR-heuristics are calculated quickly, though in absolute numbers they appear to take 18 seconds, while the heuristics of the current version only spend 1.2 seconds. Since both result in a comparable node-count, the new pre-selection heuristics are clearly an improvement.

2.4 Additional reasoning

Additional reasoning plays an important role in increasing the general performance of the solver: Although random 3-SAT does not seem to profit from most forms of additional reasoning, and sometimes even suffers from it, many benchmarks outside the random CNF domain show significant speed-ups. Various forms of additional reasoning are added over time. Addition of resolvents is an important example, and will be discussed in two debates in section 3.4 and 4.2. Another variant of additional reasoning - the detection of *necessary assignments* - will be discussed, along with an introduction to the most characteristic feature of march, *equivalence reasoning*.

2.4.1 Necessary Assignments

During the SAT 2002 conference in Cincinnati, we attended a presentation of Marques-Silva [22] of a method to increase the number of detected forced variables, called *necessary assignments*. This method is best explained by an example: Consider the formula $\mathcal{F}_{\text{NA}}$ below. A lookahead on $\neg a$, $\text{IUP}(\mathcal{F}_{\text{NA}} \cup \{\neg a\})$, results in the unary clauses b, $\neg c$ and d. Likewise, $\text{IUP}(\mathcal{F}_{\text{NA}} \cup \{a\})$ results in unary clause d. Thus, both implications $a \rightarrow d$ and $\neg a \rightarrow d$ exist. Since a is a boolean variable, it is either true or false. Therefore d must be true and is denoted a *necessary assignment*.

$$\mathcal{F}_{\text{NA}} : (a \vee b) \wedge (\neg b \vee \neg c) \wedge (\neg a \vee d) \wedge (a \vee c \vee d)$$

There are various ways to detect necessary assignments. The simple algorithm used in march is presented in DETECTNECESSARYASSIGNMENTS(DNA). This procedure is performed after the lookahead on literal l_i, but in advance of restoring the changes made during this lookahead. For every binary clause $l_i \vee l_j$ that occurs in the formula, a check is performeed whether l_j is a newly created unary clause. If this is the case, l_j is a necessary assignment, since l_j is a unary clause after $\text{IUP}(\mathcal{F} \cup \{l_i\})$, $l_i \rightarrow l_j$, and because of $l_i \vee l_j$, $\neg l_i \rightarrow l_j$. To integrate this detection algorithm in the lookahead procedure: Just replace in algorithm 1.3 and 2.1 line $\mathcal{F}' := \text{IUP}(\mathcal{F} \cup \{x_i\})$ with $\mathcal{F}' := \text{DNA}(\mathcal{F}, x_i)$ and line $\mathcal{F}'' := \text{IUP}(\mathcal{F} \cup \{\neg x_i\})$ with $\mathcal{F}'' := \text{DNA}(\mathcal{F}, \neg x_i)$.

Algorithm 2.2 DETECTNECESSARYASSIGNMENTS($\mathcal{F}$, l_i)

```
F_{l_i} := IUP(F ∪ {l_i})
for (l_i ∨ l_j) ∈ F do
    if l_j is unary clause in F_{l_i} then
        F := IUP(F ∪ {l_j})
    end if
end for
return F_{l_i}
```

Especially in a solver with a partial application of the lookahead procedure, the detection of necessary assignments could reduce the size of the formula. When performing a full lookahead procedure, the detection of necessary assignments could result in a less significant reduction, since many necessary assignments would also be detected by failed literals occurrences. Moreover, when performing a full lookahead while all constraint resolvents are added (a topic discussed in paragraph 4.2.2), all possible detected necessary assignment would be forced by the detected failed literals.

2.4.2 Equivalence reasoning

Another form of additional reasoning we became occupied with, was the two-phase algorithm by Warners and Van Maaren [30] to solve the `parity32` family. The benchmarks in this family were labelled as one of the ten Sat challenges [28], and they contain many so called *equivalence clauses*. By extracting these clauses and applying customized reasoning, *equivalence reasoning*, Warners and Van Maaren were the first to solve these instances.

The two-phase algorithm does a considerable amount of reasoning during the pre-processing (first) phase, and a minimal amount of equivalence reasoning during the solving (second) phase. These features made it ideal for integration into march: Only benchmarks with equivalence clauses would require a more costly pre-processing, but they are also the only ones that benefit. On the other hand, only a minor slowdown is expected during the solving phase due to the integration of the second phase of the algorithm.

Just before the Sat 2003 competition, we started integrating this two-phase algorithm in march. Due to this integration, twenty additional problems were solved during this competition. Afterwards, we significantly extended the algorithm: Some enhancements were implemented in the first phase (see paragraph 6.2.3) and several in the second phase (described in section 6.3 and 6.4). With its increased equivalence reasoning, march was able to solve another sixty additional problems of the Sat 2003 competition set (see section 7.2).

Chapter 3

Pre-processing

Pre-processing the boolean formula after initialisation is crucial to solve many problems using a SAT solver. The most essential pre-processing procedure is to simplify the given formula by fixing forced variables and replace equivalent literals with each other. Simplifying the formula does not change the solving procedure in itself, but its reduction of free variables and active clauses does result in a significant speed-up.

The three other procedures used in march are not that obvious; they all yield favourable and unfavourable effects: The first procedure performs equivalence reasoning. Only two other solvers, eqsatz [18, 19] and lsat [23], perform this procedure. The disadvantage of equivalence reasoning lies in a notable performance drain on benchmarks that do not profit from it. By concentrating the majority of computational costs in the pre-processor, this pain is heavily reduced during the solving phase. Even those benchmarks that show a considerable speed-up due to this additional reasoning take advantage from this shifted center of computation.

The second - and most obscure - procedure is a 3-SAT translator. To our knowledge, no other solver uses such a translator or even considers it. Since the number of clauses and literals are generally much larger after the translation, most procedures show a slowdown. However, the data-structure could be optimised substantially for the translated formula, which results in an overall performance gain on most benchmarks.

The last procedure of the pre-processing adds resolvents to the formula. Reduced performance is expected for a full addition. However, Bacchus' HyperBinRes shows that even a full addition of binary resolvents, could boost performance on various benchmarks [2]. In march, a small number of binary and ternary resolvents is added during the pre-processing, which appears optimal for a lookahead solver.

3.1 Simplifying formula

This paragraph discusses four elementary procedures to simplify the formula during the first phase of pre-processing.

On many of the benchmarks, the formula contains unary clauses after initialisation. The propagation of these unary clauses by calling IterativeUnitPropagation($\mathcal{F}$) could significantly reduce the number of active clauses and free variables.

Another useful procedure is the propagation of *binary equivalences*. Binary equivalences $l_j \leftrightarrow l_i$ could be easily detected in the CNF and propagated (replace literal l_i by l_j since they are equivalent): When both binary clauses $l_i \vee \neg l_j$ and $\neg l_i \vee l_j$ occur in the formula, literals l_i and l_j are equivalent. Therefore, all literals l_j could be replaced by l_i and all $\neg l_j$ by $\neg l_i$. In the same way, the literals l_i and $\neg l_j$ are equivalent when both binary clauses $l_i \vee l_j$ and $\neg l_i \vee \neg l_j$ occur. In this case, all occurrences of $\neg l_j$ could be replaced by l_i and all l_j by $\neg l_i$. All possible replacements of binary equivalences are executed by procedure BinaryEquivalencePropagation.

Algorithm 3.1 BinaryEquivalencePropagation($\mathcal{F}$)

```
while non-propagated binary equivalence l_i ↔ l_j occurs in F do
   replace l_i by l_j
   replace ¬l_i by ¬l_j
end while
return F
```

As a result of the propagation of binary equivalences, some clauses may have multiple literals that refer to the same variable. In case all these literals share the same sign, all but one should be removed. On the other hand, when they occur both positive and negative in the same clause, this clause is always true and should therefore be removed from the formula.

To detect such occurrences, we sort all literals within each clause in an ascending order, based on the variable it refers to. During the sorting, some clauses are shortened as result of multiple occurrences of the same literal within one clause. These shorter clauses may be new unary clauses, or they form new binary equivalences. This procedure of sorting and shortening is called SortLiterals.

Another result of replacing equivalent literals is the occurrence of identical clauses in the formula. To recognize these clauses in march, all clauses are sorted in alphabetical order. In such an arrangement, identical clauses are easily detected: they are placed directly above each other. Removing them will simplify the formula even further. The procedure that sorts the clauses and removes identicals is called SortClauses.

Algorithm 3.2 SimplifyFormula($\mathcal{F}$)

repeat
  $\mathcal{F}$:= SortLiterals($\mathcal{F}$)
  $\mathcal{F}$:= IterativeUnitPropagation($\mathcal{F}$)
  $\mathcal{F}$:= SortClauses($\mathcal{F}$)
  $\mathcal{F}$:= BinaryEquivalencePropagation($\mathcal{F}$)
until No binary equivalences were found during this iteration

Because the SortLiterals procedure may result in new unary clauses or binary equivalences, the procedures described above are executed iteratively until no new binary equivalences occur. After these iterations the formula may be a fraction of its original size. This will not affect the functioning of the solver, but a reduced formula can result in a considerable speed-up: In some cases, the SimplifyFormula procedure may even determine the satisfiability of the formula.

3.2 Equivalence reasoning

All of the march variants perform equivalence reasoning during pre-processing. Many benchmarks in the DIMACS suite [14] can easily be solved during the pre-processing phase when using equivalence reasoning. Even if one does not intend to use equivalence reasoning while solving, it is strongly recommended in pre-processing because it may either quickly solve the formula at an early stage, or create new unary clauses and binary equivalences that drastically simplify the formula. Details on this subject are described in the section 6.2.

3.3 3-Sat translation

The 3-Sat translator, a procedure that reduces the lenght of all clauses until none of them has a length > 3, is a heritage of the MAR-age: To effectively calculate the MAR pre-selection heuristics, a 3-SAT translator seemed crucial. Yet, after the first competition, we replaced these heuristics by an approximation function of the lookahead evaluation (see paragraph 2.2.3 for details). In this way, the 3-Sat translator was no longer essential to a fast performance.However, due to the use of the 3-Sat translator, the data-structure could be optimized in many ways. The most important optimazation involves the use of the local literal storage structure (see paragraph 5.3.3). Removal of the translator results in the removal of these optimasations.

Also, we were able to tackle the largest disadvantage of a 3-Sat translator in a lookahead solver: It appears that to achieve fast performance, the reduction of originally ternary clauses, should be weighted as more important than the reduction of ternary clauses obtained by translation. However, since the applied elementary Diff simply counts all ternary clauses, it is expected that this results in a performance drain. Yet, with the complex Diff presented in paragraph 4.1.3, new binary clauses from translated clauses are weighed far less than original clauses, since the dummy variables in the translated clauses occur only sporadically in the CNF.

With this in mind, we decided to give the 3-Sat translator the benefit of the doubt. To increase the change of a netto performance gain, an efficient translator is used that consists of two phases.

3.3.1 A Less Redundant Translation

The first phase of the 3-Sat translator is added just after the competition of Sat 2002, because we experienced that the initial, straightforward translation resulted in redundancies and a loss of performance. To reduce these redundancies, we added the procedure LessRedundantTranslation in advance of the straightforward translation. This procedure replaces every pair of literals that occurs multiple times in clauses, by a single dummy variable and adds three clauses for each pair to make the dummy logically equivalent to the pair.

Algorithm 3.3 LessRedundantTranslation($\mathcal{F}$)

```
(l_i, l_j) := FindMostOccuringLiteralPair(F)
while (l_i, l_j) occurs multiple times in F do
   replace all occurrences of (l_i, l_j) by dummie d_k
   add clauses to F so d_k is equivalent to l_i ∨ l_j
   (l_i, l_j) := FindMostOccuringLiteralPair(F)
end while
return F
```

Let us illustrate this procedure by an example. Consider the following formula:

$$\mathcal{F}_{\text{example}}: \quad \begin{array}{r} x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \vee \neg x_5 \\ x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \vee x_6 \\ \neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \vee \neg x_6 \\ \neg x_1 \vee \neg x_2 \vee x_4 \vee x_5 \vee x_6 \end{array}$$

The FindMostOccuringLiteralPair procedure returns the literal pair $\neg x_2 \vee x_4$ in the first iteration, since this is the only pair that occurs four times in $\mathcal{F}_{\text{example}}$. Their occurrences are replaced by the new dummy variable d_1. Also, three clauses are added to make d_1 logically equivalent to $\neg x_2 \vee x_4$. After this first iteration $\mathcal{F}_{\text{example}}$ is equal to:

$$\mathcal{F}_{\text{example}}: \quad \begin{array}{r} x_1 \vee d_1 \vee \neg x_3 \vee \neg x_5 \\ x_1 \vee d_1 \vee \neg x_3 \vee x_6 \\ \neg x_1 \vee d_1 \vee \neg x_3 \vee \neg x_6 \\ \neg x_1 \vee d_1 \vee x_5 \vee x_6 \end{array} \qquad \begin{array}{r} d_1 \vee \neg x_2 \\ d_1 \vee x_4 \\ \neg d_1 \vee x_2 \vee \neg x_4 \end{array}$$

In this formula, pair $d_1 \vee \neg x_3$ occurs most frequently. This pair is replaced by the new dummy variable d_2 in the second iteration. To ensure the formula remains logically equivalent to the original $\mathcal{F}_{\text{example}}$, the clauses $d_2 \vee d_1$, $d_2 \vee \neg x_3$ and $\neg d_2 \vee \neg d_1 \vee x_3$ are added:

$$\mathcal{F}_{\text{example}}: \quad \begin{array}{r} x_1 \vee d_2 \vee x_6 \\ \neg x_1 \vee d_2 \vee \neg x_6 \\ x_1 \vee d_2 \vee \neg x_5 \\ \neg x_1 \vee d_1 \vee x_5 \vee x_6 \end{array} \qquad \begin{array}{r} d_1 \vee \neg x_2 \\ d_1 \vee x_4 \\ \neg d_1 \vee x_2 \vee \neg x_4 \end{array} \qquad \begin{array}{r} d_2 \vee d_1 \\ d_2 \vee \neg x_3 \\ \neg d_2 \vee \neg d_1 \vee x_3 \end{array}$$

Only one clause of length > 3 remains after this replacement. Since this clause does not contain any literal pair that occurs elsewhere in the formula, the LessRedundantTranslation procedure is finished.

3.3.2 Pyramid Translation

The second phase of the 3-Sat translator is just a straightforward translation. After some experiments, it appeared that a pyramid-shaped translation resulted in the best performance. It should be stated that the various tested translations did not have a significant influence on the final performance. Let's explain this procedure with an example. Consider the following clause of length eight:

$$x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5 \vee x_6 \vee x_7 \vee x_8$$

This clause can be translated to the clauses below using four dummy variables:

$$\begin{array}{l} x_1 \vee x_2 \vee \neg d_1 \\ x_3 \vee x_4 \vee \neg d_2 \\ x_5 \vee x_6 \vee \neg d_3 \\ x_7 \vee x_8 \vee \neg d_4 \end{array} \qquad d_1 \vee d_2 \vee d_3 \vee d_4$$

In general, clauses of length n could be reduced to clauses of length $\frac{n}{2}$ by using $\frac{n}{2}$ dummy variables. If any reduced clauses are still longer than three, the translation repeats itself until all clauses have length two or three. We applied an alternative translation to a four literal clause, since it could be translated using only a single dummy variable. The reduced clause above, containing the four dummy variables, is translated into the following two ternary clauses:

$$\begin{array}{c} d_1 \vee d_2 \vee \neg d_5 \\ d_5 \vee d_3 \vee d_4 \end{array}$$

3.4 Addition of resolvents

A resolvent is a clause that could be added to the formula by applying resolution on a group of clauses. The addition of small resolvents in the pre-processing phase may increase the performance of a satisfiability solver. Since march uses a 3-SAT translator, the formula after this translation consists solely of binary and ternary clauses. Within this environment, only four patterns arise to add small resolvents by using two clauses. These patterns, $\mathcal{F}_{\text{I}-\text{IV}}$ are listed below. We refer to a resolvent of type I-VI as a resolvent that could be added to the formula by applying pattern $\mathcal{F}_{\text{I}-\text{IV}}$. Notice that new resolvents could be made with added resolvents .

$$\mathcal{F}_{\text{I}}: \quad \left.\begin{array}{l} x_i \vee x_h \\ \neg x_h \vee x_j \end{array}\right\} \quad x_i \vee x_j$$

$$\mathcal{F}_{\text{II}}: \quad \left.\begin{array}{l} x_i \vee x_h \\ x_i \vee \neg x_h \vee x_j \end{array}\right\} \quad x_i \vee x_j$$

$$\mathcal{F}_{\text{III}}: \quad \left.\begin{array}{l} x_i \vee x_h \\ \neg x_h \vee x_j \vee x_k \end{array}\right\} \quad x_i \vee x_j \vee x_k$$

$$\mathcal{F}_{\text{IV}}: \quad \left.\begin{array}{l} x_i \vee x_h \vee x_k \\ x_i \vee \neg x_h \vee x_j \end{array}\right\} \quad x_i \vee x_j \vee x_k$$

Adding resolvents has various advantages an disadvantages. A mayor disadvantage is the increased cost of almost all procedures. When all possible resolvents of all types are added, even if only during pre-processing, solving the resulting formula might be far more costly. This is mainly caused by the large number of resolvents of type I and III. The total amount of these resolvents is - on many benchmarks - much larger than the original number of clauses.

