SMART SOLVING

Tools and techniques for satisfiability solvers

Marijn J.H. Heule

Cover illustration: Arjen van Lith

SMART SOLVING

Tools and techniques
for satisfiability solvers

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College van Promoties,
in het openbaar te verdedigen op dinsdag 25 maart 2008 om 15:00 uur
door Marienus Johannes Hendrikus HEULE
informatica ingenieur
geboren te Rijnsburg

Published and distributed by: Marijn J.H. Heule
E-mail: marijn@heule.nl

ISBN: 978-90-9022877-8

Keywords: satisfiability, look-ahead solver architecture, adaptive heuristics

NWO
Netherlands Organisation for Scientific Research

This research was supported by the Netherlands Organisation for Scientific Research (NWO) under project number 617.023.306

Contents

If you want to make God laugh,
tell him about your plans.
Woody Allen

1 Introduction

Imagine that you are in a huge mansion owned by a maniac that wants to kill you. Luckily, the maniac is away and you know this is your chance to get out. However, the only way to get out is through the front door which is locked. You decide to search for the key, even though none might be around.

Trapped in this enormous house, with the maniac on his way home, one crucial question arises: How to search? What is the best strategy to find the key as fast as possible? One option is to start with the nearest room. But that one appears to be dark and filled with dead bodies. One thing is sure: By wondering how to search, the key will not be found. You have to decide. Now!

The above is not a script for a new horror B-movie, but a simplified version of the classic video game Maniac Mansion[1.]. It will be used as a framework to illustrate search strategies for the *satisfiability* (SAT) problem. Many other problems[2.] can be translated into SAT and solved by software dedicated to this problem, called a SAT solver. Thanks to the increased strength of SAT solvers, the number of applications which can be effectively solved by them grows every year: For instance class scheduling, hardware and software verification, bounded model checking and many mathematical puzzles.

New techniques to solve the SAT problem – the main subject of this thesis – therefore contribute to solving other problems as well. Due to the simplicity of the SAT problem, these ideas may inspire improvements in other fields - such as the Maniac Mansion example - as well. We will explain these new techniques with some analogies: For example, the problem instance or formula (mansion), a solution for the given problem (key), and the used SAT solver (the character that searches). The story will continue at the beginning of succeeding chapters to illustrate some special search strategies. For instance, what to do or how to exploit the situation in case: You are a nerd (Chapter 3), you can time travel (Chapter 4), you own gadgets (Chapter 5), you have intuition (Chapter 6), or you are not alone (Chapter 7).

[1.] LucasArts, 1987
[2.] All problems in complexity class $\mathcal{NP}$

1.1 The satisfiability problem

The satisfiability (Sat) problem deals with the question whether, given a *formula*, there exists an *assignment* to the *Boolean variables* such that all *clauses* are satisfied. A formula consists of a conjunction of clauses, e.g. $\mathcal{F} = C_1 \wedge C_2 \wedge C_3$ and clauses consist of a disjunction of *literals*. For instance, $C_j = l_1 \vee l_2 \vee l_3$. A literal refers either to a Boolean variable x_i or to its complement $\neg x_i$. Assigning variable x_i to true satisfies the literals x_i, while assigning it to false satisfies the literals $\neg x_i$. A clause is satisfied if at least one of its literals is satisfied.

Example 1.1: Consider the example formula

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

The formula above is easy to solve: For instance, assign variable x_1 to true. Now, the second clause can only be satisfied by assigning x_3 to true. This in turn forces x_2 to be assigned to false. The resulting assignment $x_1 = 1, x_2 = 0$, and $x_3 = 1$, satisfied $\mathcal{F}$, so the formula is satisfiable. The process of assigning forced variables – called *unit propagation* – is widely used in solving Sat problems. The other solution to this formula can be found by assigning x_1 to false.

Although the Sat problem is easily defined and the example formula was not difficult to solve, larger formulae can be very hard to solve in practice. Yet software to solve Sat problems - called Sat solvers - have shown enormous progress in recent years. Chapter 2 will present the state-of-the-art techniques of Sat solving. The succeeding chapters offer our own contributions to the field.

1.2 Motivation

First and foremost, we aimed to develop Sat solving techniques to improve the performance on a vast array of applications. The motivation to focus on the "allround" performance instead of excellence on a specific problem arises from the nature of Sat: The problem is easily defined and explained *and* it is very suitable to solve other problems. This stems for the fact that Sat is $\mathcal{NP}$-complete [Coo71], so all problems in $\mathcal{NP}$ can be transformed into Sat in polynomial time and afterwards solved with a Sat solver.

Yet the transformation of a problem into Sat always results in a loss of problem-specific information. This information may harvest problem-specific techniques that perform better than Sat solvers. Therefore, Sat solving will – in theory – rarely be the fastest technique. In practice, however, Sat solving is an effective method to solve problems like bounded model checking, equivalence checking, combinatorial puzzles, scheduling, etc. Each year the number of applications increases. So, Sat solving is a powerful general solving technique that is competitive regardless its theoretical disadvantage to problem-specific alternatives.

To advance the development of a Sat solver for general purposes, we concentrated on three topics: Efficiency, additional reasoning, and adaptive heuristics.

Efficiency

Efficient SAT solving means in practice efficient unit propagation. A fast implementation of unit propagation is a necessity for a fast SAT solver. In the dominant SAT solver architecture, the conflict-driven SAT solvers, unit propagation accounts for about 80% of the total computational costs - despite all the optimizations [ES03]. For alternative SAT solving architectures such as the LOOKAHEAD and UNITWALK architectures these costs are even much higher.

The fast unit propagation in conflict-driven SAT solvers comes at a price: The used (lazy) data-structures make it hard to extract various types of knowledge from the formula efficiently (e.g. the number of satisfied clauses). Although these data-structures resulted in a significant progress of the field, we predict that they also may hinder future progress, because new reasoning techniques may need this knowledge. Or to quote Donald Knuth[3.]: "Premature optimization is the root of all evil."

Therefore, we studied alternative techniques to reduce the burden of unit propagation. Contributions - mostly implemented in our look-ahead SAT solver march - include other efficient data-structures such as *time stamping* (Section 3.3) and *implication arrays* (Section 3.5), but also eager data-structures such as *tree-based look-ahead* (Section 3.8) and the *removal of inactive clauses* (Section 3.9).

Also, we developed an algorithm to perform unit propagation in parallel: Unit propagation can be viewed as Boolean operations *and* today's 32/64 bit computers can perform 32 or 64 of the familiar Boolean operations simultaneously (in one clock cycle). We capitalized on this by developing a parallel unit propagation algorithm. We implemented it in a UNITWALK based SAT solver called UnitMarch. This solver is the main subject of Chapter 7.

Additional reasoning

More and improved reasoning is probably the key to progress in many areas in computer science. However, as stated above, due to the lazy data-structures used in conflict-driven SAT solvers, various kinds of reasoning are hard to add efficiently.

On the other had, look-ahead SAT solvers already use much heavy computational reasoning within each step of the search process. So, more reasoning can be added cheaply and most knowledge about the formula can easily be obtained. Hence, we focused on the LOOKAHEAD architecture to develop new reasoning techniques.

This thesis presents some new reasoning techniques. Most of them have been implemented in our look-ahead solver march, and have enhanced its performance: *Equivalence reasoning* (Section 3.6), *addition of constraint resolvents* (Section 3.4), *local branching* (Section 4.5), *distribution jumping* (Section 6.3.1), and *detection of autarkies* (Section 7.5.2).

[3.] Structured Programming with go to Statements, ACM Journal Computing Surveys, Vol 6, No. 4, Dec. 1974. p.268.

Adaptive Heuristics

Although SAT solvers are used for a wide range of applications, parameter settings that are optimal for one application may result in poor performance on other applications. Since many users of SAT solvers are unaware of the optimal parameter settings for their specific application, adaptive heuristics which set the parameters such that they yield (near) optimal performance are a fruitful technique. They could improve "allround" performance and make SAT solvers more accessible to users.

This thesis offers two kinds of new adaptive heuristics. First, we discuss an algorithm to determine the number of preselected variables on the fly (Section 4.4.1). Second, Chapter 5 presents adaptive heuristics for the DOUBLELOOK procedure which is used in most look-ahead SAT solvers.

1.3 Thesis Overview

The annual SAT competitions[4.] are an important platform to boost the development of SAT solvers, to measure the progress of the field, and to objectively compare the relative strengths of the participating solvers. Chapter 2 provides an overview of the state-of-the-art techniques used in modern SAT solvers. Also, it offers an "educated guess" where future progress is expected.

Most contributions presented in this thesis are developed and implemented in our own look-ahead SAT solver march. An early version, march_eq (2004) is presented in Chapter 3. This version contains several additional reasoning techniques which have been implemented efficiently. Therefore these techniques are also useful to solve larger structured problems.

An enhanced pre-processor, two kinds of adaptive heuristics and a new branching strategy (a variable selection strategy) are the main improvements to the march_eq solver that resulted in the version march_dl (2005). These features are described in Chapter 4.

The most important new feature of march_dl, an adaptive heuristic for the DOUBLELOOK procedure, is further improved on performance and elegance. The improved heuristics are presented in Chapter 5. Details are provided together with a large-scale study of the proposed heuristics and alternatives.

The main upgrade of the current version, march_ks (2007), is the addition of a feature called distribution jumping. Chapter 6 presents this new technique. Besides a description, it offers an extensive study to the effectiveness of direction heuristics together with tools to measure and compare this effectiveness.

We also developed a local search SAT solver based on the UNITWALK architecture. We parallelized the underlying algorithm on a single processor. The resulting solver called UnitMarch is the subject of Chapter 7.

Finally, in Chapter 8 some conclusions are drawn. It summarizes the contributions and new techniques presented in this thesis and provides a short overview for future work.

[4.] see `http://www.satcompetition.org`.

Even things that are true can be proven.

Oscar Wilde

2 State-of-the-Art SAT Solving

In today's bookstores, it is hard to find a book which is not labeled "The #1 International Best-Seller". Everything is superb, or is least presented as such. Of course, this does not only hold for "literature". Each new SAT solving technique is also presented as "the next big thing". Finding state-of-the-art SAT solving techniques using their descriptive papers is as hard as finding excellent books using their covers.

Yet, the annual SAT competitions [LS03, LS04, LS06] do objectively compare the performances of a wide variety of SAT solvers. The strength of SAT solvers is measured in three categories: Industrial (e.g. verification of hardware and software benchmarks), crafted (e.g. factorization and Latin square problems), and random instances. Each category is split in some divisions. Since these competitions separate the goats from the sheep, they have become very valuable to the SAT community. This chapter will focus on state-of-the-art techniques for SAT solvers / solver architectures, which are dominant in "their" league. These are presented in three sections:

- **Complete SAT solving**: Solvers that can prove both satisfiability and unsatisfiability (the *complete* solvers) are the most successful product in the field. Especially the ones based on lazy data-structures - the *conflict-driven solvers*. These are superior on industrial benchmarks. Their counterparts, the *look-ahead solvers* are strong on random and crafted instances. See Section 2.2.
- **Incomplete SAT solving**: Some problems which cannot be solved by complete solvers within weeks, can be solved by *incomplete* (or local search) solvers instantly. Only satisfiable formulae can be solved by these solvers. This type distinguishes three main architectures that are unbeatable on satisfiable random formulae in particular. See Section 2.3.
- **Representation**: For most users, SAT solving is a *black-box technology*. Therefore, one requires additional techniques - such as an efficient transformation of the original problem and pre-processing - in order to assure that a possible application is encoded in CNF such that solvers will be able to solve the transformed problem effectively. See Section 2.4.