There are also numerous advantages of added resolvents during pre-processing. One is the increased detection of *necessary assignments*, a procedure that is discussed in paragraph 2.4.1. Resolvents of all four types may contribute to this detection increase. However, the speed-up this procedure realises, is in general much smaller than the increased costs of the other procedures as a result of adding all possible resolvents.

The most important resolvents, judged on their ability to accelerate march, increase the number of unary clauses that are created during a lookahead on the complement of all but one of its literals. Newly created unary clauses may namely result in a conflict. Handling this criterium labels resolvents added by applying pattern $\mathcal{F}_I$ as less important: Whether resolvents are added or not, both $\text{IUP}(\mathcal{F} \cup \{\neg x_i\})$ and $\text{IUP}(\mathcal{F} \cup \{\neg x_j\})$ result in the same number of unary clauses. The reader may verify that added resolvents by applying pattern $\mathcal{F}_{\text{III}}$ too, do not contribute to an increased number of unary clauses during the IUP. On the other hand, the addition of resolvents by applying both other patterns does increase the creation of unary clauses: IUP($\mathcal{F}_{\text{II}} \cup \{\neg x_j\}$) and IUP($\mathcal{F}_{\text{IV}} \cup \{\neg x_j, \neg x_k\}$) result both in unary clause x_i, due to the addition of the resolvent.

Based on the criteria "increase creation of unary clauses", only resolvents of type II and IV are important. However, the addition of resolvents of type I and III during the pre-processing phase is still useful: It may be not be very profitable during execution, but these resolvents can be used in the pre-processor to create more resolvents of type II and IV. After the pre-processing phase, all the resolvents created by applying patterns $\mathcal{F}_{\text{I}}$ and $\mathcal{F}_{\text{III}}$ should than be removed.

We tested this idea, but not as easily as described above. Instead, we designed a more complex procedure that requires considerable less computational costs. In the first stage of pre-processing we only add all type IV resolvents. At the end of the pre-processing we perform an iterative full lookahead. After this procedure, the formula contains all resolvents that would have been added as if all resolvents of type I, II and III were added during pre-processing, and all added resolvents of type I and III were removed afterwards. Details about the addition of resolvents during lookahead can be found in section 4.2 "Addition of binary resolvents".

There is only one disadvantage of this cheaper procedure: Resolvents of type III could be used to push the addition of resolvents of type IV. However, the full iterative lookahead procedure is not able to detect these resolvents, and therefore they are not added to the formula. Since no alternative procedure to cheaply add these resolvents is known, only the small amount of type IV resolvents that could be created by applying pattern $\mathcal{F}_{\text{IV}}$ on the original ternary clauses are added. Concluding, after the pre-processing phase most resolvents of type II are added, as well as those resolvents of type VI that could be added using the original set of clauses.

Chapter 4

Heuristics

The application of the lookahead procedure and addition of binary resolvents appear to reduce the computational time to solve a range of benchmarks. However, their application also results in a substantial increment of the computational time on many others. The partial application of both procedures by the heuristics provided in this chapter results in a speed-up on practically all benchmarks. It also substantially increases the range of instances in which the gains achieved by the lookahead procedure and addition of binary resolvents are larger than the computational cost of both their executions and the calculation of the heuristics.

As we have seen in chapter 1, the two most important features of the lookahead procedure are the selection of an effective branching variable and the detection of failed literals. By performing only a lookahead on a subset of the set of free variables, the so-called PartialLookahead, both features more or less remain: If the selected subset would contain both the most effective branching variable and all failed literals, there will not even be any difference, while the computation cost would be heavily reduced. However, that subset would be difficult to obtain without actually performing lookahead, but restricting the lookahead procedure only on the subset obtained by the heuristics presented here, results on average in a substantial performance gain.

Partial addition of binary resolvents is far more important than the partial lookahead, since a full addition seldom results in a speed-up. The advantages of adding binary resolvents are numerous, but every added resolvent also results in a minor slowdown. Therefore, only those resolvents that contribute to the most important advantages of the addition are added. These advantages involve a higher quality of the pre-selection heuristics and an increased detection of failed literals.

4.1 Partial lookahead

The partial execution of the lookahead procedure in march involves only a lookahead on variables in $\mathcal{P}$: the set containing the lookahead variables (see algorithm 2.1 for details). The outline of this section is as follows: First, we explain why we restrict $\mathcal{P}$ to a fixed size. Second, paragraph 4.1.2 describes the heuristics that determine which variables will be added to $\mathcal{P}$. Finally, this section concludes with experiments and decisions about which variable in $\mathcal{P}$ should be selected for branching in order to obtain optimal performance.

4.1.1 Constant percentage

March selects a fixed number of variables for examination during lookahead. The fixed number is based on a constant percentage of the original number of variables. We refer to this pre-selection procedure as $presel_c$. When a concrete percentage is used, say 10%, this is denoted as $presel_{10}$.

As far as we know, march is the only lookahead-oriented solver that examines a fixed number of variables during lookahead. Usually, either a full lookahead is performed, or pre-selection heuristics are applied in such a way, that a vast number of variables enter the lookahead procedure near the root of the search-tree, while only a few variables are selected near the leaves. The argument behind these heuristics is that higher nodes are small in number and have a major impact on the size of the search-tree. A broad lookahead at these nodes is therefore useful and relatively cheap. On the other hand, the nodes near the leaves are large in number and they have only a small effect on the search-tree size. Selecting a small amount of the free variables to enter the lookahead phase at these nodes will speed up solving. An example of this kind of heuristics is $prop_z$ by Li [17]. In short, these heuristics work as follows: Every variable that satisfies certain constraints on its occurrences in binary clauses will enter the lookahead procedure. If the amount of variables that satisfies these constraints is below a certain threshold, a full lookahead is performed.

An important property of $presel_c$ is the fixed number of variables it selects for examination during the lookahead phase. If the restriction of variables is strictly based on their occurrences in binary clauses and on a threshold, like $prop_z$, non-directed effects may be the result: These pre-selection heuristics may cause a very high or very low percentage at any position in the tree: Permanent full lookahead might occur when all variables satisfy the constraints, or when the threshold is never crossed. On the other hand, if the threshold is crossed with a small margin, the number of lookahead variables will be very low.

$Presel_c$ does not yield such side-effects. This is no guarantee for a better performance. However, it is easier to optimise pre-selection heuristics based on a fixed number of variables for global purposes, since only one parameter is

search-tree depth	number of free variables	$presel_{10}$	$prop_z$
1	298.24	30	298.24
2	296.52	30	296.52
3	294.92	30	293.89
4	292.44	30	292.21
5	288.60	30	280.04
6	285.36	30	252.14
7	281.68	30	192.82
8	277.54	30	125.13
9	273.17	30	71.51
10	268.76	30	40.65
11	264.55	30	26.81
12	260.53	30	21.55
13	256.79	30	19.80
14	253.28	30	19.24
15	249.96	30	19.16
16	246.77	30	19.28
17	243.68	30	19.57
18	240.68	30	19.97
19	237.73	30	20.46
20	234.82	30	20.97

Table 4.1 — Average number of variables selected for the lookahead procedure by $presel_{10}$ and $prop_z$ for 300 variables and 1275 clauses random 3-SAT problems.

involved. Table 4.1 shows a comparison of the number of variables returned by pre-selection heuristics $presel_{10}$ and $prop_z$.

4.1.2 Pre-selection heuristics

To be able to select only a small, fixed number of variables for the lookahead procedure, a ranking that offers a clear discrimination between the variables is necessary. Therefore, pre-selection heuristics could not merely be based on binary clauses, because this generally results in a weak discrimination. This paragraph deals with an approximation algorithm of the lookahead evaluation, which will be used to pre-select the variables.

Recall that an evaluation procedure of the lookahead that counts newly created binary clauses in the resulted formula, results in fast performances on random 3-SAT and many other benchmarks. We refer to this evaluation procedure as DIFF$_1$. Let $occ_3(x)$ be the number of occurrences of literal x in ternary clauses and $\mathcal{A}(x)$ the set of unary clauses that are generated during IUP$(\mathcal{F} \cup \{x\})$. When $\mathcal{A}(x)$ is known, DIFF$_1(x)$ could be precicely calucalted using equation (4.1). However, $\mathcal{A}(x)$ is only available through a lookahead on x. Therefore, DIFF$_1(x)$ can not be exactly calculated before actually performing the lookahead procedure.

An effective and clear discrimination between variables is found in heuristics based on an approximation function of $\textsc{Diff}_1$: Since march adds a significant part of the possible binary resolvents to the formula during the solving phase, all variables y_i, that occur in the binary clauses of the format $\neg x \vee y_i$, represent a considerable part of the variables in $\mathcal{A}(x)$. A $\textsc{Diff}_1(x)$ approximation function, $\textsc{Daf}(x)$, using these y_i variables is presented in equation (4.2). The quality of the $\textsc{Daf}$ depends predominantly on the quality of the representation of $\mathcal{A}(x)$ by y_i variables. This dependency is properly taken into consideration in the kind of binary resolvents that are added during the solving phase (see section 4.2)

$$\begin{aligned}
\textsc{Diff}_1(x) &= occ_3(\neg x) + \sum_{y_i \epsilon \mathcal{A}(x)} occ_3(\neg y_i) && (4.1)\\
\textsc{Daf}(x) &= occ_3(\neg x) + \sum_{\neg x \vee y_i \epsilon \mathcal{F}} occ_3(\neg y_i) && (4.2)\\
\textsc{Rank}(x) &= \textsc{Daf}(x) \times \textsc{Daf}(\neg x) && (4.3)
\end{aligned}$$

Finally, the $\textsc{Rank}$ of variable x is calculated by multiplying $\textsc{Daf}(x)$ and $\textsc{Daf}(\neg x)$. We refer to the "best" n variables in a certain node as those n variables that have the highest $\textsc{Rank}$. The number of variables to be selected to enter the lookahead - in order to realise optimal performance - is benchmark-dependent. However, relatively optimal results are achieved when performing only a lookahead on the "best" 10% variables. This percentage was motivated by the considerations described in paragraph 2.2.2.

4.1.3 Final branch decision

For the actual evaluation of the lookahead, we do not use $\textsc{Diff}_1$ as defined above, but a variant proposed by Dubois and Dequen [9]. Their heuristics, $\textsc{Diff}_2$, weighs new binary clauses, instead of weighing them equal. Let $\mathcal{I}(x)$ be the set of binary clauses derived form ternary clauses during the IUP($\mathcal{F} \cup \{x\}$). Both $\textsc{Diff}_{i=1,2}$ can be calculated using this set:

$$\begin{aligned}
\textsc{Diff}_1(x) &= |\mathcal{I}(x)| && (4.4)\\
w(x) &= \alpha \times occ_2(\neg x) + occ_3(\neg x) && (4.5)\\
\textsc{Diff}_2(x) &= \sum_{(a \vee b) \epsilon \mathcal{I}(x)} w(a) \times w(b) && (4.6)
\end{aligned}$$

The α in the weight-equation is a magic constant. Its value is not clear. In their paper, Dubois and Dequen [9] gave it value 2, but they use value 5 in their solver kcnfs. Since march adds many binary resolvents during execution - which increases the $occ_2(x)$ values - a lower value of α is expected.

α is optimaized on random 3-SAT benchmarks with a density of the clause-variable ratio near the threshold. The frequent occurrences of ternary clauses on these benchmarks on all the nodes of the search-tree, provides an ideal environment to optimise this parameter. Experimental results showed an optimal value of 4.

Due the great differences between benchmarks, it is expected that the optimal value of α differs over the variety of benchmark families. However, it appears that while using DIFF_2 with $\alpha = 4$, performances rise over the complete benchmark spectrum in comparison with DIFF_1. This provides solid ground for using DIFF_2 for general purposes outside of the random benchmark domain.

The performance increment that results form the application of DIFF_2 on non-random instances can partly be explained by a better adaptation to the 3-SAT translator. Industrial benchmarks for instance, have some large clauses. The reduction of a large clause is generally less important than the reduction of a small clause. The 3-SAT translation divides large clauses into ternary clauses containing dummy variables. In DIFF_1 all reductions to binary clauses have the same importance. However, since the dummies occur relatively sporadically in the formula, their weight is small. Therefore, DIFF_2 will rate the reduction of ternary clauses resulting from a larger one lower than the reduction of an original ternary clause.

To reduce computational costs to obtain DIFF_2, the weight-equation of every literal is computed on the chosen path and stored in a special array. Because these calculations are only performed once per node, the optimization experiments of these weights are quite accurate: Complex calculations are not necessarily excluded, because their computational costs may automatically tackle the possible performance gain they could achieve.

$$w^*(x) = \Big(\alpha \times occ_2(\neg x) + occ_3(\neg x)\Big)^{\beta} \tag{4.7}$$

$$\text{DIFF}_2^*(x) = \sum_{(a \vee b) \epsilon \mathcal{I}(x)} w^*(a) \times w^*(b) \tag{4.8}$$

Experiments were performed to obtain the optimal parameter settings of a more complex weigth-function (equation (4.7)). In this equation, a new parameter β is introduced. This parameter expresses the importance of the weights: $\beta = 0.0$ results in DIFF_1, while $\beta = 1.0$ results in DIFF_2. Experiments on random 3-SAT near the threshold, presented in Figure 4.1, show that $\beta = 1.0$ is relatively optimal. Therefore, we chose to use the original DIFF_2 as proposed by Dubois and Dequen [9], but with $\alpha = 4$. The final selection of the branch variable is based on the variable with the highest $H(x)$. Equation (4.9) is proposed by Freeman [10]. The 1024 in this equation forces the addition of $\text{DIFF}_2(x)$ and $\text{DIFF}_2(\neg x)$ to be only considered in case of a tie break.

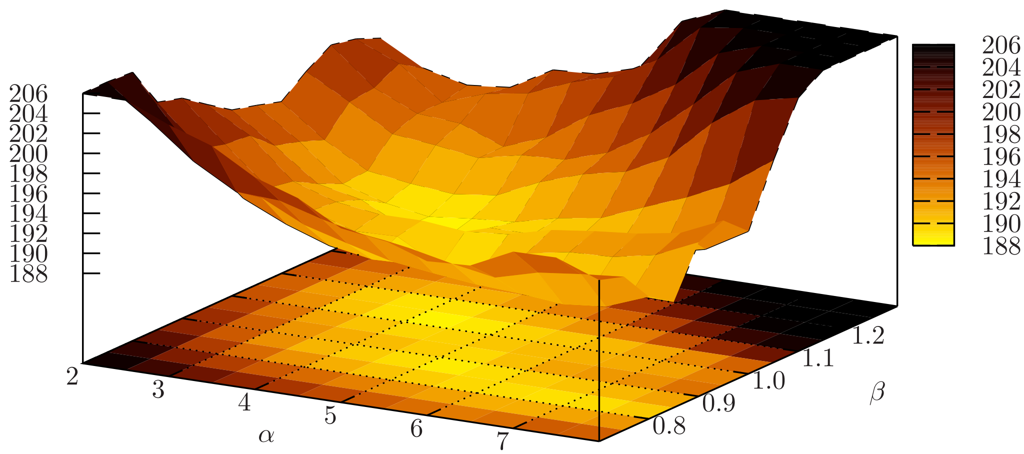


Figure 4.1 — Cumulative performances achieved by march on various settings of α and β on 50 unsatisfiable random 3-SAT formulas with 300 variables and 1275 clauses.

$$H(x) = \textsc{Diff}_2(x) \times \textsc{Diff}_2(\neg x) \times 1024 + \textsc{Diff}_2(x) + \textsc{Diff}_2(\neg x) \tag{4.9}$$

4.2 Addition of binary resolvents

During the solving phase it is also possible to add resolvents. Since only binary resolvents could be easily detected during this phase only those are considered here. The advantages of the addition of binary resolvents are numerous: It improves the detection of failed literals; the occurrences of necessary assignments (paragraph 2.4.1); the quality of pre-selection heuristics (paragraph 4.1.2) and a further reduction of the lookahead costs by a tree-base lookahead (section 4.3). Therefore, it seems sensible to add all possible binary resolvents. This was also performed in early versions of march. However, there is also a considerable disadvantage: Practically all procedures slow down a bit as a consequence of each added binary resolvent. Since the number of possible binary resolvents that could be added during execution is enormous, full addition results often in an overall loss of speed. To maintain speed, partial addition of resolvents is essential. This section includes a cost-benefit analysis to provide some insight in the number of resolvents to be added for efficient solving. Along with some indicators, a careful decision is presented on the question which resolvents should be added and why.

4.2.1 Pre-selection paradox

This paragraph deals with the interdependence between our pre-selection heuristics and the addition of binary resolvents. This produces both favourable and unfavourable effects: On one hand, the partial lookahead leads to a partial addition of resolvents, because resolvents are only partially recognised. Fortunately these are the important resolvents. On the other hand, this addition can result in 'auto-suggestion' on the importance of the variables.

Partial recognition

To increase the quality of the pre-selection heuristics, a reasonable amount of resolvents should be added. The lookahead phase is ideal for this addition, because here, resolvents are easily and cheaply recognized: During the lookahead on a variable, resolvents of length two could be formed by merging the complement of the lookahead-literal with the unary clauses generated during IUP($\mathcal{F} \cup \{lookaheadliteral\}$). For example: Lookahead on $\neg x$ may result in unary clauses y_i. All resolvents of the format $x \vee y_i$ could be added to the formula, assuming that they do not already exist.