2.1 Preliminaries

Before presenting the state-of-the-art SAT solving techniques, we will introduce, inspired by [Kul08], the basic terminology of the SAT problem (Section 2.1.1) followed by the fundamentals of (mostly complete) SAT solving (Section 2.1.2).

2.1.1 SAT basic terminology

In short, the satisfiability problem deals with the question whether there exists, for a given *formula*, an *assignment* to the *Boolean variables* such that all *clauses* are satisfied. A clause is satisfied if at least one of its *literals* is satisfied and a literal is satisfied if the corresponding variable is assigned to its value. Below we expand on the important terms:

- **Boolean variables and literals**: A Boolean variable x_i (or just variable) can be assigned to the Boolean values `0` or `1`. A literal refers either to x_i or the complement $\neg x_i$. If a variable is assigned to `0` all negative literals are satisfied, while all positive literals are falsified. Otherwise, if assigned to `1` all positive literals are satisfied, while the negative literals are falsified.

- **Clauses**: A clause consists of a disjunction of literals, $C = l_1 \vee l_2 \vee l_3$. Each clause can only be falsified when all its literals get falsified. Clauses with complementary literals (both x_i and $\neg x_i$) can be neglected since they can never be falsified. A clause of size k can be satisfied by $2^k - 1$ different assignments to its literals. There exists one special clause, the empty clause (denoted by $\emptyset$), which is always falsified.

- **Formulae**: SAT formulae are represented in *Conjunctive Normal Form* (CNF): Each formula consists of a conjunction of clauses. For instance, $\mathcal{F} = C_1 \wedge C_2 \wedge C_3$. A formula is *satisfiable* if there exists an assignment satisfying all clauses. If no such assignment exists a formula is called *unsatisfiable*. SAT formulae are not only represented in CNF, SAT solving techniques use this representation, as well: Most data-structures store the formula as a list of clauses. Learned information of various forms is generally stored as clauses, thereby maintaining the CNF representation.

- **Assignments**: An assignment φ is a mapping of the Boolean values `0` and `1` to the variables. An assignment φ applied to a formula $\mathcal{F}$ (denoted by $\varphi \circ \mathcal{F}$) results in a reduced formula $\mathcal{F}'$ where all satisfied clauses and all falsified literals from $\mathcal{F}$ are removed.

  A *satisfying assignment* - which satisfies all clauses - is a certificate that a given formula is satisfiable. Extending a satisfying assignment always yields a satisfying assignment. We refer to a *falsifying assignment* as an assignments that results, when applied to the formula, in the empty clause. A *partial assignment* is an assignment that assigns a Boolean value to a subset of the variables in a formula. Complete SAT solvers mostly work with partial assignments, while incomplete solvers use *full assignments*.

2.1.2 SAT solving terminology

- **Unit propagation**: The most applied technique in SAT solving is probably *unit propagation*. This technique follows directly from the property that clauses can only falsified by one specific assignment to its literals: Once all literals except one are assigned to their opposite value, a clause is called a *unit clause*. Unit clauses can only be satisfied by assigning the remaining literal to its truth value - thereby extending the current assignment. While propagating (assigning) unit clauses, other clauses may become units which in turn must be assigned, too. Propagation stops when either no unit clauses exist in the extended assignment applied to the formula, or when the empty clause is detected. Unit propagation is a confluent technique.

- **DPLL**: Most complete SAT solvers are based on the Davis-Putnam-Logemann-Loveland (in short DPLL) method [DLL62]. This search method starts each step by simplifying the current formula and checks whether it is satisfied (meaning that the original formula is satisfiable), then it selects a decision variable for splitting purposes: Recursively both formulae, the decision variable assigned to 0 or 1, are examined for satisfiability.

  In case a solution is hit, the algorithm stops. Only after all assignments are refuted, the method shows that the formula does not contain a solution and is therefore unsatisfiable. Details about an iterative and recursive implementation are presented in the next section.

- **Decision and implied variables**: The variables that are selected for splitting in the DPLL search-tree are called *decision variables*. Heuristics that determine the decision variables play a crucial role in the performance of SAT solvers.

  Various techniques exist to extend the current assignment (such as unit propagation). Variables that are assigned due to these techniques are called *implied variables*. The main focus of look-ahead SAT solvers is to extend φ as much as possible (by assigning implied variables using additional reasoning techniques) in order to keep the search-tree as small as possible.

- **Relation between clauses and assignments**: A clause C represents a set of falsified assignments, i.e. those assignments that falsify all literals in C. On the other hand, a falsifying assignment φ for a given formula represents a set of clauses that follow from the formula - and thus can be added. An example of such a clause is the one containing literals referring to all the decision variables in φ with the sign such that they are falsified by φ. Adding clauses to $\mathcal{F}$ based on emerging falsifying assignments is an essential technique in conflict-driven SAT solvers.

2.2 Complete Solvers

Complete SAT solvers are based on the DPLL search method [DLL62] (see above). The focus of complete SAT solvers is predominantly on the reduction of the computational effort to search the whole search space - which is required to prove unsatisfiability. Therefore, hitting a solution (satisfying assignment) is mostly a side product of the search. The domain of complete SAT solvers consists of two main types of approaches: The *conflict-driven* architecture and the *look-ahead* architecture.

2.2.1 Conflict-Driven Architecture

Solvers based on the conflict-driven architecture (conflict-driven solvers), reason before backtracking why a dead end situation has been reached and add this information as a clause to the formula. Not much computational effort is spent on the selection of decision literals. This type of solvers is based on the assumption that looking back to where mistakes have been made is more fruitful: Hindsight is always 20/20.

Example 2.1: Consider the example formula below:

$$\mathcal{F} := (x_1 \vee x_4) \wedge (x_3 \vee \neg x_4 \vee \neg x_5) \wedge (\neg x_3 \vee \neg x_2 \vee \neg x_4) \wedge \mathcal{F}_{\text{extra}}$$

The formula consists of clauses with literals $(\neg)x_i$ with $i \in \{1, ..., 5\}$ and some extra clauses (denoted by $\mathcal{F}_{\text{extra}}$) which are not further specified in the example. The example, shown as search-tree in Figure 2.1 (a), starts by selecting $x_5 = 1$ as first decision. This reduces the second clause to a binary clause. Then $x_2 = 1$ is selected as second decision. Now the third clause is reduced to a binary clause. In step three to six, variables are assigned that occur in $\mathcal{F}_{\text{extra}}$ which have no effect on x_i with $i \in \{1, ..., 5\}$. In the seventh step $x_1 = 0$ is selected as decision. This forces x_4 to be assigned to true (due to the first clause). Consequently, x_3 is forced to true (due to the second clause) and to false (due to the third clause) - which yields a contradiction.

Figure 2.1 (b) shows the conflict at depth seven in an *implication graph*. White nodes represent the decisions and black nodes the conflict. At the far right the conflict is shown. Each cut in the implication graph, having the decisions nodes left from the cut and the conflict right from the cut, represents a conflict clause - which can be added to the formula. In the figure the cut closest to the conflict in shown, representing the conflict clause $\neg x_2 \vee \neg x_4 \vee \neg x_5$.

Based on the conflict clause, the backtrack level is determined - which is the depth at which the conflict clause becomes a unit clause. In this case the backtrack level is 2 because at that depth, $x_2 = 0$ and $x_5 = 0$, which reduces the conflict clause to $\neg x_4$. The algorithm jumps back to depth 2 and then forces x_4 to false (due to the conflict clause). This in turn forces x_1 to true (due to the first clause). Now the algorithm continues by selecting a new decision.

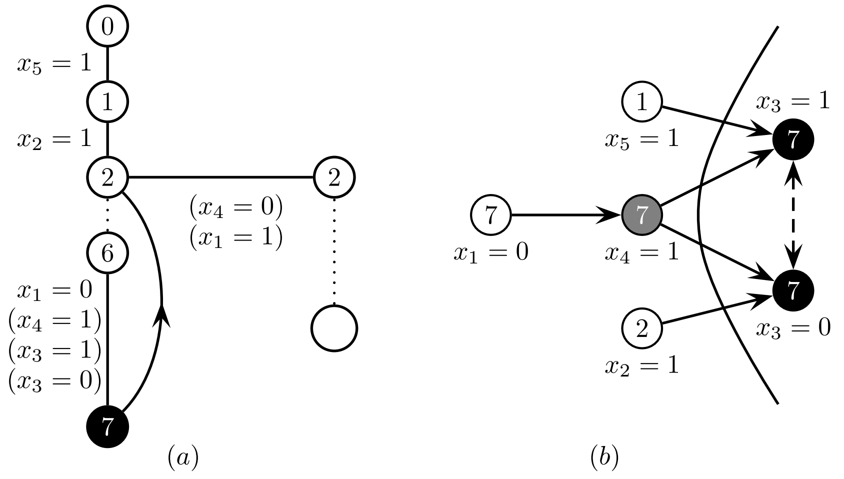

Figure 2.1 — Graphical representations of the running example. The numbers in the nodes refer to the depth. In (*a*) the search-tree is visualized and (*b*) shows the implication graph of the conflict emerged at depth 7. White nodes represent the decisions and black nodes the conflict. The cut shows conflict clause $\neg x_2 \vee \neg x_4 \vee \neg x_5$.

The conflict-driven architecture - based on an iterative version of DPLL (see Algorithm 2.1) - is clearly dominant within the field of complete SAT solvers. Out of all SAT solvers based on this architecture, minisat [ES03] is clearly the most important one[5.]. This section describes in chronological order the techniques that contributed to the success of this architecture.

The first conflict-driven SAT solver, called grasp, was developed by Marques-Silva and Sakallah [MSS96]. The implementation already included many features which are now common in "modern" conflict-driven solvers:

- **Iterative DPLL**: Each iteration of the algorithm starts by selecting an effective decision literal for further reduction. When no literal could be selected, all clauses are satisfied - a solution has been found. Otherwise, the procedure will eventually prove unsatisfiability.

- **Global learning**: If the current assignment results in a *conflicting clause* (all literals are falsified) then the ANALYZECONFLICTS procedure computes a *conflict clause* which is added to the formula. Conflict clauses are constructed in such a way, that after backtracking to the last decision level (or even beyond), the conflict clause is reduced to a unit clause. So checking the simplified formula with the decision literal assigned to false is realized implicitly. If the empty clause is learned, the formula is unsatisfiable.

[5.] based on the recent SAT competitions (see www.satcompetition.org)

- **Backjumping**: Here, the conflict clauses are used to guide backtracking. The algorithm jumps back to the highest depth where the last conflict clause becomes a unit clause. This may involve a jump of multiple levels.