However, not all resolvents are found during the lookahead phase. Those that are found are the result of the merger of a lookahead variable with an unary clause caused by its IUP. Therefore, at least one literal of these resolvents is a lookahead variable. Because march performs a partial lookahead, only a subset of all possible resolvents is found: The ones that contain a lookahead variable. This results in some mixed effects:

Addition of all possible resolvents is very time consuming. Automatic selection as a result of the partial lookahead, appears to be a favourable side-effect: Those variables labelled by the pre-selection heuristics as "important" will occur at least once in every resolvent. Resolvents of two unimportant variables will not be recognized, and so they will not be added.

However, there is also an unfavourable side-effect: Pre-selection heuristics depend heavily on the original binary clauses and added binary resolvents. Recall that variables with a high pre-selection Rank are selected for the lookahead phase. The resolvents added during lookahead have a majority of lookahead literals, raising their pre-selection Rank even higher. On the other hand, variables with a low pre-selection Rank will only sporadically occur in the added resolvents, which keeps their Rank low.

Chosen path compensation

The IterativeUnitPropagation of the selected branching variable, results in clause reductions and in the creation of new binary clauses. With these new clauses, binary resolvents could be created using existing binary clauses: Consider the new binary clause $x_i \vee x_j$. For all binary clauses $\neg x_i \vee y$ or $\neg x_j \vee y$

that occur in the formula, the clauses $x_j \vee y$ or $x_i \vee y$ could be added.

March adds these resolvents because of the following reason: The resolvents that were added during lookahead were biased by auto-suggestion: Variables with a high pre-selection Rank will occur relatively more frequent in resolvents. The addition of a group of resolvents with original binary clauses appears more in agreement with the formula. This will decrease the bias by auto-suggestion and it enhances the quality of the pre-selection heuristics.

Another reason for the addition of binary resolvents on the chosen path involves the detection of forced literals. If a clause $x \vee y$ exists and the resolvent $\neg x \vee y$ can be added, y is forced. This will result in a reduction of the size of the formula, which again increases the quality of the pre-selection heuristics.

Conclusion

Addition of binary resolvents is essential when using the pre-selection heuristics defined in paragraph 4.1.2, because of their strong interdependency. Their addition during the lookahead phase is recommended. Here it is relativity cheap to do so, and the important binary results are added, even though this has a seemingly unfavourable effect on the quality of the lookahead. To compensate, the addition of non-biased resolvents on the chosen path has a noticeable positive effect on the performance and may enhance the quality of the pre-selection heuristics. Looking at the arguments and some experimental results, the addition of non-biased resolvents on the chosen path is an important aspect of these heuristics.

Of course, the pre-selection heuristics are not perfect. While performing a partial lookahead, it is already considered an accomplishment when the variable with the highest Diff is among a small group of pre-selected ones. If ever the pre-selection heuristics themselves were sufficient to determine the most important variable for branching, a lookahead procedure would become obsolete.

4.2.2 Constraint resolvents

An impression of the usefulness of a resolvent could clarify which ones contribute to a speed-up. As mentioned in the introduction of this paragraph, a binary resolvent could be added for every unary clause that is created during the propagation of a lookahead literal - provided that the binary clause does not already exist. A special group of resolvents could be created by merging a unary clause that originates from a ternary clauses prior to the lookahead. In this case we speak of *constraint resolvents*.

Constraint resolvents have the property that they could not be found by a lookahead on the complement of the unary clause. Adding these constraint resolvents results in a more vigorous detection of failed literals. This is probably best explained with an example:

$$\left.\begin{array}{rc} \left.\begin{array}{r} r \vee \neg s \\ s \vee \neg t \\ s \vee t \vee x \end{array}\right\} & r \vee x \\ \\ \left.\begin{array}{r} u \vee \neg v \\ u \vee \neg w \\ v \vee w \vee x \end{array}\right\} & u \vee x \\ \\ \neg r \vee \neg u \vee x & \end{array}\right\} \quad x$$

(a) (b) (c)

Figure 4.2 — Detection of a failed literal by adding constraint resolvents. (a) The original clauses, (b) constraint resolvents and (c) a forced literal.

First, consider only the original clauses of an example formula (Figure 4.2 (a)). A lookahead on $\neg r$, $\text{IUP}(\mathcal{F} \cup \{\neg r\})$, results in unary clause x. Therefore one could add resolvent $r \vee x$ to the formula. Since unary clause x was originally a ternary clause (before the lookahead on $\neg r$), this is a constraint resolvent. The unique property of constraints resolvents is that when they are added to the formula, lookahead on the complement of the unary clause results in the complement of the lookahead-literal. Without this addition this would not be the case. Applying this to the example: Addition of $r \vee x$ to the formula $\text{IUP}(\mathcal{F} \cup \{\neg x\})$ will result in unary clause r, while without this addition it will not.

$\text{IUP}(\mathcal{F} \cup \{\neg r\})$ also results in unary clause $\neg t$. Therefore, resolvent $r \vee \neg t$ could be added to the formula. Since unary clause $\neg t$ was originally a binary clause, $r \vee \neg t$ is not a constraint resolvent, because $\text{IUP}(\mathcal{F} \cup \{t\})$ would already result in unary clause r.

Constraint resolvent $u \vee x$ is detected during $\text{IUP}(\mathcal{F} \cup \{\neg u\})$. After the addition of these constraint resolvents (Figure 4.2 (b)), the lookahead $\text{IUP}(\mathcal{F} \cup \{\neg x\})$ results in a conflict, making $\neg x$ a failed literal and thus forces x. It is important to notice that $\text{IUP}(\mathcal{F} \cup \{\neg x\})$ will not result in a conflict if constraint resolvents $r \vee x$ and $u \vee x$ were not added both. Concluding, by partial addition of resolvents, constraint resolvents are preferred.

4.2.3 Conclusion

March adds all identified constraint resolvent during the lookahead phase, as well as all chosen path compensation resolvents. The decision is based on the following considerations:

- Do not add all resolvents. This is very time consuming and it will hamper performance.
- Add resolvents to increase the number of detected necessary assignments, the quality of the pre-selection heuristics and the reduction achieved by tree-based lookahead.
- Add constraint resolvents. These increase the detection of failed literals.
- Add a reasonable number of resolvents on the chosen path to reduce auto-suggestive pre-selection heuristics.

4.3 Tree-based lookahead

4.3.1 The concept

The lookahead procedure is mainly expensive due to the enormous number of propagations (fixing of a variable and the consequential simplification of the formula) that are the result of the lookahead on all variables that are selected by the pre-selection heuristics. Tree-based lookahead aims to reduce the number of propagations by combining multiple lookaheads under a certain common condition, where one lookahead assignment implies an other.

To illustrate this, consider the following situation: Two variables x_i and x_j occur frequently in a formula, both positive and negative and in both binary and ternary clauses. Variables x_i and x_j are selected by the pre-selection heuristics for the lookhead. Binary clause $x_i \vee x_j$ exists.

Since x_i and x_j are both lookahead variables, $\text{IUP}(\mathcal{F} \cup \{x_i\})$, $\text{IUP}(\mathcal{F} \cup \{\neg x_i\})$, $\text{IUP}(\mathcal{F} \cup \{x_j\})$ and $\text{IUP}(\mathcal{F} \cup \{\neg x_j\})$ are called by the lookahead procedure. Due to the frequent occurrences x_i and x_j in the formula, all four of them will be costly operations. However, the required computational costs can potentially be largely reduced: First perform lookahead on x_i. If the resulted formula of $\text{IUP}(\mathcal{F} \cup \{x_i\})$, denoted as $\mathcal{F}_{x_i}$, does not contain the empty clause, then $\textsc{Diff}(\mathcal{F}, \mathcal{F}_{x_i})$ is calculated. The formula is *not* restored to perform the next lookahead: Since binary clause $x_i \vee x_j$ exists, $\neg x_j \rightarrow x_i$. The number of propagations during the lookahead on $\neg x_j$ could be strongly reduced by calling $\text{IUP}(\mathcal{F}_{x_i} \cup \{\neg x_j\})$ instead of $\text{IUP}(\mathcal{F} \cup \{\neg x_j\})$. Likewise, the number of propagations performed during a lookahead on $\neg x_i$ could be largily reduced (due to $\neg x_i \rightarrow x_j$) by first calling $\mathcal{F}_{x_j} = \text{IUP}(\mathcal{F} \cup \{x_j\})$ and afterwards $\text{IUP}(\mathcal{F}_{x_j} \cup \{\neg x_i\})$.

The tree-based lookahead is focused on building trees using those binary clauses that contain two variables from the pre-selected set. These clauses are used to reduce the number of propagations required during the execution of the lookahead procedure.

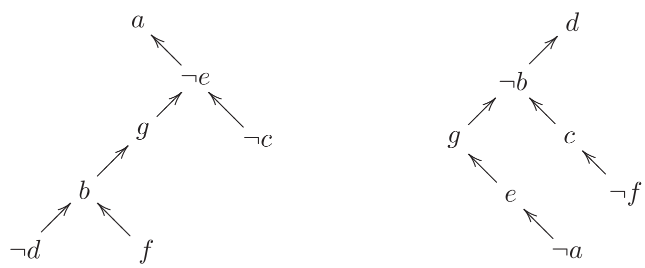


Figure 4.3 — Two example trees that can be build by $\mathcal{F}_{\text{binary}}$.

4.3.2 Building a tree

To build a tree, first all binary clauses that contain two lookahead variables are extracted from the formula. An example of this resulted sub-formula is $\mathcal{F}_{\text{binary}}$:

$$\mathcal{F}_{\text{binary}} = (a \vee e) \wedge (a \vee \neg g) \wedge (\neg b \vee \neg c) \wedge (b \vee \neg d) \wedge (b \vee \neg f) \wedge (\neg b \vee g) \wedge (c \vee \neg e) \wedge (c \vee f)$$

Using $\mathcal{F}_{\text{binary}}$, we start building trees. Our goal is to create trees that are as deep as possible: The deeper the tree, the stronger the reduction of propagations realised by applying the tree during the lookahead. Our current algorithm is as follows: First, all variables are sorted according to their Rank calculated by the pre-selection heuristics. It appeared that sorting the lookahead variables before the actual building phase resulted in deeper trees. Afterwards, we loop though the sorted list of lookahead literals l_i and examine all l_j those that occur in clauses $\neg l_j \vee l_i$ in $\mathcal{F}_{\text{binary}}$: For each l_j, we check whether l_j already occurs in a tree. If this is not the case or l_j is the root of a tree, we create branch $l_j \rightarrow l_i$. In case l_j is already somewhere in a tree - say branch $l_j \rightarrow l_k$ exists - we compare the depth of l_i and l_k. If l_i lies deeper in a tree than l_k, branch $l_j \rightarrow l_k$ is removed and branch $l_j \rightarrow l_i$ is added. Nota bene, l_i could be in the sub-tree of l_j. In this case the above transportation is not allowed.

Figure 4.3 presents two trees that cover all lookahead literals in$\mathcal{F}_{\text{binary}}$. The reduction that could be achieved by applying these trees during the lookahead is considerable: For instance, without the trees the propagation of literal a and d would have been called seven times, while by traversing the trees, both propagations will only be called once.

4.3.3 Tree-based lookahead procedure

Once trees that cover all lookahead literals are constructed, applying them during the lookahead is requires not much effort. Because march uses a timestamp data-structure (see section 5.2), the application of trees during the lookahead is explained using this data-structure. In other data-structures its application is often just as easy.

In the timestamp data-structure, one only requires a conversion algorithm that translates the trees in (1) an ordered literal array that describes the sequence in which the lookahead on literals is performed and (2) an array containing the offset in relation to the lowest timestamp used in the current lookahead procedure. We refer to this timestamp as the *thisNodeMinimumStamp*.

Both arrays can be economically created by executing algorithm 4.1 CreateTreeArrays. The result of this algorithm on the left tree in figure 4.3 is offered in figure 4.4.

Algorithm 4.1 CreateTreeArrays( )

```
global currentPosition = 0
global currentOffset = 0
for all lookahead trees t_i do
  TreeArrayRec(root of t_i)
end for
```

Algorithm 4.2 TreeArrayRec(l_i)

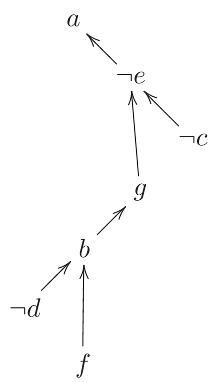
```
current := #NodesInThisBranch - 1 + currentOffset
LiteralArray[ currentPosition ] := l_i
OffsetArray[ currentPosition ] := current
currentPosition++

for all childs l_j of l_i do
  TreeArrayRec(l_j)
end for

if current = currentOffset then
  currentOffset++
end if
```

When both arrays are created, the highest level of the lookahead procedure does not require many changes: An elementary lookahead procedure is presented in algorithm 4.3, while algorithm 4.4 deals with the TreeBasedLookahead procedure.

Using TreeBasedLookahead, considerable gains were noticed on benchmarks with many binary clauses. Some even gained a factor of 10 in speed, solely due to the use of trees. On random 3-Sat benchmarks near the threshold with 250 variables, only a 10% gain was observed. For higher numbers of variables the gain was larger: The speed on random 3-Sat with 400 variables



① $currentTimeStamp + 6$
② $currentTimeStamp + 5$
③ $currentTimeStamp + 0$
④ $currentTimeStamp + 4$
⑤ $currentTimeStamp + 3$
⑥ $currentTimeStamp + 1$
⑦ $currentTimeStamp + 2$

Figure 4.4 — Converting the tree in a literal and offset array

Algorithm 4.3 ElemantaryLookahead( )

repeat
  for $0 \leq i < LiteralArrayElements$ **do**
    $currentTimeStamp{+}{+}$
    LiteralLookahead(LiteralArray[i])
  end for
until noNewFailedLiterals

Algorithm 4.4 TreeBasedLookahead( )

$thisNodeMinimumStamp := currentTimeStamp$
repeat
  for $0 \leq i < LiteralArrayElements$ **do**
    $currentTimeStamp = thisNodeMinimumStamp +$ OffsetArray[i]
    LiteralLookahead(LiteralArray[i])
  end for
  $thisNodeMinimalStamp \mathrel{+}= LiteralArrayElements$
until noNewFailedLiterals

improved by 15%. The relatively small gains on random 3-Sat formulas could be explained by the absence of initial binary clauses.

Two improvements are considered for future implementations:

- Not all implications between lookahead variables are stored in the binary clauses that contain two lookahead variables. Increasing the number of these implications can possibly result in deeper trees.
- Spend more computational time trying to find more optimal trees, possibly by utilising state of the art graph-to-tree decomposition algorithms.

Chapter 5

Data-structure

March features three important data-structures that do not only directly increase performance by their efficient use of memory, but also because they create an environment in which effectively complex procedures could be performed in quite an efficient manner. All three of them were originally designed to speed up the lookahead phase.

The first data-structure consists of *implication arrays* and significantly reduces the amount of memory used by march. This lower usage of memory frequently results in a speed-up due to a better use of the cache. March especially benefits from better cache usage during lookahead because many procedures are iteratively executed. Some smaller, additional advantages consist of increased speed in which conflicts are detected during lookahead, and a more accurate calculation of the Diff.

Both other added data-structures depend heavily on each other. The first, the *TimeAssignment* structure, was developed to avoid backtracking during the lookahead phase. Yet, this feature has a price: The procedure that handles the reduction of ternary clauses became significantly slower. To ease the pain, we replaced the data-structure for ternary clauses by a *local-literal storage* structure that substantially reduces this slowdown.

The local-literal storage also yields a disadvantage: Since clauses are no longer stored as such in this data-structure, it is not possible to cheaply verify whether they are already satisfied. The last enhancement described in this chapter, removes all satisfied clauses from the active data-structure used during the lookahead phase. This enhancement is not a data-structure, but a procedure that is required to achieve significant speed-ups when applying both the TimeAssignment and the local-literal storage data-structures.

5.1 Implication arrays

In march, binary clauses are not stored as such, but as two implications: A binary clause $x \vee y$ is equivalent to the implications $\neg x \rightarrow y$ and $\neg y \rightarrow x$. Instead of storing the clauses in a clause-database and add variable arrays that point towards it, binary clauses are stored in *implication arrays.* For binary clause $x \vee y$, this means that y is stored in the implication array of $\neg x$ and that x is stored in the implication array of $\neg y$. Both the original binary clauses, as well as those generated during execution, are stored in these arrays. This separate storage results in a lower use of memory and makes it possible to implement a variant of IterativeUnitPropagation (IUP), which has several favourable effects during the lookahead phase.

5.1.1 Reduction of memory cost

One of the advantages of storing the binary clauses in implication arrays is that it requires half as much space as conventional storage methods: Within the implication arrays, only two memory entries are used to store a binary clause. In a structure with pointers to clauses, two pointers are used to point to the clause and two memory entries are used to store the actual clause, adding up to a total of four entries.

$clause_0$	a	b
$clause_1$	$\neg b$	c
$clause_2$	$\neg a$	$\neg c$
$clause_3$	a	c

a	0	3
$\neg a$	2	
b	0	
$\neg b$	1	
c	1	3
$\neg c$	2	

(a)

a	$\neg c$	
$\neg a$	b	c
b	c	
$\neg b$	a	
c	$\neg a$	
$\neg c$	$\neg b$	a

(b)

Figure 5.1 — (a) A structure with binary clauses stored in a clause database, and variable arrays containing the indices that point to the clauses in which they occur. (b) Binary clauses stored in implication arrays.

Since march adds a lot of binary resolvents during the solving phase, the binary clauses on average outnumber the ternary clauses. Therefore, storing these binary clauses in implication arrays significantly reduces the total amount of march's memory use.

5.1.2 IFIUP

During the lookahead phase, we perform a variant of IteravtiveUnitPropagation (IUP) that propagates implications before it shortens ternary clauses:

ImplicationsFirstIUP (IFIUP), see algorithm 5.1. This variant is easy to implement since the binary clauses / implications are separately stored from the ternary clauses in the implication arrays. IFIUP starts by trying to fix as many variables as possible, using only the binary part of the formula. This task is performed by the procedure RecursiveImplicationPropagation (RIP).

Algorithm 5.1 ImplicationsFirstIUP($\mathcal{F} \cup \{l_i\}$)

```
Reset(fixstack)

result := RecursiveImplicationPropagation(l_i)
if result = "conflict" then return "conflict"

while fixstack ≠ ∅ do
  l_j := Pop( fixstack )
  result := TernaryClausePropagation(l_j)
  if result = "conflict" then return "conflict"
end while

return "continue"
```

Algorithm 5.2 RecursiveImplicationPropagation(l_i)

```
Fix(l_i)
Push(fixstack, x)

for l_j ∈ ImplicationArray[ l_i ] do
  if l_j is not fixed then
    result := RecursiveImplicationPropagation(l_j)
    if result = "conflict" then return "conflict"
  else if l_j is fixed on opposite truth value then
    return "conflict"
  end if
end for

return "continue"
```

When all variables are fixed using only the implications, the simplification of the formula is performed: For every variable x_i that was fixed during RIP, TernaryClauseReduction (TCR) is called. This procedure shortens all ternary clauses in which $\neg x_i$ occurs. If a ternary clause is reduced to a unary clause, RIP is called to fix all those variables that could be propagated due to this unary clause.

Besides the call of RIP, a *constraint resolvent* is added for every unary

Algorithm 5.3 TernaryClauseReduction(l_i)

```
for clause_k ∈ TernaryClauseArray[ ¬l_i ] do
  ClsLength[ clause_k ] := ClsLength[ clause_k ] - 1

  if ClsLength[ clause_k ] = 1 then
    l_j := GetRemainingLiteral(clause_k)
    AddConstraintResolvent(¬lookaheadliteral ∨ l_j)

    result := RecursiveImplicationPropagation(l_j)
    if result = "conflict" then return "conflict"
  end if
end for

return "continue"
```

clause detected in the TCP procedure. Apart from the advantages described in paragraph 4.2.2, these resolvents increase the number of variables that will be fixed during the first call of RIP, next time IFIUP is performed on the same lookahead-literal.

The advantage of using IFIUP instead of a general IUP is that it will find an existing conflict more quickly. Due to the addition of constraint resolvents, most conflicts will be detected in the first call of RIP. In such a case, the procedure TCP will never be executed. Since RIP is a faster procedure than TCP, a faster detection of the conflict is expected.

The most important argument in favor of performing IFIUP during the lookahead phase is described in paragraph 5.2.3. The additional information provided in this paragraph is necessary to understand why this variant is preferred.

5.2 TimeAssignments

March uses a data-structure for the lookahead called *TimeAssignments* (TA). This structure backtracks all propagations performed during a IterativeUnitPropagation by raising a single timestamp. To realise this ultra fast backtracking method, this structure contains two timestamps for every variable. Also, two pre-computed weights are stored for each variable; one positive and one negative. These are used for the complex Diff, which is explained in paragraph 4.1.3.

5.2.1 Timestamp data structure

During the lookahead of literal x, all variables that are fixed by IUP($\mathcal{F} \cup \{x\}$) are stamped using the *CurrentTimeStamp* (CTS): If a variable is assigned as true, it is stamped with CTS, and if it is assigned as false, it is stamped with CTS + 1. After the lookahead on a literal, the CTS is increased by 2 to unfix all assigned variables.

Algorithm 5.4 STAMP(l_i)

timeStamp[l_i] := CurrentTimeStamp
timeStamp[$\neg l_i$] := CurrentTimeStamp + 1

The actual truth value (true, false or unfixed) is not stored in our data structure. However, it could be obtained by comparing the stamp stored in the TA of a variable with the CTS.

$$TA[x] = \begin{cases} stamp < CTS & \text{unfixed} \\ stamp \geq CTS \textbf{ and } stamp \equiv 0 \pmod 2 & \text{true} \\ stamp \geq CTS \textbf{ and } stamp \equiv 1 \pmod 2 & \text{false} \end{cases}$$

Fixed variables on the chosen path are stamped with the *MaximumTimeStamp* (MTS). This is the maximum even value of an integer. One has to make sure that the CTS will never be equal to the MTS. This will rarely be the case and could easily be prevented by a check at the beginning of each lookahead. When backtracking the chosen path, all unfixed variables are stamped by 0.

Algorithm 5.5 UPDATESTAMPS( )

if LMTS + GAP > MTS **then**
  reset all stamp smaller then LMTS
  CTS := 2
  LMTS := GAP
else
  CTS := LMTS + 2
  LMTS := LMTS + GAP
end if

We refer to *forced variables* as failed literals and necessary assignments. Forced variables play an important role in a fast and effective lookahead procedure: If during the lookahead phase a forced variable is detected, the H(x_i) of all lookahead variables changes, because DIFF($\mathcal{F}_{x_i}$, $\mathcal{F}$) is altered due to the changes in both $\mathcal{F}_{x_i}$ and $\mathcal{F}$: Forced variables will satisfy some clauses and shorten others. Therefore, march iterates the lookahead phase until no new forced literals are found during the final iteration.

Forced variables could be stamped with the MTS. However, this will require a reset of the stamps of all free variables at the beginning of each lookahead phase. The amount of these reset procedures could be significantly reduced using a lower timestamp for forced variables. We refer to this stamp as the *LocalMaximumTimeStamp*(LMTS).

One could also consider a different value of LMTS for every node. A lower value of LMTS than MTS will reset all the variables, making the CTS just higher than the LMTS of the prior node. Then, the LMTS needs to increase in order to be larger than the CTS. We decided to let the LMTS have a constant difference with the initial value of the CTS. We refer to this difference as GAP.

The value of GAP should be chosen carefully in order to optimally reduce the number of reset procedures. If the GAP value is too low, CTS will be frequently equal to LMTS, as a consequence of the iterations of the lookahead procedure. in this case a reset has to be performed. If the GAP value is set too high, then for every $\frac{MTS}{GAP}$ nodes, LMTS will be larger than MTS. In this case too, a reset of the stamps is required.

Intuitively, we chose a GAP value of ten times the number of lookahead literals. This means that a reset operation - due to a low GAP value - is performed only when a lookahead procedure will iterate more then ten times, which occurs rarely.

The check whether the CTS is not larger than the MTS, combined with the use of GAP, is implemented in UpdatedStamps. This procedure is called at the beginning of each lookahead procedure.

In conclusion: We implemented one of the few optimisations still conceivable in a fast lookahead procedure. It drastically reduces the amount of reset operations in a timestamp data-structure. With the current use of LMTS, MTS and GAP, we approximated a minimum of reset operations.

5.2.2 Efficient calculation of lookahead evaluation

The downside of a timestamp data-structure during lookahead involves increased costs to implement the TernaryClauseReduction procedure. If this procedure is not implemented efficiently, the profits of the ultra fast backtrack method are tackled. A fast implementation of the TernaryClauseReduction procedure (TCR) in a common backtrack structure could restrict the instructions to a reduction of the length of clauses and a check whether the reduced length equals one. In that case, the remaining literal should be propagated.

An efficient algorithm to implement TCR in a timestamp data-structure is presented in algorithm 5.6 TimeStampTernaryClauseReduction (TSTCP).

Especially the two OtherLiteral procedures in line 2 and 3 are expensive. Great effort has been invested in the reduction of computational costs of this procedure. The result can be found in paragraph 5.3.3.

In alternative data-structures, the OtherLiteral procedures are not necessary when an elementary Diff is used (one that counts only new binary clauses). However, if the complex Diff - as described in paragraph 4.1.3 - is used, it becomes also necessary to obtain both literals in alternative data-structures, since one needs to obtain the weights to compute this complex Diff (see line 20 algorithm 5.6). When using the complex Diff, the costs to implement TCR procedures in a timestamp data-structure are comparable to implementation costs in alternative data-structures. Recall that only the timestamp structure has the advantage of ultra fast backtracking!

Notice that no check is implemented in algorithm 5.6 to verify whether a clause is satisfied. The check becomes redundant since all satisfied clauses are removed from the formula on the chosen path. See section 5.4 for details. This also yields a small disadvantage: An early detection of a possible solution is neglected. One knows only that a solution exists if all variables are fixed and no conflict did occur.

Algorithm 5.6 TimeStampTernaryClauseReduction(l_i)

```
1: while clause := next clause( l_i ) do
2:   lit1 := OtherLiteral1( clause )
3:   lit2 := OtherLiteral2( clause )
4:
5:   if TA[ lit1 ].stamp ≥ CurrentTimeStamp then
6:     if TA[ lit1 ].stamp && NEGATIVE then
7:       if TA[ lit2 ].stamp ≥ CurrentTimeStamp then
8:         if TA[ lit2 ].stamp && NEGATIVE then
9:           return 0
10:        end if
11:      else
12:        Propagate( lit2 )
13:      end if
14:    end if
15:  else if TA[ lit2 ].stamp ≥ CurrentTimeStamp then
16:    if TA[ lit2 ].stamp && NEGATIVE then
17:      Propagate( lit1 )
18:    end if
19:  else
20:    DiffCurrent += TA[ lit1 ].weight × TA[ lit2 ].weight
21:  end if
22: end while
```

5.2.3 Remarks on accuracy

With the calculation of the Diff one needs to comprimise between accuracy and speed: Some noise in the Diff , caused by a fast but less accurate calculation, may often result in an identical choice for the branching variable. However, if the noise is too 'loud', an alternative branching variable could be selected by the lookahead procedure. This different variable may result in a overall loss of speed.

The complex Diff used in march, described in paragraph 4.1.3, weighs all the new binary clauses during lookahead. In the current implementation of TSTCR, some noise is caused by binary clauses that are incorrectly included in the calculation of the Diff: A ternary clause that is reduced to a binary clause which will get satisfied in the same IUP, should not be considered in the Diff.

To reduce this noise, march uses IFIUP during the lookahead procedure. This procedure results in the highest possible amount of variables stamped with the CurrentTimeStamp. Due to the addition of constraint resolvents, IFIUP reduces the noise even further.

Recall that the lookahead procedure is executed iteratively until no new forced literals are found. Since most of the nodes in the search-tree contain at least one forced literal, the majority of constraint resolvents is added during the first iteration and therefore the actual Diff is calculated in latter iterations. IFIUP will stamp almost all variables before the first call of TSTCR, resulting in a minor noise in the Diff .

Two methods could be considered to calculate the exact value of diff: The first keeps track of all clauses that were reduced to a binary clause and stores them in a list. Then, every satisfied clause is checked whether it occurs on the list. If this is the case, the Diff is decreased with the same amount it was increased.

A more interesting approach to obtain an accurate Diff involves the iteratively repetition of the lookahead procedure until no new constraint resolvents could be added. Both the accuracy and the computational costs are comparable to the method described above. However, the added constraint resolvents may decrease the size of the tree, while the first method shows no additional advantages. Since the use of IFIUP already ensures a high accuracy of the Diff, the costs of both procedures resulted in a slowdown of the solver on practically all benchmarks.

5.3 Local literal storage

On practically all benchmarks, march spends a major part of computation on the lookahead of variables. During the lookahead phase, TCR (algorithm 5.3) is the most time consuming procedure. It is called for every variable that is propagated during lookahead and shortens all ternary clauses in which it occurs negative. Optimisation of this procedure is very important, even if it may result in a slowdown of other parts of the code. In the end, it is the total computational time that matters most.

A considerable part of the computational costs of TSTCR is spent on fetching both other literals in a shortened ternary clause. This section offers three data structures. For each of them, the costs to fetch these literals is analysed by presenting four variations of the FetchLiterals procedure. Each could be used as the first part of the TSTCR procedure. Since march uses a 3-Sat translator, all data structures are explained in assumption that the maximum length of the clauses is three.

To support the explanation of the three data-structures, formula $\mathcal{F}_{\text{store}}$ is used to fill an example of each data-structure. The way formula $\mathcal{F}_{\text{store}}$ is stored, is of great influence on the performance of the solver. Arranging it in such a way that the bottleneck of computation swells, the total time spent on solving a problem can decrease considerably.

$$\mathcal{F}_{\text{store}} = (a \vee b \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c)$$

5.3.1 Clause database structure

A straightforward and commonly used design for a data-structure to store a CNF, is a combination of a clause database and variable arrays pointing towards it. For every clause, the clause database (in algorithms ClsDb) contains an array of the literals that occur in it, and an array with the length of the clauses. Such an array is redundant since there are only ternary clauses. Recall that binary clauses are stored in separate arrays (see section 5.1). For each variable, the variable arrays contain the indices of the clause they appear in. A lookup table with the positions within the clause is optional. Figure 5.2 shows such a data-structure filled with $\mathcal{F}_{\text{store}}$. Notice that the indexing of the clauses and the positions within the clauses start at zero.

This data structure is not optimal for march because it does not efficiently fetch both other literals within a ternary clause. This does not mean that this structure is not optimal for most solvers. However, since the TSTCR procedure is on average the bottleneck in terms of computational time, this data-structure is not optimal for march. The reason for this bottleneck lies in the usage of the TimeAssignment data-structure (section 5.2) and the use of the complex Diff

$clause_0$	a	b	c
$clause_1$	a	$\neg b$	$\neg c$
$clause_2$	$\neg a$	b	$\neg c$

(a)

a	0	1
$\neg a$	2	
b	0	2
$\neg b$	1	
c	0	
$\neg c$	1	2

(b)

a	0	0
$\neg a$	0	
b	1	1
$\neg b$	1	
c	2	
$\neg c$	2	2

(c)

Figure 5.2 — Data structure with a clause database (a), variable arrays (b) and an optional lookup-table (c).

(paragraph 4.1.3).

Algorithm 5.7 FetchLiteralsI(l_i)

for $0 \leq j <$ LiteralArrayElements[l_i] **do**
  $clause :=$ LiteralArray[l_i][j];
  if Cl[$clause$][0] $= var$ **then**
    $literal_1 :=$ ClsDb[$clause$][1]
    $literal_2 :=$ ClsDb[$clause$][2]
  else if Cl[$clause$][1] $= var$ **then**
    $literal_1 :=$ ClsDb[$clause$][0]
    $literal_2 :=$ ClsDb[$clause$][2]
  else
    $literal_1 :=$ ClsDb[$clause$][0]
    $literal_2 :=$ ClsDb[$clause$][1]
  end if
  ...
end for

A procedure to fetch both remaining literals within this data-structure is presented in algorithm 5.7 FetchLiteralsI. The double if-then-else construction in this algorithm is quite inefficient, which makes the data structure not very suitable for a fast lookahead. However, there is the possibility to add a lookup-table with positions to the data structure. The algorithm would then look like FetchLiteralsII. This appears much faster, but the modulo 3-instructions require almost the same computational time as the if-then-else construction used in FetchLiteralsI.

5.3.2 Literal list structure

The addition of a lookup-table in the prior paragraph seems to reduce computation, because there are no branches involved in procedure FetchLiteralsII

Algorithm 5.8 FETCHLITERALSII(l_i)

for $0 \leq j <$ LiteralArrayElements[l_i] **do**
  $clause$ = LiteralArray[l_i][j];
  $index$ = LookupArray[l_i][j];
  $literal_1$ = ClsDb[$clause$][($index$ + 1) modulo 3];
  $literal_2$ = ClsDb[$clause$][($index$ + 2) modulo 3];
  ...
end for

that could have a negative effect on the execution pipelining. However, it has two disadvantages: The slow modulo-3 operations and the additional memory needed to store the lookup-table. In this paragraph, the data structure is slightly rearranged to overcome both negative effects.

First, the removal of the modulo 3 operations can be achieved by a simple exchange between space and time. Instead of storing three literals of every ternary clause, five literals are stored, where the fourth equals the first and the fifth equals the second. Algorithm FETCHLITERALSII could then be applied on this structure, but without the costly modulo operations.

It is even possible to achieve the same result with only four literals stored for each ternary clause: Choosing the table entry to point to the first position in the clause with the remaining two literals in succeeding locations, makes it sufficient to extend the clause with only the first literal.

Second, the memory overhead caused by the lookup-table can be regained by merging the variable/clause-index table with the lookup table: The 2-dimensional clause database can be viewed as a 1-dimensional literal-list. Instead of pointing towards the index of the clause, an entry can point to the index of the literal it is referring to. This results in a variable/literal-index table.

$literal_{0-3}$	a	b	c	a
$literal_{4-7}$	a	$\neg b$	$\neg c$	a
$literal_{8-11}$	$\neg a$	b	$\neg c$	$\neg a$

(a)

a	1	5
$\neg a$	9	
b	2	10
$\neg b$	6	
c	0	
$\neg c$	4	8

(b)

Figure 5.3 — Data structure with a literal-list (a) and a variable arrays (b).

Integration of both these changes results in a data structure that requires only slightly more space than the structure without the lookup-table (Figure 5.2 (a) and (b)). On the other hand, the costs of fetching the remaining literals are significantly reduced. This speed-up is shown in algorithm FetchLiteralsIII .