Algorithm 2.1 CONFLICTDRIVEN-ITERATIVE-DPLL($\mathcal{F}$)

```
while TRUE do
    $l_{\text{decision}}$ := GETDECISIONLITERAL( )
    if no $l_{\text{decision}}$ is selected then
        return satisfiable
    end if
    $\mathcal{F}$ := SIMPLIFY( $\mathcal{F}(l_{\text{decision}} \leftarrow 1)$ )
    while $\mathcal{F}$ contains empty clause do
        $C_{\text{conflict}}$ := ANALYZECONFLICTS( )
        if $C_{\text{conflict}}$ is the empty clause then
            return unsatisfiable
        end if
        BACKTRACK( $C_{\text{conflict}}$ )
        $\mathcal{F}$ := SIMPLIFY( $\mathcal{F} \cup C_{\text{conflict}}$ )
    end while
end while
```

Although grasp uses most conflict-driven ingredients, it can hardly be compared with today's conflict-driven solvers in terms of performance: The addition of conflict clauses significantly increases the costs of unit propagation.

zChaff [MMZ$^+$01] tackled this problem and gave the conflict-driven architecture a real boost. Five main techniques contributed to the increased performance:

- **2-literal watch pointers data-structure**: This technique exploits the fact that conflict-driven SAT solvers are not interested in the size of a clause, but they only want to know when a clause becomes unit. As long as there exist two unassigned literals or if one literal is satisfied then a clause is not a unit clause. Instead of pointers to all literals in a clause, only two pointers are stored. These avoid to stay on falsified literals. As soon as one associated literal is assigned to false and the other pointer is not associated with a satisfied literal, the pointer attempts to move to a new not falsified literal. In case none exists, the clause is either a unit clause or it is falsified in all literals. This technique is called a *lazy data-structure*.

- **Variable State Independent Decaying Sum (VSIDS)**: This decision heuristic prefers literals that occur in the most recently used (during the conflict analysis) conflict clauses. Notice that "old" heuristics as *Most Occurrences in clauses of Minimal Size* (MOMS) [Fre95] cannot cheaply be computed using the lazy data-structures.

- **First Unique Implication Point (1-UIP)**: Given a conflicting clause, various conflict clauses could be added. In [ZMMM01], several strategies have been examined, of which 1-UIP appeared the best learning scheme.
- **Restarts**: As observed in [KHR$^+$02], restarting the DPLL procedure while keeping the conflict clauses and using the VSIDS decision heuristic values improves the performance. Restarts, for instance, increase the chance to get lucky by hitting a solution fast. Also, they may yield more effective decision literals by using the updated VSIDS heuristic values.
- **Clause database management**: Conflict clauses which are not used in recent conflict analyses are removed from the clause database to speed-up unit propagation.

Finally, minisat is a SAT solver by Eén and Sörensson [ES03] which features all the above techniques. Most of them are slightly improved and implemented efficiently, resulting in a solver of only 600 lines of code. One technique has been added:

- **Conflict minimization**: This technique attempts to reduce the number of literals in the conflict clause - computed during the conflict analysis - using the existing clauses [ES05].

Future progress. Unit propagation consumes about 80% of the total solving time within minisat [ES03]. Regardless the use of lazy data-structures, the costs of simplifying the formula is enormous for all solvers using this architecture. Therefore, conflict-driven SAT solvers can be considered brute-force solvers: Most computational costs are not spent on reasoning. Other solver architectures show that for some families much additional reasoning is required to solve instances efficiently. This may also hold for benchmarks on which conflict-driven SAT solvers are currently very strong.

Bringing the conflict-driven architecture more into balance might be hard to accomplish because of the deadlock situation between lazy data-structures and additional reasoning: New techniques designed to improve performance may require statistical information about the formula (such as the size of the remaining clauses) which is not available while using lazy data-structures. On the other hand, without these data-structures, the costs of unit propagation will increase substantially. Therefore, new reasoning techniques should compensate for the loss in performance.

2.2.2 Look-ahead architecture

A look-ahead SAT solver's primary focus is to solve a formula by constructing a small and balanced DPLL search-tree through a substantial amount of reasoning and by branching on *effective decision variables*. Effective decision variables are those variables that yield a relatively large reduction of the formula if they are assigned to true *and* if they are assigned to false. This reduction can be approximated using various kinds of heuristics. However, actually performing a *look-ahead* - that is, to assign a variable to a truth value, simplify the formula, and measure the reduction - takes more time, but outperforms these approximation heuristics.

Look-aheads can also be used to reduce the formula (by detecting and assigning forced variables) or to further constrain it (by adding resolvents). For example, if look-ahead on literal x (assigning x to true) results in a conflict, then x is a *failed literal* and therefore variable x is forced to be assigned to false. Also, if look-ahead on x assigns y to true than this information can be stored as a *local learned* binary clause $\neg x \vee y$ (in case the clause is not in the formula).

The look-ahead architecture switches between the DPLL search procedure, which does the global searching, and the LOOKAHEAD procedure, which selects the decision variable in each node and searches for implied variables by additional reasoning. A graphical representation of the architecture is shown in Figure 2.2. In this figure, the variables in the nodes refer to the decision variables (in the DPLL procedure) and to the look-ahead variables (in the LOOKAHEAD procedure). In black nodes a conflict has been detected and therefore they refer to leaf nodes.

Most look-ahead SAT solvers do not use backjumping. So, the DPLL search-tree in the look-ahead architecture is a just a binary search-tree. The LOOKAHEAD procedure first performs many look-aheads and measures the reduction of the formula (caused by these look-aheads). This reduction is often expressed by the sum of newly created (reduced, but not satisfied) clauses.

The look-ahead architecture is usually implemented using a recursion version of the DPLL framework - see Algorithm 2.2. The selection of the decision variable, reduction of the formula, and addition of learned clauses are performed by the LOOKAHEAD procedure. Optionally, the architecture uses a GETDIRECTION procedure to determine which *branch* (reduced formula) should be evaluated first. The preferred truth value (denoted by $\mathbf{B}$) for the decision variable influences the performance on satisfiable instances. Figure 2.2 also shows this choice: In the node with decision variable x_a, the negative branch ($x_a = \mathtt{0}$) is examined before the positive branch ($x_a = \mathtt{1}$), while in the node with decision variable x_b the opposite is chosen.

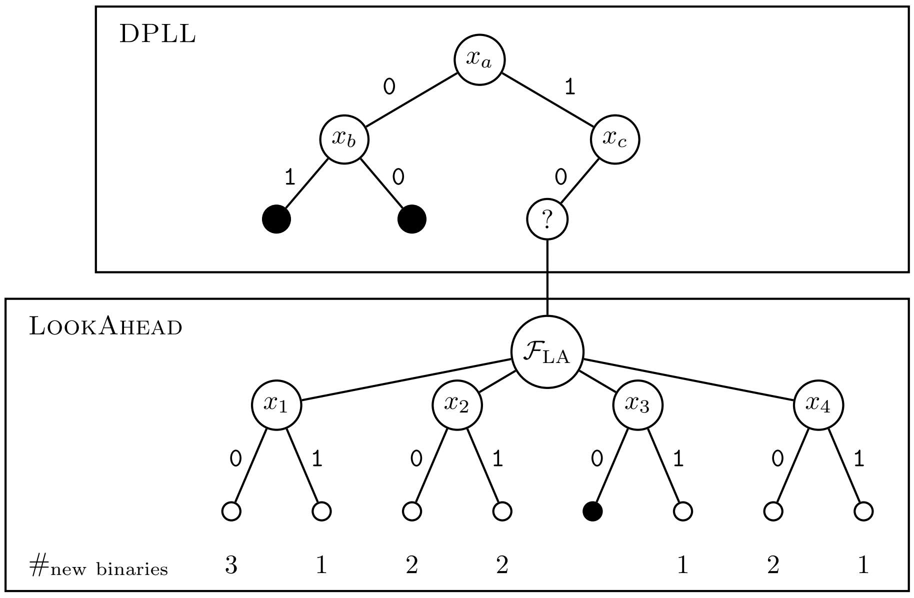

Figure 2.2 — A graphical representation of the look-ahead architecture. Above, the DPLL super-structure (a binary tree) is shown. In each node of the DPLL-tree, the LookAhead procedure is called to select the decision variable and to compute implied variables by additional reasoning. Black nodes refer to leaf nodes and variables shown in the vertices refer to the decision variables and look-ahead variables, respectively.

Example 2.2: Consider the following example formula below:

$$\mathcal{F}_{\text{LA}} = (\neg x_1 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$$

Since the largest clauses in $\mathcal{F}_{\text{LA}}$ have size three, only new binary clauses can be created. For instance, during the look-ahead on $\neg x_1$, three new binary clauses are created (all clauses in which literal x_1 occurs). The look-ahead on x_1 will force x_3 to be assigned to true by unit propagation. This will reduce the last clause to a binary clause, while all other clauses become satisfied. Similarly, we can compute the number of new binary clauses (denoted by $\#_{\text{new binaries}}$) for all look-aheads (see Figure 2.2).

Notice that the look-ahead on $\neg x_3$ results in a conflict. So $\neg x_3$ is a *failed literal* and forces x_3 to be assigned to true. Due to this forced assignment the formula changes. To improve the accuracy of the look-ahead heuristics (in this case the reduction measurement), the look-aheads should be performed again. However, by assigning forced variables, more failed literals might be detected. So, for accuracy, first iteratively perform the look-aheads until no new failed literals are detected.

Algorithm 2.2 LOOKAHEAD-DPLL($\mathcal{F}$)

```
1: $\mathcal{F}$ := SIMPLIFY($\mathcal{F}$)
2: if $\mathcal{F}$ is empty then
3:     return satisfiable
4: else if $\mathcal{F}$ contains empty clause then
5:     return unsatisfiable
6: end if
7: $\langle \mathcal{F}, x_{\text{decision}} \rangle$ := LOOKAHEAD( $\mathcal{F}$ )
8: $\mathbf{B}$ := GETDIRECTION( $x_{\text{decision}}$ )
9: if DPLL( $\mathcal{F}(x_{\text{decision}} \leftarrow \mathbf{B})$ ) = satisfiable then
10:     return satisfiable
11: end if
12: return DPLL( $\mathcal{F}(x_{\text{decision}} \leftarrow \neg\mathbf{B})$ )
```

Finally, the selection of the decision variable is based on the reduction measurements of both the look-ahead on $\neg x_i$ and x_i. Generally, the product is used to combine the numbers. In this example, x_2 would be selected as decision variable, because the product of the reduction measured while performing look-ahead on $\neg x_2$ and x_2 is the greatest (i.e. 4).

Decision heuristics

Decision heuristics predict which free variable is the most effective. These heuristics consist of two parts: A difference heuristic (DIFF) that quantifies the reduction of the formula by a look-ahead, and a MIXDIFF heuristic that combines two DIFF values of look-aheads on complementary literals.

Various DIFF heuristics are used in look-ahead SAT solvers. All these heuristics have three properties in common: (1) All parameters are optimized on random k-SAT formulae; (2) reduction is measured by counting the number of *newly created* (reduced, but not satisfied) clauses in a weighed manner; and (3) newly created clauses of size i are regarded about 5 times as important as those of size $i + 1$ [Kul02].

The most effective DIFF heuristic on random k-SAT formulae is the *backbone search heuristic* developed by Dubois and Dequen [DD03]. In addition to a weight for the size of the new clauses, clauses are weighed based on the number of resolution possibilities of each newly created clause. This heuristic is by far the most costly one. Also, regardless of these costs, on many structured instances cheaper heuristics yield a smaller DPLL search-tree.

In practice, all look-ahead SAT solvers use the same MIXDIFF heuristic. Let L be the measured DIFF of the left (first) branch, and R the one of the right (second) branch. The product (LR) is generally considered to be an effective MIXDIFF heuristic [Fre95]. It attempts to produce a balanced search-tree. Most solvers use the sum ($L + R$) for tie-breaking purposes.

Direction heuristics

Direction heuristics (used for the GETDIRECTION procedure) are in theory very powerful: Perfect direction heuristics will solve all satisfiable formulae in a linear number of decisions. Moreover, existence of perfect direction heuristics (computable in polynomial time) would prove that $\mathcal{P} = \mathcal{NP}$. Contrary to its theoretical potential, it proves hard to develop direction heuristics that significantly reduce the number of decisions in practice.

Two examples: The strongest SAT solver on recent SAT competition, minisat [ES03], always prefers the branch $\mathcal{F}(x_{\text{decision}} \leftarrow 0)$. This direction heuristic appeared "optimal" on many industrial benchmarks. However, this result is likely an artifact of the encoding of these benchmarks.

Second, two main strategies are common for direction heuristics: I) Elect the branch which has the smallest estimated subtree; and II) elect the branch with the highest probability of being satisfiable. In practice, these strategies are complementary: The branch with the smallest estimated subtree is rarely most likely of being the more satisfiable.

Most look-ahead SAT solvers use direction heuristics based on strategy II which results in faster performances on satisfiable random formulae. Yet, on many structured instances strategy I appears more helpful.

Additional Reasoning

Look-ahead on literals which will not result in a conflict appear only useful to determine which variable has the highest decision heuristic value. However, by applying additional reasoning, look-ahead on some literals can also be used to reduce the formula (e.g. failed literals). Look-ahead on the remaining literals can be used to add *resolvents* (learned clauses) to further constrain the formula. For these purposes, three kinds of additional reasoning are used in look-ahead SAT solvers:

- **Local learning**: During the look-ahead on x, other variables y_i can be assigned by unit propagation. Some are a result of the presence of binary clauses $\neg x \vee (\neg)y_i$, called *direct implications*. Variables assigned by other clauses are called *indirect implications*. For those variables y_i that are assigned to true (or false) by a look-ahead on x through indirect implications, a *local learned* binary clause $\neg x \vee y_i$ (or $\neg x \vee \neg y_i$, respectively) can be added. To optimize the positive effect on the performance, only a subset of the local learned clauses should be added [HDvZvM04].

- **Autarky detection**: An *autarky* (or autark assignment) is a partial assignment φ that satisfies all clauses that are "touched" by φ. Hence, all satisfying assignments are autark assignments. Autarkies that do not satisfy all clauses can be used to reduce the size of the formula: Let $\mathcal{F}_{\text{touched}}$ be the clauses in $\mathcal{F}$ that are satisfied by an autarky. The remaining clauses $\mathcal{F}^* := \mathcal{F} \setminus \mathcal{F}_{\text{touched}}$ are satisfiability equivalent with $\mathcal{F}$. So if we detect an autark assignment, we can reduce $\mathcal{F}$ by removing all clauses in $\mathcal{F}_{\text{touched}}$.

In case a look-ahead on x creates only a single new clause (e.g. $y_i \vee y_j$), called an *1-autarky*, then local learned clauses $x \vee \neg y_i$ and $x \vee \neg y_j$ can be added [Kul00]. In other words, if either y_i or y_j is assigned to true, then assigning x to true yields an autarky.

- **Double look-ahead**: If many new binary clauses are created during a look-ahead, the reduced formula is possibly unsatisfiable [Li99]. Unsatisfiability of the reduced formula can be checked using double look-aheads: Additional look-aheads on a second level of propagation (on the resulted formula after a look-ahead). Either double look-aheads detect unsatisfiability of the reduced formula, thereby finding a forced literal, or the information of double look-aheads can be stored as local learned binary clauses.

Eager data-structures

In contrast to the lazy data-structures used in conflict-driven Sat solvers, look-ahead Sat solvers use *eager data-structures*. To accurately measure a Diff heuristic, the exact sizes of all newly created clauses should be computed efficiently - which is not possible using lazy data-structures.

To reduce the number of cache misses while performing look-aheads (which can significantly improve performance), the formula should be stored using as little memory as possible. Storing binary clauses requires only half the memory required to store non-binary clauses [HDvZvM04]. Therefore, it is more efficient to store the formula in separate binary and non-binary data-structures. Also, satisfied clauses should not be processed during a look-ahead. So the data-structures should be designed in such a way that clauses can be removed easily to reduce the required memory while performing look-aheads.

Future progress. Look-ahead Sat solvers are strong on random k-Sat formulae and many structured (crafted) instances. Techniques that work for one class do not necessarily work for the other. Only three techniques significantly improve the performance on random k-Sat instances: (1) An effective decision heuristic; (2) detection of failed literals (enhanced with double look-aheads); and (3) restriction of the look-ahead variables. Other techniques do not reduce the computational time by more than 10%. For the last four years no clear gains have been reported on hard random k-Sat formulae. Therefore, there are no high expectations of progress in the short term.

However, on structured instances, quite some progress has been made. Additional reasoning substantially boosts performance. For instance, many structured benchmarks are solved much faster (some up to 100 times) by adding a specific form of local learned clauses [HDvZvM04]. Yet, on random k-Sat formulae only about 5% can be gained. The same holds for the use of eager data-structures. By adding more reasoning and extending the power of eager data-structures, even more progress of look-ahead Sat solvers on structured benchmarks can be established.

2.2.3 Domain of application

Conflict-driven SAT solvers focus on fast performances on industrial benchmarks [ES03], while look-ahead SAT solvers are traditionally optimized on random k-SAT formulae [DD03, Li99], especially on the unsatisfiable instances.

In practice, look-ahead SAT solvers are strong on benchmarks where either the *density* (ratio clauses to variables) or the *diameter* (longest shortest path in the resolution graph[6.], for instance) is small [Her06]. On the other hand, conflict-driven solvers outperform look-ahead solvers on benchmarks with either a large diameter or high density.

Figure 2.3 illustrates this. Using the structured (crafted and industrial) benchmarks from the SAT 2005 competition and SATlib, the relative performance is measured of the solvers minisat (conflict-driven) and march (look-ahead). For a given combination of density and diameter of the resolution graph, the strongest solver is plotted.

Notice that Figure 2.3 compares minisat and march and therefore it should not be interpreted blindly as a comparison between conflict-driven and look-ahead SAT solvers in general. The solver march is the only look-ahead SAT solver that is optimized for large and structured benchmarks. Selecting a different look-ahead SAT solver would change the picture in favor of minisat.

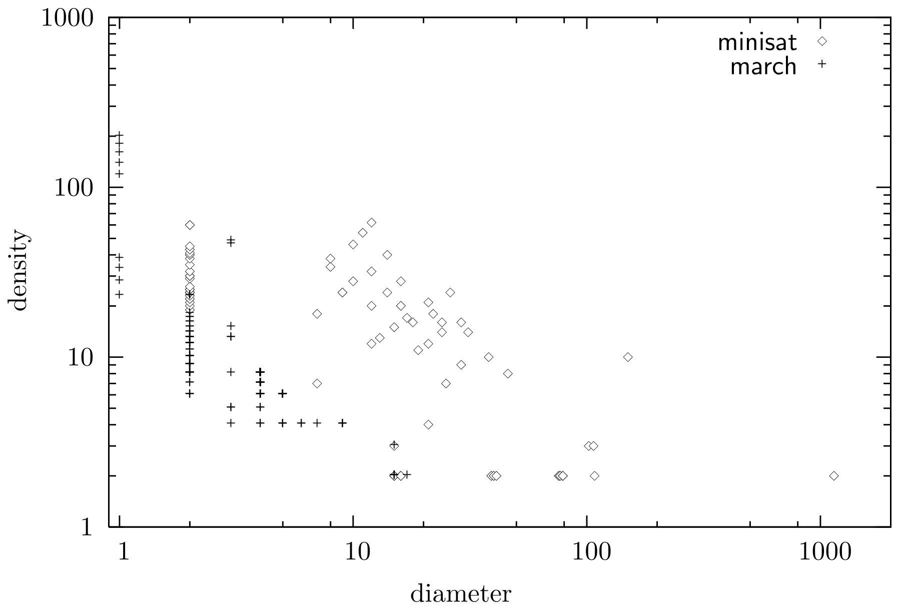

Figure 2.3 — Strongest architecture on structured benchmarks split by density and diameter. Architectures represented by minisat (conflict-driven) and march (look-ahead).

[6.] The resolution graph is a clause-based graph. Its vertices are the clauses and clauses are connected if they have exactly one clashing literal.

The almost clear separation shown in Figure 2.3 can be explained as follows: The density expresses the cost of unit propagation. The higher the density, the more clauses need to be reduced for each assigned variable. Look-ahead becomes very expensive for formulae with high densities, while using lazy data-structures, the cost of unit propagation does not increase heavily on these formulae.

The diameter expresses the global connectivity of the clauses. The larger the diameter, the more multiple local clusters occur within the formula. Local clusters reduce the effectiveness of reasoning by look-ahead SAT solvers: Assigning decision variables to a truth value will modify the formula only locally. Therefore, expensive reasoning is only expected to learn facts within the last modified cluster. On the other hand, conflict-driven solvers profit from local clusters: Conflict clauses will arise from local conflicts and therefore they will be reused frequently.

2.3 Incomplete Solvers

Most incomplete SAT solvers are local search solvers. They can be divided into two architectures: *Stochastic local search* solvers and *unit propagation local search* solvers. There is also a third incomplete SAT solver architecture, called *survey propagation*. It has been developed by physicists and does not use local search [BMZ05]. Currently, it only performs well on a specific type of benchmarks: Random 3-SAT formulae with a huge number of variables. Since it yet has to prove itself in true competition, it is not discussed here in detail.

The generic structure of local search SAT solvers is shown in Algorithm 2.3: An initial random assignment is attempted to be modified in such a way that it satisfies the given formula. The structure contains two parameters: 1) `MAX_TRIES` to make sure that the algorithm will eventually terminate and 2) `MAX_STEPS` to guarantee that a "poor" algorithm can jump out of a local minimum.

Algorithm 2.3 Generic structure of local search SAT solvers.

```
procedure SOLVE( $\mathcal{F}$ )
    for $i$ in 1 to MAX_TRIES do
        $\varphi$ := random initial assignment
        for $j$ in 1 to MAX_STEPS do
            if $\varphi$ satisfies $\mathcal{F}$ then
                return satisfiable
            end if
            $\varphi$ := FLIP( $\varphi$ )
        end for
    end for
    return unknown
end procedure
```

2.3.1 Stochastic Local Search

Mainstream local search solvers are the so-called *stochastic local search* (SLS) solvers. They dominate the random SAT division of the SAT competitions. The first SLS SAT solver was gsat by Selman *et al.* [SLM92]. Gsat uses an initiative global heuristic to improve the assignment: In each step the variable is flipped which mostly increases the number of satisfied clauses.

Later, SLS SAT solvers attempted to improve the assignment using a more local heuristic. The WALKSAT algorithm [SKC94] (see Algorithm 2.4) is the most frequently used architecture. It starts each step by selecting a random falsified clause C in the current assignment φ applied to $\mathcal{F}$ (denoted by $\varphi \circ \mathcal{F}$). C is satisfied by one of three possible flips:

- **Free flip**: If the assignment on a variable in C can be flipped so that no clause becomes unsatisfied, this flip is regarded as *free*. Such a flip is always preferred by the algorithm.
- **Random walk**: To prevent the algorithm to get stuck in a local minimum, the assignment on a random variable in C is flipped with probability p - known as the *noise* setting.
- **Heuristic flip**: With probability $1 - p$ the "optimal" variable is selected to be flipped in the assignment. Optimality is based on a heuristic. In general, the variable that mostly improves the number of satisfied literals is selected. Also, the recent history of flipped variables could be taken into account as in novelty [MSK97].