Algorithm 5.9 FetchLiteralsIII(l_i)

for $0 \leq j <$ LiteralArrayElements[l_i] **do**
  $index$:= LiteralArray[l_i][j];
  $literal_1$:= literallist[$index$];
  $literal_2$:= literallist[$index + 1$];
  ...
end for

5.3.3 Local literal structure

The data-structure of the prior paragraph can be even further optimised towards the fetching of both other literals. When observing algorithm 5.9, the only possibility left for another speed-up is the removal of the lookups in the literal list. Obtaining the literals from the list could be a costly operation when the list is too large to fit in the cache. This is usually the case with industrial instances. In these cases, the data-structure of this paragraph could result in a speed-up of a double digit percentage. However, when the literal-list is small, the optimisations described below do not accomplish any performance boost.

The literal-list (Fig. 5.3 (a)) can be removed by placing both literals that have to be fetched in the variable array. This results in a data-structure in which every literal occurs twice in the variable arrays. The literal list then becomes redundant, because nothing points towards it anymore.

a	b	c	$\neg b$	$\neg c$
$\neg a$	b	$\neg c$		
b	a	c	$\neg a$	$\neg c$
$\neg b$	a	$\neg c$		
c	a	b		
$\neg c$	a	$\neg b$	$\neg a$	b

Figure 5.4 — Data structure with a literals stored in the variable arrays

Since the data structure is completely optimised towards the fetching the remaining literals, the algorithm becomes quite simple. However, this specialization comes with some additional conditions for a correct application: The removal of the literal pointers rules out the possibility to see whether a clause is satisfied or not. A procedure that removes all inactive ternary clauses, described in the next section, remains no longer an optimisation but will become a necessary enhancement.

Algorithm 5.10 FetchLiteralsIV(l_i)

for $0 \leq j <$ LiteralArrayElements[l_i] **do**
  $literal_1$:= LiteralArray[l_i][$2j$];
  $literal_2$:= LiteralArray[l_i][$2j + 1$];
  ...
end for

5.4 Removal of inactive clauses

Inactive clauses could increase the computational costs of the procedures performed during the lookahead phase. Two important causes could be appointed: First, the larger the number of clauses considered during the lookahead, the poorer the performance of the cache. Second, if both active and inactive clauses occur in the active data-structure during the lookahead, a check is necessary to determine the status of every clause. Removal of inactive clauses from the active data-structure prevents these unfavourable effects from taking place.

The procedure that profits most from the removal of inactive clauses is the TernaryClauseReduction procedure. Two types of this procedure are presented in algorithm 5.11 and 5.12. Type I is a variant that does not remove inactive clauses, while type II does. The latter has two advantages: (1) ActiveClausesArray is smaller than ClausesArray, making the for-loop shorter and (2) the active-check is removed. Both result in a speed-up.

Algorithm 5.11 TernaryClauseReductionI(l_i)

for all clauses in ClausesArray[l_i] **do**
  if *currentClause* is active **then**
    reduce length of *currentClause* by 1
    if reduced length equals 1 **then**
      Propagate(remaining literal)
    end if
  end if
end for

Algorithm 5.12 TernaryClauseReductionII(l_i)

for all clauses in ActiveClausesArray[l_i] **do**
  reduce length of *currentClause* by 1
  if reduced length equals 1 **then**
    Propagate(remaining literal)
  end if
end for

When a variable x is assigned to a certain truth value on the chosen path, all the ternary clauses in which it occurs become inactive in the arrays pointing to clause indices: The clauses in which x occurs positive gets satisfied, while those clauses in wich it occurs negative are reduced to binary clauses. They become inactive, since binary clauses are moved to the implication arrays.

In the example below, inactive clause 28 will be removed from the formula. This clause exists in the arrays of both remaining literals that haven't been fixed yet: x_i and x_j. The index of the clause is swapped with the last active one of the array. Afterwards, the size of the array is decreased to hide the clause. Backtracking this procedure is cheap: Increasing the size of the array is sufficient.

x_i [6]	12	45	28	15	3	22	31	9
x_j [5]	34	28	10	26	5	41	7	13

$$\Downarrow$$

x_i [5]	12	45	22	15	3	28	31	9
x_j [4]	34	5	10	26	28	41	7	13

Figure 5.5 — Swapping a satisfied clause out of the arrays.

Swapping indices, without an additional lookup-table is usually very expensive. The position of the inactive clause in the array must be known. It can be obtained by a loop through the active clauses, which will result in high costs when it is located at the end of a long array. Enhancing the data structure with a lookup-table containing those positions, renders the swapping procedure much more efficient.

However, the memory required to store such a lookup-table is relatively large: Its size equals the number of literals. This results in an increment of used memory by almost 50% on problems with a high literal-to-variable ratio. Because this table will only be used on the chosen path, it will not raise the number of cache misses during the lookahead phase.

Chapter 6

Equivalence reasoning

Structural logical formulas sometimes yield a substantial fraction of so called equivalence clauses after translating to CNF. The best known example of this feature is probably provided by the `parity`-family. The larger such CNF formulas cannot be solved in reasonable time if no extra reasoning with - and detection of - these clauses is incorporated. That is, in solving these formulas, there is a more or less separated algorithmic device in dealing with these equivalence clauses - called equivalence reasoning - and another device dealing with the remaining clauses. This chapter proposes a way to align these two reasoning devices by introducing parameters for which we establish optimal settings over a variety of existing benchmarks. We obtain a truly convincing speed up in solving such formulas with respect to the best solving methods existing so far.

To illustrate this alignment, we first present a historical overview of equivalence reasoning in SAT solvers. This is followed by the description of some procedures required in the pre-processor to effectively initialise the data-structure containing the equivalences. Section 6.3 offers our proposed alignment with some experimental evidence. Here, the necessary changes of the pre-selection heuristics are also described. Finally, three additional forms of equivalence reasoning have been considered and experimented with to further increase the general performance of march. Section 6.4 deals with these procedures.

6.1 Historical overview

The notorious `parity-32` benchmarks [7] remained unsolved by general purpose SAT solvers for a considerable time. In [30] a method was proposed which, for the first time, could solve these instances in a few minutes. The key to this method was to detect the clauses which represented so called *equivalences* $l_1 \leftrightarrow l_2 \leftrightarrow \cdots \leftrightarrow l_n$ (where the l_i are literals, or their negations, that appeared in the formula at hand) and to pre-process the set of these equivalences in such away that dependent and independent variables became visible. The remaining clauses then where tackled with a rather straightforward DPLL procedure but in such away that it kept track of the role of these dependent and independent variables. As it was developed, it was a two-phase method, in which the equivalence part was established and transformed in a pre-processing phase.

The next important step was made by Li [18, 19], who incorporated a kind of equivalence reasoning in any node of an emerging search tree. His approach did not incorporate a pre-processing phase and thus he established the first one-phase SAT solver eqsatz which could tackle these instances in reasonable time.

A disadvantage of his method is that Li uses a full lookahead approach (very costly for larger size formulas) and does not evaluate the speed in reduction of the formula in an (to our opinion) optimal way. Also, his equivalence reasoning is restricted to equivalences with a length of three at most.

Some years later Ostrowski *et al.* [23] extended the above pre-processing ideas from [30] to logical gates other than equivalences, resulting in the lsat solver. However, their DPLL approach to deal with the remaining CNF-part uses a Jeroslow-Wang branching rule and they do not perform a lookahead phase, which is (again to our opinion) not an optimal alignment.

We propose an alignment of equivalence reasoning and DPLL reasoning which does not assume a full lookahead approach. This will enforce us to introduce adequate pre-selection heuristics for selecting variables which are allowed to enter an Iterative Unit Propagation phase. Further, we will evaluate the progress in enrolling the formula at hand in a more detailed manner as was done in eqsatz . We are forced to introduce parameters to guide this search. These parameters are needed to aggregate the reduction of the equivalence part of the formula and that of the remaining CNF part. Further, our method is able to deal with equivalences of arbitrary size. In turn, this leads us to an investigation of the relative importance of equivalences of different size. Surprisingly, this relative importance turns out to be measured rather differently as would be expected when taking similar relative importance of ordinary clause-lengths as a guiding principle.

We optimise the various parameters to ensure a convincing speed up in solving formulas with a substantial equivalence part, both with respect to the various alternative solvers available and with respect to a variety of benchmarks known of this type.

6.2 Pre-processing

This section deals with additional procedures on equivalence reasoning in the pre-processor. First, the basics of equivalence reasoning are presented. This is followed by a short description of the first phase of the algorithm by Warners and Van Maaren [30] to solve the `parity32` family. Finally, two enhancements are proposed to this algorithm that considerably reduces the computational time to solve instances with many equivalence clauses.

6.2.1 Preliminaries

The *sign* of clauses is an important concept in de detection of equivalence clauses within the CNF. A clause is called to have a *positive sign* when the number of positive literals in that clause is odd. When a clause has an even number positive literals, it is called to have a *negative sign*.

If in the CNF, 2^{n-1} clauses exist of length n and all literals of these clauses refer to the same variables and all clauses have equal signs, then these clauses could be replaced by an equivalence clause of length n. Since the clauses are sorted during the pre-processing (see section 3.1) equivalence clauses could be easily detected. In figure 6.1 (a) four example clauses are presented that could be replaced by the equivalence clause in figure 6.1 (b). In the CoE-part, variables $x_i \in \{-1, 1\}$. This enables us to represent equivalence clauses as a multiplication.

$$\begin{array}{l} x_1 \vee x_2 \vee x_3 \\ \neg x_1 \vee \neg x_2 \vee x_3 \\ x_1 \vee \neg x_2 \vee \neg x_3 \\ \neg x_1 \vee x_2 \vee \neg x_3 \end{array}$$

(a)

$$x_1 * x_2 * x_3 = 1$$

(b)

Figure 6.1 — (a) All four clauses with a positive sign and literals referring to x_1, x_2 and x_3. (b) The logical equivalent equivalence clause of (a).

All detected equivalence clauses are placed in a separate data-structure. We refer to this data-structure as the *Conjunction of Equivalences* (CoE). There are two important differences between the CoE- and CNF-part of the formula:

- Assigning variable x_i to false will result in the propagation with $x_i = -1$ in the CoE-part and a propagation with $x_i = 0$ in the CNF-part.
- Equivalence clauses are always reduced by one in length when one of its literals is assigned to a truth value. Clauses in the CNF-part are either reduced by one in length or get satisfied, which could be considered a reduction to length zero.

The aim of our equivalence reasoning enhanced pre-processor is to solve the extracted CoE sub-formula. A key procedure required for solving is the substitution of equivalence clauses. Consider the following equivalence clauses:

$$x_1 * x_2 * x_3 = 1 \tag{6.1}$$
$$x_2 * x_3 * x_4 * x_5 = -1 \tag{6.2}$$

Recall that $x_i \in \{-1, 1\}$ in the CoE, so $x_i * x_i = 1$. Elimination of variable x_2 from equivalence clause (6.2) could be achieved by replacing its occurrence by $x_1 * x_3$. This changes the above equivalence clauses to:

$$\begin{aligned} x_1 * x_2 * x_3 &= 1 \\ x_1 * (x_3 * x_3) * x_4 * x_5 &= -1 \end{aligned}$$

Which is equal after removal of the square to:

$$\begin{aligned} x_1 * x_2 * x_3 &= 1 \\ x_1 * x_4 * x_5 &= -1 \end{aligned}$$

The substitution of one equivalence clause by another can be realised through an elementary procedure. This procedure is formulated in algorithm 6.1 SUBSTITUTE(Q_k, Q_l), in which both parameters Q_k, Q_l refer to equivalence clauses. Notice that the particular variable eliminated is not explicitly needed in algorithm 6.1: The resulted equivalence clauses above would be identical when we substituted x_3 by $x_1 * x_2$.

Algorithm 6.1 SUBSTITUTE(Q_k, Q_l)

```
Ql.sign := Ql.sign × Qk.sign
for every variable xi ∈ Qk do
  if xi ∈ Ql then
    remove xi from Ql
  else
    add xi to Ql
  end if
end for
```

A contradiction could only be derived when during the substitution process, two equivalence clauses exist in the CoE whose literals are identical and whose signs are opposite. Substituting both equivalence clauses with each other would result in the inequality 1 = -1.

6.2.2 The original algorithm

After the SIMPLIFYFORMULA procedure, all detected equivalence clauses are extracted and placed in the CoE. To obtain a solution from the CoE, the first phase of the algorithm by Warners and Van Maaren [30] is executed: We initialise set $\mathscr{I} = \{x_1, \cdots, x_m\}$, the set of *independent* variables, with m referring to the initial number of variables. We loop through the equivalency clauses once, choosing variable x_i in each one to eliminate from all other equivalence clauses. Subsequently, we remove x_i from $\mathscr{I}$, and call it a *dependent* variable. The result is a set of equivalence clauses for which all satisfiable assignments can be constructed by assigning all possible combinations of truth values to the independent variables. The values of the dependent variables are uniquely determined by an assignment of the independent variables. Note that during the elimination process a contradiction might be derived, which implies unsatisfiability of the original problem.

This procedure is described in algorithm 6.2. Note that the procedure Choose in this algorithm just selects a random variable. Let us look at the algorithm more in detail. Consider the following example CoE consisting of three ternary equivalences:

$$x_1 * x_2 * x_3 = 1 \tag{6.3}$$
$$x_2 * x_4 * x_5 = -1 \tag{6.4}$$
$$x_5 * x_3 * x_1 = 1 \tag{6.5}$$

A dependent variable is denoted as $\underline{x_i}$. After initialisation of $\mathscr{I}$, we loop thought the equivalence clauses. In (6.3) we randomly select a variable, say x_2, to eliminate from all other equivalence clauses. The result of the substitutions is presented in figure 6.2 (a). In the second step, we randomly select x_3 from the second equivalence clause in figure 6.2 (a) to eliminate. The resulting CoE is equal to figure 6.2 (b). Yet, the last equivalence clause in this figure contains one variable. After eliminating it from the other equivalences by fixing it on its forced value false, the solved CoE is equal to figure 6.2 (c). Notice that we end up with $\mathscr{I} = \{x_1, x_5\}$.

$$\begin{array}{rcl} x_1 * \underline{x_2} * x_3 &=& 1 \\ x_1 * x_3 * x_4 * x_5 &=& -1 \\ x_5 * x_3 * x_1 &=& 1 \end{array} \qquad \begin{array}{rcl} \underline{x_2} * x_4 * x_5 &=& -1 \\ x_1 * \underline{x_3} * x_4 * x_5 &=& -1 \\ x_4 &=& -1 \end{array} \qquad \begin{array}{rcl} \underline{x_2} * x_5 &=& 1 \\ x_1 * \underline{x_3} * x_5 &=& 1 \\ \underline{x_4} &=& -1 \end{array}$$

(a) (b) (c)

Figure 6.2 — Possible execution of the algorithm on the example CoE. (a) After elimination of x_2, (b) after elimination of x_3 and (c) after the elimination of x_4.

6.2.3 Enhancements to the algorithm

Numerous independent sets could be obtained by the above algorithm. The performance of a solver may vary significantly under different choices of the independent set, as we have observed using our march solver. Therefore, two enhancements are added to the original algorithm to find an independent set that would result in a relatively fast performance: The first addition is an explicit

Algorithm 6.2 SUBSTITUTEEQUIVALENCES( )

$\mathscr{I}$:= all variables occurring in $\mathcal{F}$
for every equivalence clause Q_k in CoE **do**
  if Q_k is equal to "1 = -1" **then**
    return "unsatisfiable"
  end if
  $y :=$ CHOOSE(Q_k)
  for every equivalence clause Q_l, so $y \in Q_l$ **and** $k \neq l$ **do**
    SUBSTITUDE(Q_k, Q_l)
  end for
  remove y from $\mathscr{I}$
end for
return "satisfiability undetermined"

prescription for the selection of the variables to eliminate and the second is a procedure that shortens the equivalence clauses after executing the algorithm.

Explicit variable selection

Changing the implementation of the CHOOSE procedure form selecting a random varaible to a more "educated guess" resulted in a speed-up. Our procedure CHOOSE(Q_k) selects the variable in equivalence clause Q_k that occurs least frequent in the CNF-part of the formula. Occurrences in binary clauses (occ_2) are weighed ten times as important as occurrences in ternary clauses (occ_3). The factor ten is just a "guessed" magical constant.

The motivation for selecting the least occurring variable is twofold: First, if one of the variables in Q_k does not occur in the CNF at all, Q_k becomes a *tautological equivalence clause* after substitution of that variable, because the variable could always satisfy the clause. This means that Q_k is not needed during the solving procedure. Neglecting tautological clauses during the solving phase could result in a substantial speed-up.

Second, faster reduction of the formula is expected when the variables in the independent set occur frequently in the CNF-part: Independent variables will be forced earlier to a certain truth value by constraints from both the CoE- and CNF-part. High frequency of the independent variables in the CNF-part is achieved by making the variables with the lowest frequency dependent.

Algorithm 6.3 CHOOSE(Q_k)

$$y := \min_{x_i \in Q_k} \Big(10 \times \big(occ_2(x_i) + occ_2(\neg x_i)\big) + occ_3(x_i) + occ_3(\neg x_i)\Big)$$

return y

Shortening of equivalence clauses

After the execution of SubstituteEquivalences, the resulting equivalence clauses may be very long. Reducing their size will increase solving speed. Two methods can be distinguished regarding the shortening of equivalence clauses. Both will be illustrated using the following example:

$$\begin{aligned}
\underline{x_1} * x_2 * x_3 &= -1 && (6.6)\\
\underline{y_1} * y_4 * x_2 * x_3 &= 1 && (6.7)\\
\underline{y_2} * y_3 * y_4 * x_3 &= -1 && (6.8)\\
y_3 * y_4 * \underline{y_5} &= 1 && (6.9)
\end{aligned}$$

We refer to *global substitution* as the full replacement of a variable in the CoE. This substitution does not change the size of the independent set. A variable becomes eligible for global substitution if its substitution will decrease the sum of the lengths of the non-tautological equivalence clauses. In the example above, only variable x_2 in equivalence clause (6.6) satisfies this requirement. The CoE will change to:

$$\begin{aligned}
x_1 * \underline{x_2} * x_3 &= -1\\
\underline{y_1} * y_4 * x_1 &= -1\\
\underline{y_2} * y_3 * y_4 * x_3 &= -1\\
y_3 * y_4 * \underline{y_5} &= 1
\end{aligned}$$

With *local substitution* it is not necessary to replace a variable in all equivalence clauses in which it occurs. When local substitution is applied, the size of the independent set may increase. The advantage of local substitution is that one could further reduce to sum of lengths. An example of local substitution is the replacement of variable y_4 in equivalence clause (6.9) with y_4 in (6.8): The dependency of variable y_5 disappears and the total length of CoE is additionally reduced by one. Applying this to the example would result in the following CoE:

$$\begin{aligned}
x_1 * \underline{x_2} * x_3 &= -1\\
\underline{y_1} * y_4 * x_1 &= -1\\
\underline{y_2} * y_5 * x_3 &= -1\\
y_3 * y_4 * y_5 &= 1
\end{aligned}$$

Only few experiments have been performed regarding the optimal combination of global and local substitution. The best result we obtained so far reduced the sum of lengths of all non-tautological equivalences without increasing the size of the independent set. This goal was achieved by the two-phase procedure ShortenEquivalences: The first phase searches for pairs of equivalence clauses that could be combined to create a binary equivalence (local substitution). Note that binary equivalences are always tautological, since one of its literals could be removed from the CNF by replacing it by the other. Due to the creation of a tautological equivalence clause, the size of the independent set does not increase. The second phase loops through all equivalence clauses and checks whether a different choice for the dependent variable in that clause would result in a smaller sum of lengths of non-tautological equivalences (global substitution). Both phases are iteratively repeated until both yield no further reduction. In table 6.1 the gain of ShortenEquivalneces is presented. The number between braces denotes the number of instances in a family. The `parity32` family is contributed by Crawford [7], the `2000009987x` by Purdom [24] and both `hwb` families by Stanion [3].

	Without ShortenEquivalences		**With** ShortenEquivalences	
Benchmarks	length	time(s)	length	time(s)
`parity32` (10)	23.22	148.47	5.26	**3.53**
`2000009987x` (3)	6.69	6877.34	3.02	**430.00**
`hwb-n20` (3)	6.35	83.08	4.49	**43.02**
`hwb-n22` (3)	6.74	307.32	4.68	**131.41**

Table 6.1 — Influence of the ShortenEquivalences procedure on the average length of the equivalence clauses and the average time required by march to solve the given benchmarks

6.3 Aligning CNF- and CoE-reasoning

Since the boolean formula is split in a CoE- and a CNF-part, it seems reasonable to base the evaluation of the lookahead not merely on the reduction of the CNF-part, but also on the reduction of the CoE-part. To obtain an adequate alignment between both parts, we first reviewed existing evaluation procedures that could somehow be applied. Second, experiments were performed using a linear regression model. With the insights provided by its results, we defined heuristics to measure the importance of the reduction of equivalence clauses. By these heuristics, march is able to outperform all other existing procedures on benchmarks which contain a considerable CoE-part. Because our pre-selection heuristics are based on an approximation function of the Diff, and the Diff will change due to the alignment, a modification of the pre-selection heuristics that integrates the alignment is presented at the end of this section.

6.3.1 Existing evaluation procedures for equivalence clauses

As stated before, a Diff that counts newly created binary clauses, is widely used. This Diff is powerful on random 3-Sat and many other benchmarks.

However, epsecially on random k-Sat formulas, with k > 3, it appears fruitful to use sa Diff that counts *all* newly created clauses, and to multiply each new clause of length n with a weight that expresses the importance of a new clause of length n. Weights that result in optimal performance on random k-Sat formulas could be described by linear regression: e.g. Kullmann [15] uses weights in his OKsolver that could be approximated by 0.22^{n-2}. In this equation, n refers to the length of a clause, with $n \geq 2$.

Little is known about effective evaluation functions to measure the importance of a new equivalence clause. In eqsatz by Li [18, 19], only new binary equivalences are counted. These are weighed twice as important as a new binary clause. The importance of the new equivalence clauses of various length could be obtained by measuring the reduction of its translation into CNF. Applying the approximation of the weights by Kullmann [15] results in a weight function of $2^{n-1} \times 0.22^{n-2} = 10.33 \times 0.44^n$ for a new equivalence of length n. However, this reference is vague since the weights are optimized with respect to random formulas.

6.3.2 Combined lookahead evaluation

Although we have indications that other models might be more appropriate when equivalence clauses are involved, we take this regression model as a first start. Performances were measured for various parameter settings of equation 6.10. In this equation, n refers to the reduced length of an equivalence clause. Parameter q_{base} denotes the factor that describes the decreasing importance of equivalence clauses of various lengths and parameter q_{const} expresses the relation between the reduction of the CNF-clauses and the equivalence clauses. Since march uses a 3-Sat translator, only new binary clauses are created. The evaluation of the lookahead is calibrated by defining the importance of a new binary clause to value 1. The result of eq_n then defines the importance of a new equivalence clause of length n in relation to a new binary clause.

$$eq_n = q_{\text{const}} \times {q_{\text{base}}}^n \tag{6.10}$$

Wide scale experiments were troubled by the lack of useful benchmarks: Many benchmark families that contain a significant part of equivalence clauses are easily solved using only the pre-processing procedures: Either the solving procedure for the CoE results in a contradiction, or the propagation of the unary clauses and the binary equivalences found during pre-processing are sufficient to solve the formula. Many benchmarks families with a significant CoE-part that require a sophisticated solving procedure after pre-processing are also not useful for these experiments, because most or all of their equivalence clauses have length 3. For comparison: The Sat 2003 [25] competition set consisted of 11 families which are solved in pre-processing while only five needed further searching. Of those five, only two had a large number of long equivalences after pre-processing.

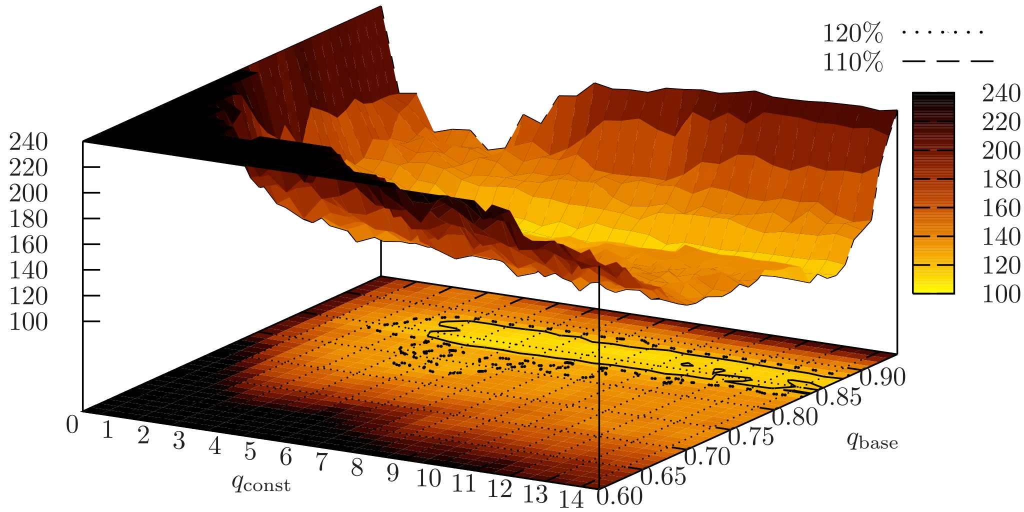


Figure 6.3 — Performances achieved by march on various settings of q_{base} and q_{const}. The values on the z-axis are the cumulated performances on the whole `parity-32` and `hwb-n20` families in seconds. Contour lines are drawn at 110% and 120% of optimal performance.

Reduced length (n):	2	3	4	5	6	7	8	9	10
CNF-reference:	2.00	0.88	0.39	0.17	0.07	0.03	0.01	0.01	0.00
Found optimum:	3.97	3.38	2.87	2.44	2.07	1.76	1.50	1.27	1.08

Table 6.2 — Weights to measure the reduction of equivalence clauses of various length.

These two families are the `parity32` and the `hwb`. The first family consists of the SAT-encoding of minimal disagreement parity problems contributed by Crawford *et al.* [7]. The second consists of equivalence checking problems that arise by combining two circuits computed by the hidden weighed bit function. The latter are contributed by Stanion [25]. We used both families to determine the parameter setting for equation (6.10) that results in optimal performance. The results of these experiments are shown in Fig. 6.3. The optimal values we found were $q_{\text{const}} = 5.5$ and $q_{\text{base}} = 0.85$. Two conclusions can be drawn regarding the results: (1) Parameter q_{base} has a much stronger influence on the performance than q_{const}. (2) Using optimal settings, the reduction of equivalences is considered far more important than the reduction of the equivalent CNF-translations would suggest: Tab. 6.2 shows the weights for both settings.

6.3.3 Pre-selection heuristics

A combined evaluation of the lookahead results in a less accurate ranking of the variables by pre-selection heuristics: The lookahead evaluation is based on the reduction of both equivalence clauses and CNF-clauses, while the pre-selection heuristics consider only CNF-clauses. Additional heuristics for equivalence clauses are therefore necessary to increase accuracy. Recall the equation used to rank the variables:

$$\begin{aligned} \textsc{Daf}(x) &= occ_3(\neg x) + \sum_{\neg x \vee y_i \epsilon \mathcal{F}} occ_3(\neg y_i) && (6.11) \\ \textsc{Rank}(x) &= \textsc{Daf}(x) \times \textsc{Daf}(\neg x) && (6.12) \end{aligned}$$

In this equation, $occ_3(x)$ refers to the occurrences of literal x in ternary clauses. The reduction of equivalence clauses caused by the lookahead on a literal could be measured in a similar way as the approximation of new binary clauses: The occurrences in ternary clauses approximate the new binary clauses. Likewise, equivalence clauses of length n approximate the new equivalence clauses of length $n-1$.

To obtain an accurate approximation of the importance of the reduction of the CoE-part of the formula, the reduction of equivalence-clauses of various lengths need to be measured according their relative importance. To this extent, the equation that appeared optimal for measuring the reduction of equivalence clauses during the lookahead is applied:

$$eq_n = 5.5 \times 0.85^n$$

Here, n refers to the length of the reduced equivalence clause. We refer to $\mathcal{E}(x)$ as the set of all equivalence clauses in which variable x occurs. Note that $\mathcal{E}(x)$ is equal to $\mathcal{E}(\neg x)$, since there is no distinction between positive and negative literals in equivalence clauses. Using this set we could define an approximation function of the combined lookahead evaluation (Ace). The ranking of variable x could then calculated by multiplying $\textsc{Ace}(x)$ and $\textsc{Ace}(\neg x)$.

$$\textsc{Ace}(x) = occ_3(\neg x) + \sum_{Q_i \epsilon \mathcal{E}(x)} eq_{|Q_i|-1} + \sum_{\neg x \vee y \epsilon \mathcal{F}} \left(occ_3(\neg y) + \sum_{Q_i \epsilon \mathcal{E}(y)} eq_{|Q_i|-1} \right) \tag{6.13}$$

The computational cost to compute Ace for all free variables at every node in the tree could become rather large if a benchmark contains many long equivalence clauses, since a double sum is required to obtain Ace. However, experimental results show that for many benchmarks that contain equivalence clauses, the costs to compute these pre-selection heuristics are modest enough to achieve significant gains in performance.

6.4 Additional equivalence reasoning

The alignment of the CNF- and CoE-reasoning is probably the most powerful additional equivalence reasoning after the proposed pre-processor enhancements. Various other forms of equivalence reasoning have been considered and experimented with to extend the performance gain: The first consists of the removal of tautological equivalences and CNF-clauses that are the consequence of inactive variables.

Second, we considered "real" binary equivalence propagation - replacing a variable throughout the formula by its logical equivalent - was considered, but it appeared very time consuming. Therefore no "real" binary equivalence propagation was fully implemented. Yet, we did implement two other forms of equivalence reasoning that similize the two most important advantages of binary equivalence propogation and require far less computational time. These two advantages are (1) preventing equivalent variables from entering the lookahead phase and (2) reducing the size of the equivalence clauses.

6.4.1 Inactive variables

This paragraph describes the uses and advantages of communicating inactive variables between the CNF- and the CoE-part of the formula. If a variable is inactive in either the CNF- or the CoE-part of the formula, all clauses in which this variable still occurs could be removed in some situations. These situations can be divided into two distinct groups:

I) if a variable does not occur in the CNF-part of the formula and only once in the CoE-part, then it occurs only in a tautological equivalence clause which could be removed; this clause can always be satisfied at the end of the solvings.

II) if a variable does not occur in the CoE-part of the formula and occurs monotone (only positive or only negative) in the CNF-part, then all clauses in which it occurs could be satisfied by fixing the variable on its dominant value.

The advantages of the removal of these clauses is twofold: First, their removal may decrease the total execution time: Fewer propagations need to be performed on the chosen path and during the lookahead. However, the detection of inactive variables and their actual removal may increase computational costs. These costs could largely flatten the potential performance gain.

The second and most important advantage is the increased accuracy of the evaluation of the lookahead as result of the removal of these clauses: The evaluation of the lookahead is measured by the reduction of clauses. This measurement is based on the idea that reduction of clauses renders a formula less satisfiable, since every reduction is an extra constraint. Branching on a variable with a high evaluation of its lookahead may lead to an early contradiction. However, when clauses can be removed, their reduction will not produce extra constraints and they will trouble the accuracy of the lookahead evaluation.

In order to benefit form these advantages, the number of checks whether clauses could be removed is significantly reduced: First, no checks are performed during the lookahead because frequent checking will cost more than it would gain. Second, the checks are only performed at the beginning of each node on the chosen path: This is the only location where these checks could result in a more accurate lookahead in a particular node. Third, at the beginning of each node, the number of checks is reduced to those that could result in a positive outcome:

Let us consider the conditions under which both situations occur: Situation I could only occur at the beginning of a node, when a variable that still existed in the CNF of the parent of that node is removed: The frequency of a variable in the CoE could not be decreased to one by IUP, because an equivalence clause could only be satisfied during IUP when all its literals are fixed. This decimates the frequency of all these literals in the CoE to zero.

Situation II may occur at the beginning of a node, because the IUP of the parent node may yield satisfied clauses, which could result in monotone occurrences of variables. However, when all monotone variables are fixed directly at the beginning of the node, new monotone variables could only occur when the following conditions are met: (1) Two variables must both occur only once in the CoE-part; (2) both variables must occur in the same equivalence clause and (3) one of these variables must not occur in the CNF-part, in order for this clause to be labelled as tautological.

Our algorithm uses two lists that keep track of the inactive variables: The CoE-inactive-list and the CNF-inactive-list. Using both inactivity lists, the number of checks could be reduced to a minimum. The algorithm is presented in the three procedures CNF-FixMonotoneVariables, CoE-TautologyCheck, and CNF-CheckMonotone. Only CNF-FixMonotone-Variables is called by default at the beginning of each node. Every new inactive variable in either the CNF- or CoE-part is added to its inactive list and the Check-procedure of the complementary part is called as a consequence. Note that tautological clauses are disabled instead of being removed from the CoE-part: whenever a solution is found, disabled equivalence clauses are necessary to determine the truth value of the variables that disabled them.

Algorithm 6.4 CNF-FixMonotoneVariables( )

```
while variable x_i occurs monotone in CNF and x_i ∈ CoE-inactive-list do
  fix x_i on dominant occurrence and simplify CNF
end while
for variable y_j ∉ CNF and y_j ∉ CNF-inactive-list do
  add y_j to CNF-inactive-list
  CoE-CheckTautology(y_j)
end for
```

Algorithm 6.5 CoE-CheckTautology(x_i)

```
if x_i occurs only once in CoE then
  Q := the only equivalence clause in which x_i occurs
  Disable( Q )
  for variable y_j ∈ Q do
    if y_j occurs only in Q then
      add y to CoE-inactive-list
      CNF-CheckMonotone(y_j)
    else if y_j ∈ CNF-inactive-list then
      CoE-CheckTautology(y_j)
    end if
  end for
end if
```

Algorithm 6.6 CNF-CheckMonotone(x_i)

```
if x_i occurs monotone in CNF then
  CNF-FixMonotoneVariables( )
end if
```

6.4.2 Preventing two equivalent variables from both entering the lookahead

It is obvious that one wants to prevent two equivalent variables to both enter the lookahead phase: Either they result both in a conflict or they yield the same Diff. A simple algorithm that always disqualifies a free variable in every newly detected binary equivalence could cause problems: Cycles of disabled variables could be created. This could result in a situation where no variables are selected by the pre-selection heuristics at all. Consider the following three binary equivalences:

$$x_1 * x_3 = -1 \tag{6.14}$$
$$x_3 * x_6 = 1 \tag{6.15}$$
$$x_1 * x_6 = -1 \tag{6.16}$$

The third binary equivalence follows logically from the first two.