Algorithm 2.4 FLIP_WALKSAT(φ)

```
1: $C$ := random falsified clause by $\varphi \circ \mathcal{F}$
2: if  a variable $\in C$ can be flipped for free then
3:     flip in $\varphi$ that variable
4: else
5:     flip in $\varphi$ with probability $p$ a random variable $\in C$
6:     flip in $\varphi$ with probability $1 - p$ the "optimal" variable of $\in C$
7: end if
8: return $\varphi$
```

Crucial to fast performance of WALKSAT-based solvers is an optimal setting for the noise parameter p. However, the optimum is different for each family of benchmarks. Optimal values range from about 0.2 to 0.8. The AutoWalkSAT solver estimates the optimal value as a pre-processing step [PK01].

Hoos [Hoo02] proposes to use a more dynamic heuristic which aims to adapt p towards the optimal value while solving a formula. After each $m\theta$ steps (with m referring to the number of clauses and $\theta = \frac{1}{6}$), it either increases or decreases p depending on the progress made, in terms of the number of satisfied clauses.

$$\textsc{NoiseDecrease}: \quad p := p + (1-p)\phi \qquad \lfloor 2.1$$
$$\textsc{NoiseIncrease}: \quad p := p - 0.5\,p\,\phi \qquad \lfloor 2.2$$

Although the adaptive noise algorithm uses three magic constants[7.] to replace a single one, the performance is generally comparable with the optimal static heuristic. Moreover, on some instances adaptive noise outperforms optimal static noise. Apparently, while solving, different noise settings are optimal during different phases.

Future progress. Recent state-of-the-art SLS SAT solvers are no longer implemented using the pure WALKSAT algorithm. The framework is extended by adding for example global / greedy flips (flipping the variable that results in the largest increase of satisfied clauses). The fast solver $\mathsf{g^2wsat}$ is based on this combination [LH05]. Another spin-off is the addition of resolvents as in $\mathsf{R^+AdaptNovelty^+}$ [APSS05] - the winner of the random SAT division during the SAT 2005 competition. Future SLS solvers will probably consist of a bricolage of many effective techniques.

Currently, complete SAT solvers are stronger on many satisfiable structured problems. Yet, there are many recent developments in the field of SLS SAT solvers. Future progress will depend on the success of these new techniques to make SLS solvers competitive on these formulae, as well.

2.3.2 Unit Propagation Local Search

The UNITWALK algorithm (see Algorithm 2.5) flips variables using unit propagation. During each step (called a *period*), the current assignment (called φ_{master}) is modified as follows: An ordering π of variables is computed, and an empty assignment φ_{active} is created. φ_{active} is filled in such a way that it satisfies the unit clauses in $\varphi_{\text{active}} \circ \mathcal{F}$. If there are no unit clauses left in $\varphi_{\text{active}} \circ \mathcal{F}$, φ_{active} is extended by assigning the next free variable in π according to its truth value in φ_{master}. A period ends when all variables are assigned in φ_{active}. Then, the current assignment is replaced by φ_{active}.

The UnitWalk SAT solver based on the UNITWALK algorithm won the random SAT division during the SAT 2003 competition[8.]. UnitWalk is also strong on bounded model checking benchmarks [VB03].

[7.] In [Hoo02] only two parameters (θ and ϕ) are discussed, so the 0.5 in NOISEINCREASE is not regarded as a parameter. The proposed settings are $\theta = \frac{1}{6}$ and $\phi = \frac{1}{5}$.

[8.] The version of UnitWalk which participated during the SAT 2003 competition was a hybrid solver that switches between the UNITWALK and WALKSAT algorithm from time to time. Since no WALKSAT-based solver participated during that competition, it is not clear which algorithm contributed most to this success. However, on the random k-SAT formulae - which are a substantial part of the random category - the WALKSAT algorithm is stronger.

Algorithm 2.5 FLIP_UNITWALK(φ_{master})

```
1: $\pi$ := random order of the variables
2: $\varphi_{\text{active}} := \{x_i = *\}$
3: for $i$ in 1 to $n$ do
4:     while unit clause $u \in \varphi_{\text{active}} \circ \mathcal{F}$ do
5:         $\varphi_{\text{active}}[\ VAR(u)\ ] := TRUTH(u)$
6:     end while
7:     if variable $x_{\pi(i)}$ is not assigned in $\varphi_{\text{active}}$ then
8:         $\varphi_{\text{active}}[\ x_{\pi(i)}\ ] := \varphi_{\text{master}}[\ x_{\pi(i)}\ ]$
9:     end if
10: end for
11: if $\varphi_{\text{active}} = \varphi_{\text{master}}$ then
12:     random flip variable in $\varphi_{\text{active}}$
13: end if
14: return $\varphi_{\text{active}}$
```

Future progress. The UNITWALK algorithm has not yet received much attention of the SAT community. Currently, only the UnitWalk solver is based on this algorithm. Notice that no heuristics are used. In all likelihood, there is quite some room for improvements. Judging from the performance of UnitWalk, this deserves serious research. Especially considering that UnitWalk outperforms other local search SAT solvers on benchmarks with many binary clauses - which is the case in most structured problems.

2.4 Representation

Current state-of-the-art SAT solvers can quickly solve enormous formulae with millions of clauses. Most of these instances arise from problems that are naturally encoded as SAT problems - such as electronic circuit verification [VB03]. Also, SAT solvers outperform alternative techniques on problems that are originally encoded as SAT - such as random k-SAT formulae.

On the other hand, some easy problems are impossible to solve. For instance, is it possible to put n pigeons in $n-1$ holes so that each hole contains at most one pigeon? Clearly, the answer is NO. However, SAT solving techniques have severe difficulties solving any encoding of this problem (say for $n > 15$). It requires a higher level of representation (such as *cardinality solving* [vL06]) to tackle this problem efficiently.

The important difference is the SAT and cardinality representation of the constraints forbidding a pigeon to be in more than one hole. In general, the translation to CNF uses binary clauses of type $\neg p_{i,j} \vee \neg p_{i,j+1}$ (if pigeon i is in hole j it cannot be in hole $j+1$). The size of a resolution proof of such an translation is at least exponential in the number of pigeons [BIK+92]. On the other hand, if these constraints are encoded as a cardinality constraints (i.e. $\sum_j \neg p_{i,j} \leq 1$), it is possible to construct a cardinality resolution proof which size is only quadratic in the number of pigeons [vL06].

Many problems lie somewhere in between: Although they could not be transformed naturally to SAT, SAT solvers can effectively solve them, given that they are properly encoded. We discuss two aspects of encoding: First, problems can be transformed in numerous ways into SAT, but straightforward transformations are rarely optimal. Second, even with an "optimal" transformation, some pre-processing techniques may be required.

2.4.1 Transformation

The transformation of a problem from its original representation to *Conjunctive Normal Form* (CNF) could be realized in many ways. An efficient transformation could make the difference whether or not SAT solving is an effective technique to tackle the problem.

A straightforward transformation is generally not the most efficient one. Consider for instance the constraint `AtMostOne`$(x_1, x_2, \ldots, x_n)$ - meaning that at most one Boolean variable x_i could be `1` and the others must be `0`. A straightforward transformation costs $\mathcal{O}(n^2)$ binary clauses, while by introducing $\mathcal{O}(n)$ additional variables, several transformations exist which require only $\mathcal{O}(n)$ binary clauses. In [AM04] the effects of these transformations on the performance of state-of-the-art SAT solvers are presented: More sophisticated transformations boost performance.

For more difficult constraints, like pseudo-Boolean (PB) constraints, little research has been done regarding the "optimal" transformation. However, even without this knowledge, a significant positive result has been achieved: In the recent PB competitions[9.], one competitor, minisat$^+$, which translates all constraints to CNF (also the optimization function) [ES06], performs best. Apparently, the PB-based solvers cannot capitalize on this richer representation.

2.4.2 Pre-processing

Pre-processing the original formula is generally required to repair an inefficient transformation. It can also be used to modify the formula to the solver's preferred (hybrid) representation. Four pre-processing techniques are discussed:

- **Simplification**: Reduction of a formula can be achieved by propagation of original unit clauses, substitution of equivalent variables $x_i \leftrightarrow (\neg)x_j$, subsumption (removal of redundant clauses), and self-subsumption (removal of redundant literals)

- **Resolution**: Adding (non-redundant) resolvents could significantly improve performance of SAT solvers. In general, complete SAT solvers solve the formulae that are constrained further, faster. While local search may have more difficulties to satisfy all clauses once many are added. Yet, R$^+$AdaptNovelty$^+$ showed - by winning the random SAT division (2005) - that SLS solvers can also profit from added resolvents [APSS05].

[9.] see `http://www.cril.univ-artois.fr/PB05` and `http://www.cril.univ-artois.fr/PB06`

In addition, DP (variable elimination) resolution [DP60] can be used to speed-up solving: This technique is used in the SatElite pre-processor [EB05] which makes the strong minisat solver even faster. This combination won four divisions during the Sat 2005 competition.

- **Syntactical structure detection**: Straightforward transformation of high level (non-clausal) constraints can often be traced cheaply using a syntactical structure detection. This is done in march_eq to extract equivalence (or XOR) constants [HDvZvM04] for its hybrid representation. Syntactical structure detection can also be used to replace a poor transformation by a more sophisticated one.
- **Blocked Clauses**: Graph coloring problems can be encoded using for all vertices v_i `ExactlyOneColor`(v_i) constraints and `NotTheSameColor`(v_i, v_j) constraints for all edges (v_i, v_j) Another valid encoding uses `AtLeastOne-Color`(v_i) for all vertices v_i. The transformation to Sat of the former encoding adds *blocked clauses*, while the latter transformation does not. Blocked clauses arise from the *Extended Resolution Rule* [Kul99]. Although these clauses are redundant (removing them yields a satisfiability equivalent formula), adding them could speed up the solving process. Like graph coloring, many problems can be encoded with or without blocked clauses. Whether the presence of blocked clauses is a curse or a blessing depends on the type of solver. The performance of conflict-driven solvers is generally improved by adding blocked clauses, while SLS Sat solvers slow down. So, pre-processing can be used to add or remove blocked clauses to meet the solver's preference.

2.5 Future progress and contributions

Throughout this chapter, we presented state-of-the-art Sat solving techniques combined with some interesting areas for future research. These topics can be divided in five categories, which will be discussed in this section.

2.5.1 Enhancing the look-ahead architecture

More and optimized reasoning will likely be the key to future progress in Sat solving. This may cast some trouble for conflict-driven Sat solvers: Recall that the use of lazy data-structures will make it difficult to add some techniques of sophisticated reasoning. On the other hand, look-ahead Sat solvers already use quite some reasoning and there appear no obstacles to add even more. However, conflict-driven Sat solvers outperform other architectures on industrial problems, because in recent years, the majority of the Sat solving community has concentrated on these specific benchmarks. Yet, if the focus would shift towards boosting the performances of look-ahead Sat solvers, this gap could be closed.

To contribute on this, we implemented our own look-ahead SAT solver march. The development was focused on maximizing the spectrum of SAT applications on which it achieved competitive performance - while using the look-ahead architecture. An early version, called march_eq, presented in Chapter 3, improved performance by modifying existing look-ahead techniques such that they would work on medium-sized and large problems as well. Also, we implemented additional reasoning techniques (e.g. equivalence reasoning) in such a way that the reasoning would boost the performance if applicable *without* increasing the computational time if not applicable.

We continued by improving performance on structured formulae. The version march_dl, presented in Chapter 4, includes various techniques contributing to this goal. First, we added an advanced pre-processor. It removes various forms of redundancy that often occur in the encoding of structured formulae. After the removal, the pre-processor adds some clauses to increase the number of implied variables while solving the problem. Second, we started replacing some of the static heuristics - mostly optimized towards random formulae - by adaptive heuristics. We will elaborate on this in Section 2.5.3. Third, we developed a new branch strategy that selects decision variables in such a way that the formula is solved locally (instead of globally). This technique appeared essential to solve various structured problems.