In march a slightly more complex algorithm is used to prevent the creation of cycles. To this extend a linked list biPrior is used. This list contains a pointer for every variable x_i. It points to the variable by which x_i is disqualified to enter the lookahead phase. All pointers of biPrior are initialised to point towards itself, indicating that the variables are still enabled. For every new detected binary equivalence, the procedure DisqualifyVariable is called, which disqualifies one of the variables whenever possible.

A representation of the propagation of the three binary equivalences above is presented in figure 6.4. Figure 6.4(a) shows the initial status of the linked list biPrior. The modification by DisqualifyVariables on the first equivalence results in figure 6.4(b). Likewise, figure 6.4(c) represents the situation after the propagation of the first two binary equivalences. Performing DisqualifyVariables on the third binary equivalence after the propagation of the first two, also results in figure 6.4(c), because GetFreeVariable(x_3) is equal to GetFreeVariable(x_6). This is done on purpose, since disqualifying x_1 would result in a cycle.

Besides its use within the context of equivalence reasoning, the above enhancement is worth considering in a general framework: The only costs of the algorithm is a check whether its complement exists for every new binary clause and binary resolvent on the chosen path. When a new binary equivalence is detected, only the cheap procedure DisqualifyVariable needs to be called.

In contrast, the advantages could be significant: When a full lookahead is performed, every propagated binary equivalence results in the removal of a variable whose evaluation value is already known. If the number of detected binary equivalences is substantial in relation to the number of free variables, the propagation will result in a reasonable reduction of the computational time.

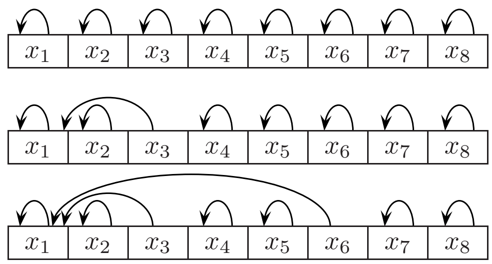


Figure 6.4 — Schematic representation of the biPrior linked list. (a) initial status, (b) after propagation of $x_1 * x_3 = -1$, (c) after the propagation of $x_1 * x_3 = -1$ and $x_3 * x_6 = 1$.

Algorithm 6.7 DisqualifyVariable(x_i, x_j)

```
free_i := GetFreeVariable(x_i)
free_j := GetFreeVariable(x_j)

biPrior[ free_i ] := free_j
```

The possible gain could be even even more significant when a partial lookahead is performed: On average, variables which occur in equivalences tend to have a relatively high evaluation of their lookahead and thus probably have a high pre-selection value. Preventing equivalent variables from entering the lookahead phase will result in a wider lookahead phase with equal costs.

6.4.3 Reduction of equivalence clauses

The reduction of the CoE-part that could be achieved by the propagation of a single binary equivalence, is substantial. The reason is that every ternary equivalence clause in which both literals of the binary equivalence occur could be reduced to a new unary clause. Its propagation could result in new binary equivalences that could further reduce the CoE-part. Likewise, an equivalence clause of length four could be reduced to a new binary equivalence if it contains both literals of the propagated binary equivalence. Propagation of this new binary equivalence may also lead to an additional reduction of the CoE-part.

Algorithm 6.8 GetFreeVariable(x_i)

```
while biPrior[ x_i ] ≠ x_i do
  x_i := biPrior[ x_i ]
end while
return x_i
```

Algorithm 6.9 describes this iterative propagation of new binary equivalences. After the propagation of a binary equivalence throughout the CoE-part of the formula, its CNF equivalent is added to the CNF-part. This will result in the calling of the procedure DisqualifyVariable as presented above. Note that new unary clauses that are created during CoE-IterativeBinaryEquivalencePropagation are propagated throughout the whole formula $\mathcal{F}$ (CoE- and CNF-part).

Algorithm 6.9 CoE-IterativeBinaryEquivalencePropagation( )

```
while binary equivalences exist in CoE do
  while binary equivalence l_i ↔ l_j exists in CoE do
    CoE-BinaryEquivalencePropagation(l_i, l_j)
    add l_i ↔ l_j to CNF
  end while
  F := IUP(F)
end while
```

Algorithm 6.10 CoE-BinaryEquivalencePropagation(l_i, l_j)

```
while equivalence clause Q exists in which literal l_j occurs do
  if literal l_i ∈ Q then
    remove both l_i and l_j from Q
  else
    replace l_j by l_i
  end if
end while
```

Physical replacement of literals in the CoE data structure, as suggested in algorithm 6.10 is a costly operation. Therefore, no physical replacement is performed; merely some symbolic replacement. For every literal a list containig all literals that are equivalent to it, is stored. Propagation of a binary equivalence $a \leftrightarrow b$ could then be achieved by removing all pairs (x_i, y_j) with $a \leftrightarrow x_i$ and $b \leftrightarrow y_j$.

Chapter 7

Results

This chapter offers the performance of march during the two years of its development. Because the improvements where focused on fast performances in general, the results of the SAT 2002 and SAT 2003 competitions are probably the best illustration to the extent we succeeded in realising this goal. Besides that, not a large number of computers was at our disposal to experiment with many solvers and even more benchmarks. The first two upcoming sections present the final results of both competitions.

Last year, large effort was made to integrate equivalence reasoning into march with the intention that it would lead to faster performance in general. Two different versions did arise: One that uses all equivalence reasoning as described in chapter 6, and a modificated version that uses the original CNF-clauses during the lookahead. Together they outperform the existing state-of-the-art SAT solvers on most benchmarks containing equivalence clauses. A table offered in section 7.3 endorses these results.

7.1 Sat 2002 competition

No large scale experiments have been performed with march before the SAT 2002 competition. Therefore, we decided to submit four variants of the solver (marchI, marchIse, marchII and marchIIse) to this competition to obtain some insight of the performance of march on different settings. All four versions performed a partial lookahead on the 12,5 % "best" variables with ranking based on MAR-heuristics. The final branch decision applied a DIFF that counts the newly created binary clauses.

The four versions only differed on two features: The data-structure and the MAR-weight. The two versions marchI used an optimised data-structure for clauses of various length and both versions marchII used data-structures that were optimised for 3-SAT. Consequently, these last two versions used a 3-SAT

Industrial		**Handmade**		**Random**	
solver	#solved	solver	#solved	solver	#solved
berkmin	68	zchaff	35	OKsolver	548
zchaff	64	berkmin	33	marchII	**543**
limmat	54	limmat	29	marchIIse	**543**
2clseq	46	simo	27	2clseq	369
unitwalk	41	OKsolver	23	rb2cl	338
simo	39	2clseq	21	zchaff	305
dlmsat2	37	unitwalk	11	simo	298
dlmsat1	36	dlmsat1	11	berkmin	263
rb2cl	35	marchIIse	**10**	dlmsat1	255
dlmsat3	35	dlmsat3	10	dlmsat2	234
saturn	34	saturn	9	dlmsat3	233
usat10	33	usat05	9	unitwalk	227
usat05	31	usat10	9	limmat	221
OKsolver	30	dlmsat2	9	modoc	219
marchIIse	**26**	marchII	**7**	saturn	208
jquest	24	rb2cl	6	usat10	206
blindsat	20	modoc	5	usat05	205
modoc	20	jquest	4	jquest	122
ga	18	blindsat	0	blindsat	3
marchII	**13**	ga	0	ga	3

Table 7.1 — SAT 2002 competition results

translator. The versions with the se suffix used a procedure that calculated the MAR-weight based on the binary to ternary clause ratio. Both other versions used a MAR-weight equal to 11, which appeared optimal on random instances.

Between the submission deadline and the actual competition, we noticed that our implementation of the calculated weight resulted in an integer. Some experiments showed that theu use of real numbers realised better performances. Therefore, we additionally submitted two versions with this modification. Both these versions participated hors-concours. Both marchI variants were disqualified due to a bug in the pre-processor. They participated also hors-concours.

In the final results of the competition (see table 7.1) march finished second in the random category and poorly in both others. Judging on these results marchIIse was the clear winner of the six versions that participated. However, for each version there were some benchmarks on which it outperformed the others.

7.2 Sat 2003 Competition

Three versions of march (marchsp, marchtt and marchxq) have been submitted to the Sat 2003 competition. These versions were far more optimised than the 2002 versions and contain most elements discussed in this thesis. However, on four topics they significantly differ: (1) The final branch decision applied the elementary Diff; (2) a clause database with variable arrays was used; (3) inactive clauses were not removed from the active data-structure of the lookahead; and (4) only a prototype of the equivalence reasoning was used.

Because the equivalence reasoning was still in a experimental phase, we decided to submit only one version (marchqx) including it. Since no appropriate pre-selection heuristics using equivalence clauses were implemented yet, this version performed a full lookahead. Due to a bug in the equivalence reasoning code, this version was disqualified and participated hors-concours. Both other version differed only slightly: Marchsp selected only the "best" 7% of the variable for the lookahead and adds only a few resolvents during the pre-processing phase, while marchtt performs lookahead on the "best" 10% and adds relatively many resolvents.

The results of the Sat 2003 competition are presented in table 7.2. Due to their small differences, marchsp and marchtt had almost identical performance. On the random benchmarks, march finished third and we solved many more problems than the the winner of the previous year (OKsolver). In the hand-made category march finished fifth, a gain of four places compared to 2002. The results on the industrial benchmarks are still poorly. Lots of progress is required before march can compete in this category.

To give an impression of the performance of the current version of march, a row is added at the bottom of table 7.2. This version applies all equivalence reasoning and all enhancements presented in this thesis. The results offered in this row, show that march significantly increased its performance on the hand-made category and considerably on the industrial instances. Moreover, march now solves the largest number of hand-made benchmarks and the largest total number of problems. Of course, it should be expected that competing solvers have also gained substantially over the past year, judging on the great improvements the Sat solvers showed between their performances during the Sat 2002 and Sat 2003 competition.

	Total		Industrial		Random		Hand-made	
SatSolver	#Series	#Benchs	#Series	#Benchs	#Series	#Benchs	#Series	#Benchs
amvo	2	12	1	2	0	0	1	10
berkmin561	58	255	32	136	10	54	16	65
berkmin62	57	244	32	133	9	51	16	60
farseer	15	48	8	28	1	6	6	14
forklift	62	259	34	143	11	53	17	63
funex	49	217	29	127	5	37	15	53
jerusat1a	54	247	28	119	6	46	20	82
jerusat1b	57	244	28	120	7	43	22	81
jerusat1c	52	222	27	119	6	40	19	63
jquest2	49	196	29	116	4	23	16	57
kcnfs	47	231	6	17	25	163	16	51
limmat	47	190	28	114	5	24	14	52
lsat	33	226	8	46	10	52	15	128
marchsp	**54**	**278**	**17**	**63**	**18**	**131**	**19**	**84**
marchtt	**54**	**278**	**18**	**67**	**18**	**128**	**18**	**83**
oepir	52	229	30	126	5	39	17	64
oksolver	16	96	0	0	16	96	0	0
opensat	35	132	24	94	3	13	8	25
qingting	16	102	4	17	9	73	3	12
satnik	62	268	29	116	13	64	20	88
sato	49	207	25	101	9	54	15	52
saturn	24	178	3	22	14	127	7	29
satzilla	58	293	18	83	19	121	21	89
satzoo0	59	254	31	123	10	49	18	82
satzoo1	63	293	31	124	11	67	21	102
unitwalk	24	174	4	15	15	139	5	20
xqingting	23	109	9	44	4	27	10	38
zchaff	56	235	29	115	10	53	17	67
march	**66**	**376**	**26**	**93**	**19**	**129**	**21**	**154**

Table 7.2 — Sat 2003 comeptition results

7.3 Equivalence reasoning

Four solvers are used to compare the results of march on equivalence reasoning: eqsatz[1], lsat[2], satzoo[3] and zchaff[4]. The choice for eqsatz and lsat is obvious since they are the only other Sat solvers performing equivalence reasoning. Since equivalence clauses merely occur in handmade and industrial problems, we added some solvers that are considered state-of-the-art in these categories: satzoo and zchaff, respectively. All solvers were tested on a AMD 2000+ with 128Mb memory running on Mandrake 9.1. Besides the `parity32` and the `hwb`

[1]version 2.0, available at http://www.laria.u-picardie.fr/∼cli/EnglishPage.html
[2]version 1.1, provided by authors
[3]version 1.02, available at http://www.math.chalmers.se/∼een/Satzoo/
[4]version 2003.07.01, available at http://www.ee.princeton.edu/∼chaff/zchaff.php

benchmarks, we experimented on the `barrel` and `longmult` families that arise from bounded model checking [4], five unsolved benchmarks (`pyhala-braun-x` and `lisa21-99-a`) from the SAT 2002 competition contributed by Pyhala and Aloul, respectively [29] and three factoring problems (`2000009987x`) contributed by Purdom [24].

Two versions of our solver are used to evaluate performance: The first, march$^\circ$ uses the equation $eq_n = 5.5 \times 0.85^n$ to measure the reduction of the CoE during the lookahead, and applies it on the calculation of ACE from the pre-selection heuristics. The second variant, march* does not use the CoE-part during the lookahead but operates using the original CNF instead. Both march variants use a 10 % partial lookahead, since the versions of march without equivalence reasoning achieved on average fast performance by performing only lookahead on the "best" 10 % of the variables. This constant percentage is not optimal in general. It is not even optimal for the benchmarks used in this used in table 7.3. To illustrate the diversity of partial lookahead optima, march$^\circ$ requires 1120 secondes to solve the benchmark provided by Philips using the 10 % setting, while it requires only 761 seconds at the optimal setting of 8 ‰. The performances on this benchmark are offered in the bottom row of table 7.3.

In table 7.3 the performances are presented for these solvers together with five properties of each benchmark:
#Cls refers to the initial number of clauses
#Var refers to the initial number of variables
#Ind refers to the number of variables in the independent set
#Eq refers to the number of detected equivalence clauses.
#Nt refers to the number of non-tautological equivalences after pre-processing.

We conclude that aligning equivalence- and CNF- reasoning as carried out pays off convincingly, but that, although some instances are *not* solved without incorporating the CoE reductions during the lookahead phase (march$^\circ$), others suffer from this additional overhead and are better solved by updating and investigating this CoE part at the chosen path only (march*).

instance	#Cls	#Var	#Ind	#Eq	#Nt	march$^{\circ}$	march*	eqsatz	satzoo	lsat	zchaff
par32-1	10227	3176	157	1158	218	**0.55**	>2000	568.31	>2000	90.85	>2000
par32-2	10253	3176	157	1146	218	**0.3**	>2000	>2000	>2000	88.43	>2000
par32-3	10297	3176	157	1168	218	**1.08**	>2000	>2000	>2000	7.54	>2000
par32-4	10313	3176	157	1176	218	**7.93**	>2000	>2000	>2000	79.87	>2000
par32-5	10325	3176	157	1182	218	**8.82**	>2000	>2000	>2000	34.41	>2000
par32-1-c	5254	1315	157	1158	218	**0.47**	>2000	>2000	>2000	3.91	>2000
par32-2-c	5206	1303	157	1146	218	7.82	>2000	>2000	>2000	**4.45**	>2000
par32-3-c	5294	1325	157	1168	218	**5.06**	>2000	>2000	>2000	33.59	>2000
par32-4-c	5326	1333	157	1176	218	**0.39**	>2000	>2000	>2000	52.39	>2000
par32-5-c	5350	1339	157	1182	218	**6.77**	>2000	>2000	>2000	71.98	>2000
hwb-n20-1	630	134	96	36	35	**18.05**	39.43	78.51	47.04	771.48	300.72
hwb-n20-2	630	134	96	36	35	**23.24**	50.82	83.25	73.38	738.16	461.49
hwb-n20-3	630	134	96	36	35	**16.16**	38.56	75.09	24.37	564.81	257.07
hwb-n22-1	688	144	104	38	37	**68.27**	164.65	299.45	108.	>2000	785.89
hwb-n22-2	688	144	104	38	37	**53.67**	145.51	297.3	85.79	>2000	1097.33
hwb-n22-3	688	144	104	38	37	**58.6**	148.29	306.71	60.6	>2000	1710.10
hwb-n24-1	774	162	116	44	43	**556.25**	796.56	>2000	624.17	>2000	>2000
hwb-n24-2	774	162	116	44	43	**463.46**	832.48	>2000	862.86	>2000	>2000
hwb-n24-3	774	162	116	44	43	**332.**	670.21	>2000	471.73	>2000	>2000
hwb-n26-1	832	172	124	46	45	**1203.99**	>2000	>2000	>2000	>2000	>2000
hwb-n26-2	832	172	124	46	45	**1777.78**	>2000	>2000	>2000	>2000	>2000
hwb-n26-3	832	172	124	46	45	**1703.12**	>2000	>2000	>2000	>2000	>2000
barrel-5	5383	1407	-	720	-	**0.07**	**0.07**	0.15	0.85	1.9	0.52
barrel-6	8931	2306	-	1260	-	**0.15**	**0.15**	0.33	4.94	644.49	2.73
barrel-7	13765	3523	-	2016	-	**0.29**	**0.29**	0.38	23.63	>2000	11.38
barrel-8	20083	5106	-	3024	-	**0.5**	**0.5**	0.61	69.94	>2000	31.22
barrel-9	36606	8903	-	5760	-	**1.19**	**1.19**	1.33	258.37	>2000	240.29
longmult-6	8853	2848	1037	174	90	7.13	3.98	11.96	5.1	66.32	**1.59**
longmult-7	10335	3319	1276	203	105	35.79	21.37	55.15	21.74	109.19	**13.32**
longmult-8	11877	3810	1534	232	120	100.92	70.8	185.85	**66.22**	192.56	68.17
longmult-9	13479	4321	1762	261	135	202.03	**130.46**	347.75	138.8	289.53	131.46
longmult-10	15141	4852	2014	290	150	260.06	**168.21**	520.1	232.96	404.27	252.95
longmult-11	16863	5403	2310	319	165	226.35	**151.36**	662.19	307.62	542.81	344.64
longmult-12	18645	5974	2620	348	180	128.57	**89.37**	741.19	338.48	709.15	305.07
longmult-13	20487	6565	2598	377	195	67.92	**52.69**	855.14	272.67	901.13	255.96
longmult-14	22389	7176	2761	406	210	44.33	**31.78**	985.37	383.08	1112.49	266.54
longmult-15	24351	7807	2784	435	225	25.26	**24.54**	1108.80	215.12	1236.6	207.72
pb-sat-40-4-03	31795	9638	2860	3002	3001	>2000	510.51	>2000	**184.21**	>2000	357.98
pb-sat-40-4-04	31795	9638	2860	2936	2935	>2000	**600.41**	>2000	>2000	>2000	>2000
pb-unsat-35-4-03	24320	7383	2132	2220	2219	771.24	**698.22**	>2000	>2000	>2000	>2000
pb-unsat-35-4-04	24320	7383	2131	2277	2276	821.43	**736.31**	>2000	>2000	>2000	>2000
lisa21-99-a	7967	1453	1310	460	459	**21.26**	1170.12	>2000	>2000	>2000	>2000
2000009987fw	12719	3214	1615	1358	1319	175.68	**115.89**	521.47	267.81	181.86	116.4
2000009987nc	10516	2710	1303	1286	1262	137.41	**84.16**	197.27	167.4	159.55	94.61
2000009987nw	11191	2827	1342	1322	1299	135.25	**87.9**	157.03	218.09	166.55	104.84
philips	4456	3642	1005	342	224	1120.61	**1032.82**	3390.59	2114.64	1277.68	>3600

Table 7.3 — Performances of the solvers march, eqsatz, satzoo, lsat and zchaff in seconds on various benchmarks with equivalence clauses.