Focus on improving look-ahead SAT solvers does not mean that one should neglect the achievements of conflict-driven SAT solvers. On the contrary: A bright future for look-ahead SAT solvers could only be possibly using features that contributed to the current success of conflict-driven SAT solvers. The most important feature is arguably global learning. Modifying this technique to improve the overall performance of look-ahead solvers is the topic of current research.

2.5.2 Enhancing the UnitWalk architecture

The raison d'être of incomplete SAT solvers is that they outperform complete SAT solvers on a wide range of satisfiable instances. On satisfiable random k-SAT formulae they clearly do - in particular the WALKSAT-based solvers. However, on most satisfiable benchmarks with many binary clauses - which is generally the case in structured instances - the incomplete SAT solvers can hardly compete. The UnitWalk SAT solver is a relatively strong incomplete SAT solver on these instances, even though it has not enjoyed much academic attention. Based on the SAT 2003 competition results [LS03], we can state that its costly computations during each period probably disqualify it as a solver for formulae of a certain, substantial, size. Improving this solver with optimizations and heuristics, possibly with additional techniques of WALKSAT-based solvers, may result in a solver that outperforms complete SAT solvers on structured benchmarks, as well.

We implemented our own UnitWalk-based Sat solver called UnitMarch. Chapter 7 describes this solver in detail. Our main contribution is the parallelization of the UnitWalk algorithm by exploiting the architecture of modern computers. Although we implemented the parallelization only for this type of Sat solver, it may also be used to improve others.

A second contribution is the addition of a new reasoning technique [KMT07] to detect autarkies - which to our knowledge has not been used before. The technique can in theory be added to all Sat solvers, but it is better suited for incomplete solvers since they use full assignments. Especially in combination with our parallel solver, this technique seems useful: An autarky found by one of the parallel paths can be communicated to the other ones.

2.5.3 Adaptive heuristics

Of course, some techniques that contributed to WalkSat-based solvers are also expected to improve future Sat solvers - such as the AdaptiveNoise algorithm [Hoo02]. Adaptive heuristics will become more important in the future: As Sat solvers will apply more reasoning, more parameters will be used. Since parameters influence each other, the “optimal” settings may be too complex to determine. In such a case, adaptive heuristics may prove to be very useful.

Also, the success of Sat solving will increase the range of its applications. For many users, Sat solvers are regarded as a black box technology. Since they have no clue about the “optimal” setting for certain parameters, Sat solvers should automatically tune them - with some pre-processing, but preferably using adaptive heuristics. Chapter 5 presents an adaptive algorithm for the DoubleLook procedure. Similar to the AdaptiveNoise algorithm, the proposed algorithm yields better performances on many benchmarks compared to the optimal static setting.

2.5.4 Direction heuristics

Besides modifying and optimizing existing techniques, research should also be focused on promising features that have been more or less neglected in the past. Direction heuristics which can effectively predict the satisfiable branch, is one of these features. They could significantly improve performance on satisfiable instances. Also, they could be used to create short conflict clauses. Chapter 6 describes a method to measure the effectiveness of the direction heuristic used in a solver on a specific family. Moreover, we explain how to capitalize on the observed effectiveness.

2.5.5 Representation

Regarding the optimal representation of a formula (with respect to solving it) not much research has been done, yet. This thesis contains only minor contributions to this research area. These include a new 3-SAT translator (Section 3.2) and addition of ternary clauses (Section 4.3.2). Probably the most influential work is the SatElite pre-processor [EB05], which has been used by the winners of industrial divisions in both the SAT 2005 and SAT 2007 competitions. Although there exist many pre-processing techniques, they rarely are useful for all different kind of CNF formulae. Studying the effectiveness of the various techniques on a large range of benchmarks is required to develop a powerful general purpose pre-processor.

Two roads diverged in a wood, and I –
I took the one less traveled by,
and that has made all the difference.
Robert Frost

3 March_eq*

In the computer game Maniac Mansion you can choose between three avatars to play with. Available avatars are Razor, a hip rock chick with a hairdo to die for, and Jeff Woodie, the ultimate cool surf dude. Appealing as they may be, both Razor and Jeff Woodie lack even the most basic searching skills. Not surprisingly, the most skilled avatar is Bernard, the nerd. Recall that we have to be pragmatic - there is a killer on the hunt. So we do not go for looks, we go for Bernard Bernoulli: A nerd with his gadgets and engineering skills may be a blessing in disguise.

Besides Bernard, there is another avatar you can choose in our Maniac Mansion example (which is not in the actual computer game) that may have what it takes to find the front door key: A maid. She may be modest, but she is quick, thorough and she has the right working morale.

SAT solvers share many similarities with the characters above. Some search fast without much reasoning. Other solvers are clever. They think more, but they are much slower, especially on huge formulae. The former type can be associated with the conflict-driven architecture (maid), which is also the dominant structure of modern SAT solvers, while the latter describes features of the look-ahead architecture (nerd).

Look-ahead SAT solvers are relatively slow, but they have great potential thanks to their additional reasoning. This chapter describes an early version (2004) of our own look-ahead solver, called march_eq. This solver reduces the computational costs of many additional reasoning techniques used by look-ahead SAT solvers – to make it competitive.

*This chapter is based on: Marijn J.H. Heule, Joris van Zwieten, Mark Dufour and Hans van Maaren. *March_eq: Implementing Additional Reasoning into an Efficient Lookahead Sat Solver.* SAT 2004 Springer LNCS **3542** (2005), 345–359.

3.1 Introduction

Look-ahead SAT solvers usually consist of a simple DPLL algorithm [DLL62] and a more sophisticated *look-ahead procedure* to determine an effective decision variable. The look-ahead procedure measures the effectiveness of variables by performing *look-ahead* on a set of variables and evaluating the reduction of the formula. We refer to the look-ahead on literal x as the Iterative Unit Propagation (IUP) on the union of a formula with the unit clause x (in short IUP($\mathcal{F} \cup \{x\}$)). The effectiveness of a variable x_i is obtained using a look-ahead evaluation function (in short DIFF), which evaluates the differences between $\mathcal{F}$ and the reduced formula after IUP($\mathcal{F} \cup \{x_i\}$) and IUP($\mathcal{F} \cup \{\neg x_i\}$). A widely used DIFF counts the newly created binary clauses.

Besides the selection of a decision variable, the look-ahead procedure may detect *failed literals*: If the look-ahead on $\neg x$ results in a conflict, x is forced to true. Detection of failed literals can result in a substantial reduction of the DPLL-tree.

During the last decade, several enhancements have been proposed to make look-ahead SAT solvers more powerful. In satz by Li [LA97b] heuristics PROP_z are used, which restrict the number of variables that enter the look-ahead procedure. Especially on random instances the application of these heuristics results in a clear performance gain. However, the use of these heuristics is not clear from a general viewpoint. Experiments with our pre-selection heuristics show that different benchmark families require different numbers of variables entering the look-ahead phase to perform optimally.

Since much reasoning is already performed at each node of the DPLL-tree, it is relatively cheap to extend the look-ahead with (some) additional reasoning. For instance: Integration of equivalence reasoning in satz - implemented in eqsatz [Li03] - made it possible to solve various crafted and real-world problems which were beyond the reach of existing techniques. However, the performance may drop significantly on some problems, due to the integrated equivalence reasoning. Our variant of equivalence reasoning extends the set of problems which benefit from its integration and aims to remove the disadvantages.

Another form of additional reasoning is implemented in OKsolver[10.] [Kul02]: *Local learning*. When performing look-ahead on x, any unit clause y_i that is found means that the binary clause $\neg x \vee y_i$ is implied by the formula, and can be "learned", i.e. added to the current formula. As with equivalence reasoning, addition of these local learned resolvents could both increase and decrease the performance (depending on the formula). We propose a partial addition of these resolvents which results in a speed-up practically everywhere.

Generally, look-ahead SAT solvers are effective on relatively small, hard formulas. Le Berre [LeB01] proposes a wide range of enhancements of the look-ahead procedure. Most of them are implemented in march_eq. Due to the high computational costs of the an enhanced look-ahead procedure, elaborate problems are often solved more efficiently by other techniques. Reducing these costs

[10.] Version 1.2 at http://cs-svr1.swan.ac.uk/~csoliver/OKsolver.html is essential for making look-ahead techniques more competitive on a wider range of benchmarks problems. In this chapter, we suggest (i) several techniques to reduce these costs and (ii) a cheap integration of additional reasoning. Due to the latter, benchmarks that do not profit from additional reasoning will not be significantly harder to solve.

Most topics discussed in this chapter are illustrated with experimental results showing the performance gains by our proposed techniques. The benchmarks range from `uniform random` 3-SAT near the observed phase transition [MSL92], to bounded model checking (`longmult` [BCCZ99], `zarpas` [LS03]), factoring problems (`pyhala braun` [SLH05]) and crafted problems (`stanion/hwb` [LS03], `quasigroup` [ZS00]). Only unsatisfiable instances were selected to provide a more stable overview. Comparison of the performance of march_eq with performances of state-of-the-art solvers is presented in [HvM04].

All techniques have been implemented into a reference variant of march_eq, which is essentially a slightly optimised version of march_eq_100, the solver that won two categories of the SAT 2004 competition [LS04]. This variant uses exactly the same techniques as the winning variant: Full (100%) look-ahead, addition of all constraint resolvents, tree-based look-ahead, equivalence reasoning, and removal of inactive clauses. All these techniques are discussed below.

3.2 Translation to 3-SAT

The translation of the input formula to 3-SAT stems from an early version of march_eq, in which it was essential to allow fast computation of the pre-selection heuristics. Translation is not required for the current pre-selection heuristics, yet it is still used, because it enables significant optimization of the internal data-structures.

The formula is pre-processed to reduce the amount of redundancy introduced by a straightforward 3-SAT translation. Each pair of literals that occurs more than once together in a clause in the formula is substituted by a single *dummy* variable, starting with the most frequently occurring pair. Three clauses are added for each dummy variable to make it satisfiability equivalent to the disjunction of the pair of literals it substitutes. In the example below, $\neg x_2 \vee x_4$ is the most occurring literal pair and is therefore replaced with the dummy variable d_1.

$$\begin{array}{r} x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \vee \neg x_5 \\ x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \vee x_6 \\ \neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4 \vee \neg x_6 \\ \neg x_1 \vee \neg x_2 \vee x_4 \vee x_5 \vee x_6 \end{array} \Leftrightarrow \begin{array}{r} x_1 \vee d_1 \vee \neg x_3 \vee \neg x_5 \\ x_1 \vee d_1 \vee \neg x_3 \vee x_6 \\ \neg x_1 \vee d_1 \vee \neg x_3 \vee \neg x_6 \\ \neg x_1 \vee d_1 \vee x_5 \vee x_6 \end{array} \wedge \begin{array}{r} d_1 \vee x_2 \\ d_1 \vee \neg x_4 \\ \neg d_1 \vee \neg x_2 \vee x_4 \end{array}$$

Notice that resolution on dummy variable d_1 always requires $\neg d_1 \vee \neg x_2 \vee x_4$. Yet, for the clauses $d_1 \vee x_2$ and $d_1 \vee \neg x_4$ this results in tautological clauses which contain either both x_2 and $\neg x_2$ or both x_4 and $\neg x_4$, respectively. In the literature these clauses are known as *blocked clauses* [Kul99]. Blocked clauses can be removed resulting in a satisfiability equivalent formula. However, we do not remove them because their presence does in practice improve performance of march_eq. This result may not hold for other SAT solvers.

It appears that to achieve good performance, binary clauses obtained from the *original* ternary clauses should be given more weight than binary clauses obtained from ternary clauses which were *generated by translation*. This is accomplished by an appropriate look-ahead evaluation function, such as the variant of DIFF proposed by Dubois *et al.* [DD01], which weighs all newly created binary clauses.

3.3 Time Stamps

March_eq uses a time stamp data structure, *TimeAssignments* (TA), which reduces backtracking during the look-ahead phase to a single integer addition: Increasing the *CurrentTimeStamp* (CTS).

All the variables that are assigned during look-ahead on a literal x are stamped: If a variable is assigned the value true, it is stamped with the CTS; if it is assigned the value false, it is stamped with $CTS + 1$. Therefore, simply adding 2 to the CTS unassigns all assigned variables.

The actual truth value that is assigned to a variable is not stored in the data structure, but can be derived from the time stamp of the variable:

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

Variables that have already been assigned before the start of the look-ahead phase, i.e. during the solving phase, have been stamped with the *Maximum-TimeStamp* (MTS) or with $MTS + 1$. These variables can be unassigned by stamping them with the value *zero*, which happens while backtracking during the solving phase (i.e. *not* during the look-ahead phase). The MTS equals the maximal even value of an (32-bit) integer. One has to ensure that the CTS is always smaller than the MTS. This will usually be the case and it can easily be checked at the start of each look-ahead.

3.4 Constraint Resolvents

As mentioned in the introduction, a binary resolvent could be added for every unary clause that is created during the propagation of a look-ahead literal - provided that the binary clause does not already exist. A special type of resolvent is created from a unary clause that was a ternary clause prior to the look-ahead. In this case we speak of *constraint resolvents*.

Constraint resolvents have the property that they cannot be found by a look-ahead on the complement of the unary clause. Adding these constraint resolvents results in a more vigorous detection of failed literals. An example:

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

(i) (ii) (iii)

Figure 3.1 — Detection of a failed literal by adding constraint resolvents. (i) The original clauses, (ii) constraint resolvents and (iii) a forced literal.

First, consider only the original clauses of an example formula (Figure 3.1 (i)). A look-ahead on $\neg r$, IUP$(\mathcal{F} \cup \{\neg r\})$, results in the unary clause x. Therefore, one could add the resolvent $r \lor x$ to the formula. Since the unary clause x was originally a ternary clause (before the look-ahead on $\neg r$), this is a constraint resolvent. The unique property of constraints resolvents is that when they are added to the formula, look-ahead on the complement of the unary clause results in the complement of the look-ahead-literal. Without this addition this would not be the case. Applying this to the example: After addition of $r \lor x$ to the formula, IUP$(\mathcal{F} \cup \{\neg x\})$ will result in unary clause r, while without this addition it will not.

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

Constraint resolvent $u \lor x$ is detected during IUP$(\mathcal{F} \cup \{\neg u\})$. After the addition of both constraint resolvents (Figure 3.1 (ii)), the look-ahead IUP$(\mathcal{F} \cup \{\neg x\})$ results in a conflict, making $\neg x$ a failed literal and thus forces x. Obviously, IUP$(\mathcal{F} \cup \{\neg x\})$ will not result in a conflict if the constraint resolvents $r \lor x$ and $u \lor x$ were not added both.

Table 3.1 shows the usefulness of the concept of constraint resolvents: In all our experiments, the addition of mere constraint resolvents outperformed a variant with full local learning (adding all binary resolvents). This could be explained by the above example: Adding other resolvents than constraint resolvents will not increase the number of detected failed literals. These resolvents merely increase the computational costs. This explanation is supported by the data in the table: The tree-size of both variants is comparable.

When we look at `zarpas/rule_14_1_30dat`, it appears that only adding constraint resolvents is essential to solve this benchmark within 2000 seconds. The node-count of zero means that the instance is found unsatisfiable during the first execution of the look-ahead procedure.

Table 3.1 — Performance of march_eq on several benchmarks with three different settings of addition of resolvents during the look-ahead phase.

	no resolvents		**all binary resolvents**		**all constraint resolvents**	
Benchmarks	time(s)	treesize	time(s)	treesize	time(s)	treesize
random_unsat_250 (100)	1.61	4059.1	1.51	3389.2	**1.45**	3391.7
random_unsat_350 (100)	55.41	89709	51.28	72721	**48.78**	73357
stanion/hwb-n20-01	31.52	282882	24.76	180408	**23.65**	183553
stanion/hwb-n20-02	41.32	345703	33.94	219915	**30.91**	222251
stanion/hwb-n20-03	30.54	280561	23.48	161687	**21.7**	163984
longmult8	139.13	15905	341.46	8054	**90.8**	8149
longmult10	504.92	330094	915.84	11877	**226.31**	11597
longmult12	836.78	41522	847.95	5273	**176.85**	5426
pyhala-unsat-35-4-03	781.19	29591	1379.33	19100	**662.93**	19517
pyhala-unsat-35-4-04	733.44	28312	1366.19	18901	**659.04**	19364
quasigroup3-9	11.67	2139	11.09	1543	**7.97**	1495
quasigroup6-12	117.49	3177	66.13	1362	**58.05**	1311
quasigroup7-12	14.47	346	11.06	248	**10.03**	256
zarpas/rule14_1_15dat	> 2000	-	46.59	0	**20.7**	0
zarpas/rule14_1_30dat	> 2000	-	> 2000	-	**186.27**	0

3.5 Implication Arrays

Due to the 3-SAT translation the data structure of march_eq only needs to accommodate binary and ternary clauses. We will use the following formula as an example:

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

Binary and ternary clauses are stored separately in two *implication arrays.* A binary clause $a \vee c$ is stored as two implications: c is stored in the binary implication array of $\neg a$ and a is stored in the binary implication array of $\neg c$. A ternary clause $(a \vee \neg b \vee d)$ is stored as three implications: $\neg b \vee d$ is stored in the ternary implication array of $\neg a$ and the similar is done for b and $\neg d$. Figure 3.2 shows the implication arrays that represent the example formula $\mathcal{F}_{\text{example}}$.

Storing binary clauses in implication arrays requires only half the memory that would be needed to store them in an ordinary clause database / variable index data structure - see Figure 3.3. Since march_eq adds many 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 memory used by march_eq. Furthermore, the implication arrays improve data locality. This often leads to a speed-up due to better usage of the cache.

March_eq uses a variant of iterative unit propagation (IFIUP) that propagates binary implications before ternary implications. The first step of this procedure is to assign as many variables as possible using only the binary implication arrays. Then, if no conflict is found, the ternary implication array of each variable that was assigned in the first step is evaluated. We will illustrate this second step with an example.

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

(i)

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

(ii)

Figure 3.2 — The binary (i) and ternary (ii) implication arrays that represent the example formula $\mathcal{F}_{\text{example}}$.

#	clause		
0	a	c	
1	$\neg b$	$\neg d$	
2	b	d	
3	a	$\neg b$	d
4	$\neg a$	b	$\neg d$
5	$\neg a$	b	c

(i)

a	0	3	
$\neg a$	4	5	
b	2	4	5
$\neg b$	1	3	
c	0	5	
$\neg c$			
d	2	3	
$\neg d$	1	4	

(ii)

Figure 3.3 — A common clause database / variable index data structure. All clauses are stored in a clause database (i), and for each literal the variable index lists the clauses in which it occurs (ii).

Suppose look-ahead is performed on $\neg c$. The ternary implication array of $\neg c$ contains $(\neg a \vee b)$. Now there are five possibilities:

1. If the clause is already satisfied, i.e. a has already been assigned the value false or b has already been assigned the value true, then nothing needs to be done.
2. If a has already been assigned the value true, then b is implied and so b is assigned the value true. The first step of the procedure is called to assign as many variables implied by b as possible. Also, the constraint resolvent $(c \vee b)$ is added as two binary implications.

3. If b has already been assigned the value false, then $\neg a$ is implied and so a is assigned the value false. The first step of the procedure is called to assign as many variables implied by $\neg a$ as possible. Also, the constraint resolvent $(c \vee \neg a)$ is added as two binary implications.

4. If a and b are unassigned, then we have found a new binary clause.

5. If a has already been assigned the value true and b has already been assigned the value false, then $\neg c$ is a failed literal. Thus c is implied.

The variant of DIFF used in march_eq weighs new binary clauses that are produced during the look-ahead phase. A ternary clause that is reduced to a binary clause that gets satisfied in the same iteration of IFIUP, should *not* be included in this computation. However, in the current implementation these clauses are in fact included, which causes noise in the DIFF heuristics. The first step of the IFIUP procedure, combined with the addition of constraint resolvents, ensures that the highest possible amount of variables are assigned *before* the second step of the IFIUP procedure. This reduces the noise significantly.

An advantage of IFIUP over general IUP is that it will detect conflicts faster. Due to the addition of constraint resolvents, most conflicts will be detected in the first call of the first step of IFIUP. In such a case, the second step of IFIUP is never executed. Since the second step of IFIUP is considerably slower than the first, an overall speed-up is expected.

Storage of ternary clauses in implication arrays requires an equal amount of memory as the common alternative. However, ternary implication arrays allow optimisation of the second step of the IFIUP procedure. On the other hand, ternary clauses are no longer stored as such: It is not possible to efficiently verify if they have already been satisfied and early detection of a solution is neglected. One knows only that a solution exists if all variables have been assigned and no conflict has occurred.

3.6 Equivalence Reasoning

During the pre-processing phase, march_eq extracts the so-called equivalence clauses $(l_1 \leftrightarrow l_2 \leftrightarrow \cdots \leftrightarrow l_i)$ from the formula and places them into a separate data-structure called the *Conjunction of Equivalences* (CoE). After extraction, a solution for the CoE is computed as described in [HvM04, WvM98].

In [HvM04], we propose a new look-ahead evaluation function for benchmarks containing equivalence clauses: Let eq_n be a weight for a reduced equivalence clause of new length n, $\mathcal{C}(x)$ the set of all reduced equivalence clauses $(\mathcal{Q}_i)$ during a look-ahead on x, and $\mathcal{B}(x)$ the set of all newly created binary clauses during the look-ahead on x. Using both sets, the look-ahead evaluation can be calculated as in Equation 3.2. Variable x_i with the highest $\text{DIFF}_{eq}(x_i) \times \text{DIFF}_{eq}(\neg x_i)$ is selected for branching.

$$eq_n = 5.5 \times 0.85^n \tag{⌊3.1}$$

$$\textsc{Diff}_{eq} = |\mathcal{B}| + \sum_{\mathcal{Q}_i \epsilon \mathcal{C}} eq_{|\mathcal{Q}_i|} \tag{⌊3.2}$$

Besides the look-ahead evaluation and the pre-selection heuristics (discussed in Section 3.7), the intensity of communication between the CoE- and CNF-part of the formula is kept rather low (see Figure 3.4). Naturally, all unary clauses in all phases of the solver are exchanged between both parts. However, during the solving phase, all binary equivalences are removed from the CoE and transformed to the four equivalent binary implications which in turn are added to the implication arrays. The reason for this is twofold: (i) The binary implication structure is faster during the look-ahead phase than the CoE-structure, and (ii) for all unary clauses y_i that are created in the CoE during IUP($\mathcal{F} \cup \{x\}$), constraint resolvent $\neg x \vee y_i$ can be added to the formula without having to check the original length.