Chapter 8

Conclusion and Future Work

Based on the results in chapter 7, we can conclude that march is able to perform fast on a wide range of problems. Four of its features appear to be essential for this performance:

- A fast partial lookahead procedure by optimised data-structures, a reduction of redundant operations and effective pre-selection heuristics.
- A balanced addition of mainly binary resolvents reduces redundancy of lookahead operations. It also results in more accurate pre-selection heuristics and an increased and faster detection of forced variables.
- Cheap and powerful equivalence reasoning requires only extensive computation during pre-processing. This makes the presented additional reasoning ideal for integration in all kinds of Sat solvers.
- Our focus on fast general performance resulted in procedures and heuristics with broad applications.

Below, a brief description is offered of how these features were integrated in march and in what way they may be extended and enhanced in the future.

8.1 Fast partial lookahead

Pre-selection heuristics are probably the most obvious method to reduce computational costs of the lookahead procedure. The pre-selection heuristics presented in paragraph 4.1.2 appear to be useful for this and other purposes. However, other enhancements resulted in more considerable gains, while these do not alter the essentials of the lookahead procedure: They do not change the pre-selected set, nor the number of detected forced variables or the effectiveness of the final branch decision. The six most important enhancements are:

- Reduced memory usage by storing binary clauses in implication arrays, which reduces the number of cache misses.
- Application of IFIUP contributes to a faster detection of conflicts. IFIUP is a cheap procedure to accurately calculate the Diff.
- Application of the TimeAssignment structure results in a lookahead procedure that does not require a costly backtrack procedure.
- The lookahead procedure deals with fewer instructions, due to transportation of calculations from the lookahead phase to the chosen path.
- The local-literal storage data-structure reduces the costs of the most expensive sub-procedure in the lookahead phase.
- Performing a tree-based lookahead results a substantial speed-up: It significantly reduces the number of redundant operations.

After these enhancements, the lookahead phase became considerably faster compared to the version we submitted to the Sat 2002 competition. On some industrial and handmade benchmarks, the PartialLookahead procedure requires to 200 times less computational time compared to the corresponding procedure of the previous version. This results in a balanced lookahead solver. The procedure that forms the bottleneck of computation depends on the benchmarks: The lookahead procedure often remains the bottleneck, but on hard instances with only a few variables (< 500), the chosen path procedures can become the bottleneck. On benchmarks with many initial binary clauses ($>$ 100.000), pre-selection heuristics could become the bottleneck of computation.

The most important work regarding the lookahead has already been done. Still, there is much to gain. Future enhancements of the lookahead procedure will focus on two subjects: First, we experienced that the speed-up by the tree-based lookahead was extended when all variables were sorted before entering this procedure. This sorting was based on ranking the variables by pre-selection heuristics. Since sorting apparently has a strong influence on the reduction of computation, it would be fruitful to investigate which kinds of sorting procedures will result in optimal performance. After all, there is no reason to assume that sorting by ranking is the optimal procedure. Therefore, an additional performance gain is expected with a more effective sorting algorithm.

The second subject of research will focus on heuristics that determine the optimal number of variables to be examined during PartialLookahead. As stated in paragraph 4.1.2, it appears that on average, selection the 10% “best” variables results in relatively optimal performance when using pre-selection heuristics and when optimization is performed with a single percentage on all benchmarks. However, if one optimizes the number of variables for each individual benchmark, some show optimal performance with 1%, others with 10%, and some even with 100% of the variables. Developing heuristics to “estimate” the optimal percentage of a benchmark could result in additional speed-up.

8.2 Addition of resolvents

Sections 3.4 and 4.2 offers two debates regarding the addition of resolvents. The outcome favors a small addition of both binary and ternary resolvents during pre-processing, and the addition of all detected constraint resolvents during the solving phase along with the chosen path compensation resolvents. In the current version of march, this results in a balanced addition between the intrinsic slowdown as a result of the addition of resolvents on the one hand, and on the other in the four most important advantages: (1) More effective pre-selection heuristics; (2) increased detection of necessary assignments; (3) reduced redundancy of lookahead operations by the tree-based lookahead; and (4) increased and faster detection of failed literals.

We consider it a progression when a smaller addition of resolvents produces the same advantages as the current addition. There are two ways to reduce the addition: First, some constraint resolvents do not contribute to an increased and faster detection of failed literals. They merely increase the computational costs and the auto-suggestion effect of the pre-selection heuristics:

Suppose that IFIUP($\mathcal{F} \cup \{x_i\}$) results in the addition of constraint resolvent $\neg x_i \vee y$. If the implication $x_j \rightarrow x_i$ exits, then IFIUP($\mathcal{F} \cup \{x_j\}$) would also result in the addition of constraint resolvent $\neg x_j \vee y$, when performed prior to the lookahead on x_i. Changing the sequence of the lookaheads; first on x_i and then on x_j, will not result in the detection of constraint resolvent $\neg x_j \vee y$: Variable y is already fixed during the first call of RIP(x_j) in IFIUP($\mathcal{F} \cup \{x_j\}$). If a cheap procedure could be developed to remove constraint resolvents of the type $\neg x_j \vee y$, the reduced auto-suggestion effect of the pre-selection heuristics could result in a speed-up .

Second, if we would be able to remove - or largely reduce - the auto-suggestion effect of the pre-selection heuristics, the most important motivation for adding compensation resolvents is removed. Therefore, fewer of these resolvents could then be added while the expected performance remains the same. With this idea in mind we experimented with an alternative DIFF approximation function DAF_{rec}. The general principle of DAF_{rec} is described in equation (8.2). Since cycles may occur using binary clauses, the actual calculation is performed as in algorithm 8.1 and 8.2.

$$\text{DAF}(x) = occ_3(\neg x) + \sum_{\neg x \vee y_i \in \mathcal{F}} occ_3(\neg y_i) \tag{8.1}$$

$$\text{DAF}_{\text{rec}}(x) = occ_3(\neg x) + \sum_{\neg x \vee y_i \in \mathcal{F}} \text{DAF}_{\text{rec}}(\neg y_i) \tag{8.2}$$

DAF_{rec} will probably result in a more accurate approximation if just a few resolvents are added and if a small redundancy exists within the original set of binary clauses. Yet, the actual implementation of DAF_{rec} did not result in faster performances during experiments on various benchmarks, either with or without the addition of chosen path compensation resolvents.

Algorithm 8.1 CALCULATEDAF( )

```
for each free variable x_i in F do
  if VisitedYet[x_i] = false then
    DAFREC(x_i)
  end if
  if VisitedYet[¬x_i] = false then
    DAFREC(¬x_i)
  end if
end for
```

Algorithm 8.2 DAFREC(x)

```
VisitedYet[x] := true
DAF_rec(x) := occ_3(¬x)
for each binary clause x ∨ y_i in F do
  if VisitedYet[y_i] = false then
    DAFREC(y_i)
  end if
  DAF_rec(x) += DAF_rec(¬y_i)
end for
```

This is not surprising, since all redundant constraint resolvents are still present in the formula. These need to be removed before serious experiments with DAF_{rec} could be performed. Adding less resolvents may produce a faster solver, but note that this may also change the optimal parameters of the complex DIFF: The optimal value of magic constant α in equation (4.5) is expected to be higher due to the smaller number of resolvents.

8.3 Equivalence reasoning

We presented a new alignment between equivalence reasoning and lookahead in a DPLL SAT solver. As a result, the solver outperforms existing techniques on benchmarks that contain a significant part of equivalence clauses. Two main features appeared sufficient for successful equivalence reasoning:

- an effective solving procedure of the CoE during the pre-processing.
- a combined lookahead evaluation function to measure the reduction of equivalence clauses.

Additional features of equivalence reasoning may boost performance further, but substantial gains have not been noticed yet. However, two procedures are worth mentioning: First, integration of the combined lookahead evaluation function into the pre-selection heuristics produced a performance gain up to 30%. Second, preventing equivalent variables to enter the lookahead phase produced a small speed-up on practically all benchmarks - with and without equivalence clauses.

Further research on additional features of equivalence reasoning does not seem very fruitful. The reason is twofold: First, the independent set and the combined lookahead evaluation have a far greater impact on the performance. Therefore, research on these features may produce more results. Second, benchmarks without equivalence clauses will probably slow down as a result of additional features. This would reduce the general performance of march.

Two subjects will have our attention in future research on equivalence reasoning: First, as stated in paragraph 6.2.3, the choice of the independent set has a large influence on the eventual performance of the solver. The independent set, selected by our pre-processor, results in a considerably faster performance compared to using the independent set selected by [30]. However, experiments showed that, if the pre-processor algorithm loops randomly though the equivalence clauses, the selected independent set varies heavily, as does the performance of the solver. This is far less using our independent sets, but further research on the properties of those independent sets that result in optimal performance, may enhance our pre-processor to the extend that the selected independent set will approximate optimal performance.

Although our weight function to measure the reduction of equivalences of various length shows fast performances on many benchmarks with a significant part of equivalences, a version of march with used the orinigal CNF-clauses during the lookahead produced in some cases an even faster performance (see table 7.3). Further research in this field is needed to obtain insight in this field. Therefore, we plan to construct a variety of benchmarks containing a significant equivalence part. Then we will examine which weights will generally result in even faster performances.

8.4 General purpose

Comparison between the competition results of Sat 2002 (table 7.1) and Sat 2003 (table 7.2) shows that march has indeed developed towards a general purpose solver. Besides the good results in the random category, march shows a clear gain in the handmade benchmarks. Especially if one considers the fast performance of the current version, presented in the last row of table 7.2. Here, march clearly solves the most problems of the Sat 2003 competition benchmark set.

However, the position in the industrial category was and still is disappointing. With the latest data-structure modifications march was able to solve a few additional problems, but still far too few in comparison with most conflict driven solvers. Since further optimizations and enhancements of the data-structures will result in only a modest performance gain, success in this category requires at least additional reasoning. Attempts to develop a lookahead solver which performs fast on industrial benchmarks, will be a challenging task for future research.

Bibliography

[1] D. Mitchel, B.Selmon and H. Levesque, *Hard and easy distributions of SAT problems*. Proceedings of AIII-1992, pages 459-465, San Jose, CA, July 1992.

[2] F. Bacchus, *Enhancing Davis Putnam with Extended Binary Clause Reasoning*. AAAI/IAAI 2002: 613-619, 2002.

[3] D. Le Berre and L. Simon, *The essentials of the SAT'03 Competition*. Springer-Verlag, LNCS 2919, pages 452-467, 2004.

[4] A. Biere, A. Cimatti, E.M. Clarke, Y. Zhu, *Symbolic model checking without BDDs*. in Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems, Vol. 1579 of Lecture Notes in Comput. Sci., pp. 193-207, 1999.

[5] C. Coarfa, D.D. Demopoulos, A. Aguitte, D, Subramanian and M.Y. Vardi, *Random 3-SAT: The Plot Thickens*. in Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Proceedings, Springer LNCS 1894, pp. 143-159, 2000.

[6] S.A. Cook, *The Complexity of Theorem Proving Procedures*. In Proceedings of 3rd ACM Sympoium on Theory of Computing, pp. 151-158, Ohio, 1971.

[7] J.M. Crawford, M.J. Kearns, R.E.Schapire, *The Minimal Disagreement parity problem as a hard satisfiability problem*. Draft version 1995.

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

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

[10] Freeman J. W., Improvements to Propositional Satisfiabiliy Search Algorithms. Ph.D. thesis, Department of computer and Information science, Univ. of Pennsylvania, Philadepia, PA, 1995

[11] I.P. Gent and T. Walsh, *The Search for Satisfaction*. Department of Computer Science University of Strathclyde, Scotland, 1999.

[12] I.P. Gent and P. Prosser, *SAT encodings of the stable marriage problem with ties and incomplete lists*. Conference proceedings of Fifth International symposium on Theory ans Applications of Staisfiability Testing, pp. 133-140, May 2002

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

[14] D. S. Johnson and M. A. Trick, *Second DIMACS implementation challenge*. In DIMACS benchmarks available at ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/ , 1993.

[15] O. Kullmann, *Investigating the behaviour of a SAT solver on random formulas*. Submitted to Annals of Mathematics and Artificial Intelligence, September, 2002.

[16] O. Kullmann, *OKgenerator homepage*. http://cs-svr1.swan.ac.uk/ csoliver/OKgenerator.html

[17] C.M. Li and Anbulagan, *Look-Ahead versus Look-Back for Satisfiability Problems*. Springer-Verlag, LNCS 1330, pages 342-356, Autriche, 1997.

[18] C.M. Li, *Integrating Equivalency Reasoning into a Davis-Putman Procedure*. In Proceedings of the AIII-2000, pages 291-296, Austin, USA, August 2000.

[19] C. M. Li, *Equivalent literal propagation in Davis-Putnam procedure*. In Discrete Applied Mathematics, Vol 130/2 pp 251-276, 2003.

[20] H. van Maaren, J.P. Warners, *Solving Satisfiability Problems Using Elliptic Approximations. A Note on Volumes and Weights*. Annals of Mathematics and Artificial Intelligence 37, 273-283, 2003.

[21] H. van Maaren, T. van der Velden, *Stabilization of parameter based look-ahead Sat solvers*. To be published.

[22] J. Marques-Silve and I. Lynce, *Complete unresticted backtracking algorithms for Satisfiability*. Conference proceedings of Fifth International symposium on Theory ans Applications of Staisfiability Testing, pp. 214-221, May 2002

[23] R. Ostrowski, E. Gregoire, B. Mazure, L. Sais, *Recovering and exploiting structural knowledge from CNF formulas*, in Proc. of the Eighth International Conference on Principles and Practice of Constraint Programming, Vol. 2470 of Lecture Notes in Comput. Sci., Springer Verlag, pp 185-199, Ithaca (N.Y.), Septembre 2002.

[24] P. Purdom and A. Sabry, *CNF Generator for Factoring Problems*. http://www.cs.indiana.edu/cgi-pub/sabry/cnf.html

[25] L. Simon, *Sat'03 competition homepage*. http://www.lri.fr/ simon/contest03/results/

[26] N. Een, *Satzoo home page*. http://www.cs.chalmers.se/∼een/Satzoo/

[27] B. Selman, D. Mitchell and H.J. Levesque, *Generating Hard satisfiability Problems*. Artificial Intelligence, Volume 81(1-2), pages 17-29, March 1996.

[28] B. Selman, H. Kautz and D. McAllester, *Ten Challenges in Propositional Reasoning and Search*. Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97), 1997.

[29] L. Simon, D. Le Berre, and E. Hirsch, *The sat2002 competition*. Submitted to AMAI, August 2002.

[30] J.P. Warners and H. van Maaren, *A two phase algorithm for solving a class of hard satisfiability problems*. Operations research letters 23, 81-88, 1998.

[31] L. Zhang, *Zchaff sat solver*. http://www.ee.princeton.edu/∼chaff/zchaff.php 2000.

[32] S.A. Cook and D.G. Mitchel, *Finding hard instances of the satisfiability problem: A survey*. DIMACS series volume 5, 1997.

[33] L. Zhang and S. Malik, *Cache Performance of SAT Solvers: A case Study for Efficient Implementation of Algorithms*. Springer-Verlag, LNCS 2919, pages 287-298, 2004.

[34] H. Zhang and M. E. Stickel, *Implementing the Davis-Putnam Method*. SAT2000, pp. 309-326, 2000