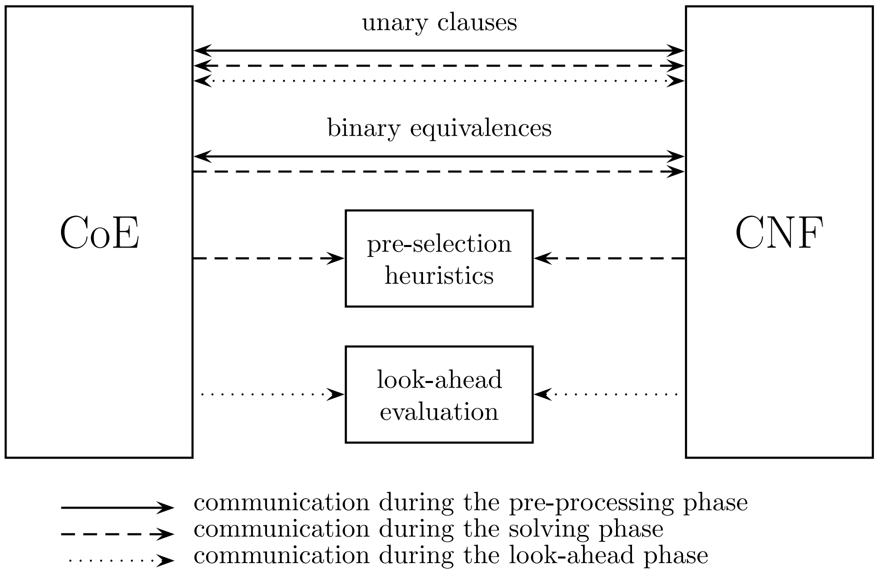
unary clauses
binary equivalences
CoE
CNF
pre-selection heuristics
look-ahead evaluation
communication during the pre-processing phase
communication during the solving phase
communication during the look-ahead phase

Figure 3.4 — Various forms of communication in march_eq

We examined other forms of communication, but only small gains were noticed on only some problems. Mostly, performance decreased due to higher communication costs. For instance: Communication of binary equivalences from the CNF- to the CoE-part makes it possible to substitute those binary equivalences in order to reduce the total length of the equivalence clauses. This rarely resulted in an overall speed-up.

We tried to integrate the equivalence reasoning in such a manner that it would only be applied when the performance would benefit from it. Therefore, march_eq does not perform any equivalence reasoning if no equivalence clauses are detected during the pre-processing phase (if no CoE exists), making march_eq equivalent to its older brother march.

Table 3.2 shows that the integration of equivalence reasoning in march rarely results in a loss of performance: On some benchmarks like the random_unsat and the quasigroup family no performance difference is noticed, since no equivalence clauses were detected. Most families containing equivalence clauses are solved faster due to the integration. However, there are some exceptions, like the longmult family in the table.

If we compare the integration of equivalence reasoning in march (which resulted in march_eq) with the integration in satz (which resulted in eqsatz), we note that eqsatz is much slower than satz on benchmarks that contain no equivalence clauses. While satz[11.] solves 100 random_unsat_350 benchmarks near the treshold on average in 22.14 seconds using 105798 nodes, eqsatz[12.] requires on average 795.85 seconds and 43308 nodes to solve the same set. Note that no slowdown occurs for march_eq.

Table 3.2 — Performance of march_eq on several benchmarks with and without equivalence reasoning.

	without equivalence reasoning		**with equivalence reasoning**		
Benchmarks	time(s)	treesize	time(s)	treesize	speed-up
random_unsat_250 (100)	1.45	3391.7	1.45	3391.7	-
random_unsat_350 (100)	48.78	73357.2	48.78	73357.2	-
stanion/hwb-n20-01	42.88	182575	23.65	183553	44.85 %
stanion/hwb-n20-02	55.34	222487	30.91	222251	44.15 %
stanion/hwb-n20-03	42.08	164131	21.70	163984	48.43 %
longmult8	76.69	8091	90.80	8149	-18.40 %
longmult10	171.66	11597	226.31	11597	-31.84 %
longmult12	126.36	6038	176.85	5426	-39.96 %
pyhala-unsat-35-4-03	737.15	19513	662.93	19517	10.07 %
pyhala-unsat-35-4-04	691.04	19378	659.04	19364	4.63 %
quasigroup3-9	7.97	1495	7.97	1495	-
quasigroup6-12	58.05	1311	58.05	1311	-
quasigroup7-12	10.03	256	10.03	256	-
zarpas/rule14_1_15dat	21.68	0	20.70	0	4.52 %
zarpas/rule14_1_30dat	219.61	0	186.27	0	15.18 %

[11.] Version 2.15.2 at http://www.laria.u-picardie.fr/~cli/EnglishPage.html
[12.] Version 2.0 at http://www.laria.u-picardie.fr/~cli/EnglishPage.html

3.7 Pre-selection Heuristics

Overall performance can be gained or lost by performing look-ahead on a subset of the free variables in a node: Gains are achieved by the reduction of computational costs, while losses are the result of either the inability of the *pre-selection heuristics* (heuristics that determine the set of variables to enter the look-ahead phase) to select effective decision variables or the lack of detected failed literals. When look-ahead is performed on only a subset of the variables, only a subset of the failed literals is most likely detected. Depending on the formula, this could increase the size of the DPLL-tree.

During our experiments, we used pre-selection heuristics which are an approximation of our combined look-ahead evaluation function (Ace) [HvM04]. These pre-selection heuristics are costly, but because they provide a clear discrimination between the variables, a small subset of variables could be selected. Experiments with a *fixed number* of variables entering the look-ahead procedure is shown in Figure 3.5. The fixed number is based on a percentage of the original number of variables and the "best" variables (with the highest pre-selection ranking) are selected.

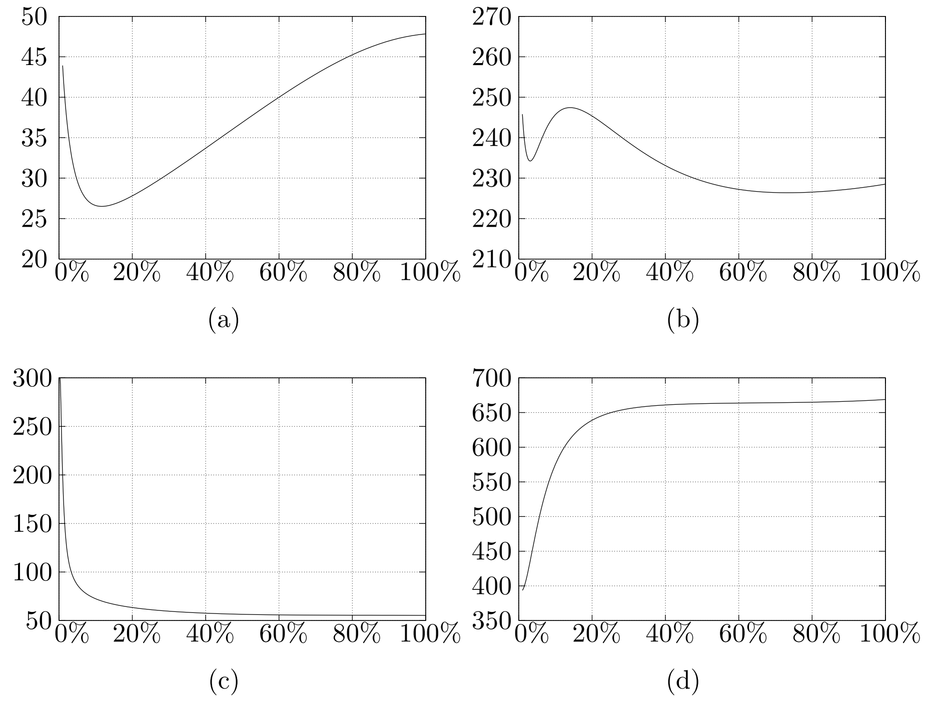

Figure 3.5 — Runtime(s) vs. percentage look-ahead variables on single instances: (a) `random_unsat_350`; (b) `longmult10`; (c) `quasigroup6-12`; and (d) `pyhala-braun-unsat-35-4-04`.

The plots in this figure do not offer any indication of which percentage is required to achieve optimal general performance: While for some instances 100% look-ahead appears optimal, others are solved faster using a much smaller percentage.

Two variants of march_eq were submitted to the SAT 2004 competition [LS04]: One which selects in every node the "best" 10 % variables (march_eq_010) and one with full (100%) look-ahead (march_eq_100). Although during our experiments the first variant solved the most benchmarks, at the competition both variants solved the same number of benchmarks, albeit different ones. The differences can be explained by the behavior of versions of march_eq with different percentages shown in Figure 3.5.

3.8 Tree-based Look-ahead

The structure of our look-ahead procedure is based on the observation that different literals to perform look-ahead on, often entail certain shared implications, and that we can form 'sharing' trees from these relations, which in turn may be used to reduce the number of times these implications have to be propagated during look-ahead.

Suppose that two look-ahead literals share a certain implication. In this simple case, we could propagate the shared implication first, followed by a propagation of one of the look-ahead literals, backtracking the latter, then propagating the other look-ahead literal and only in the end backtracking to the initial state. This way, the shared implication has been propagated only once.

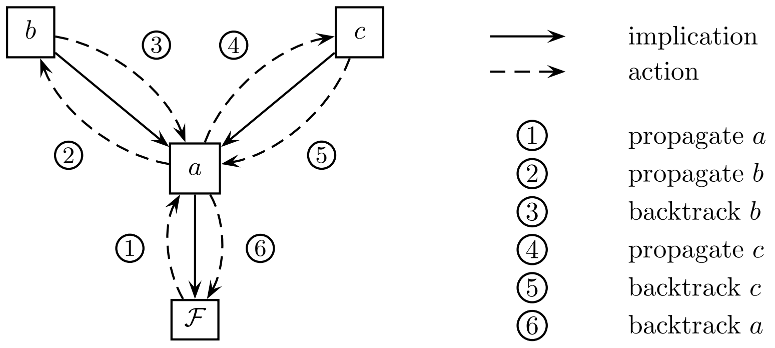

Figure 3.6 — Graphical form of an implication tree with corresponding actions.

Figure 3.6 shows this example graphically. The implications among a, b and c form a small tree. Some thought reveals that this process, when applied recursively, could work for arbitrary trees. Based on this idea, our solver extracts - prior to look-ahead - trees from the implications among the literals selected for look-ahead, in such a way that each literal occurs in exactly one tree. The look-ahead procedure is improved by recursively visiting these trees. Of course, the more dense the implication graph, the more possibilities are available for forming trees, so local learning will in many cases be an important catalyst for the effectiveness of this method.

Unfortunately, there are many ways of extracting trees from a graph, so that each vertex occurs in exactly one tree. Large trees are obviously desirable, as they imply more sharing, as does having literals with the most impact on the formula near the root of a tree. To this end, we have developed a simple heuristic. More involved methods would probably produce better results, although optimality in this area could easily mean solving NP-complete problems again. We consider this an interesting direction for future research.

Our heuristic requires a list of predictions to be available, of the relative amount of propagations that each look-ahead literal implies, to be able to construct trees that share as much of these as possible. In the case of march_eq, the pre-selection heuristic provides us with such a list.

The heuristic now travels this list once, in order of decreasing prediction, while constructing trees out of the corresponding literals. It does this by determining for each literal, if available, one other look-ahead literal that will become its parent in some tree. When a literal is assigned a parent, this relationship remains fixed. On the outset, as much trees are created as there are look-ahead literals, each consisting of just the corresponding literal.

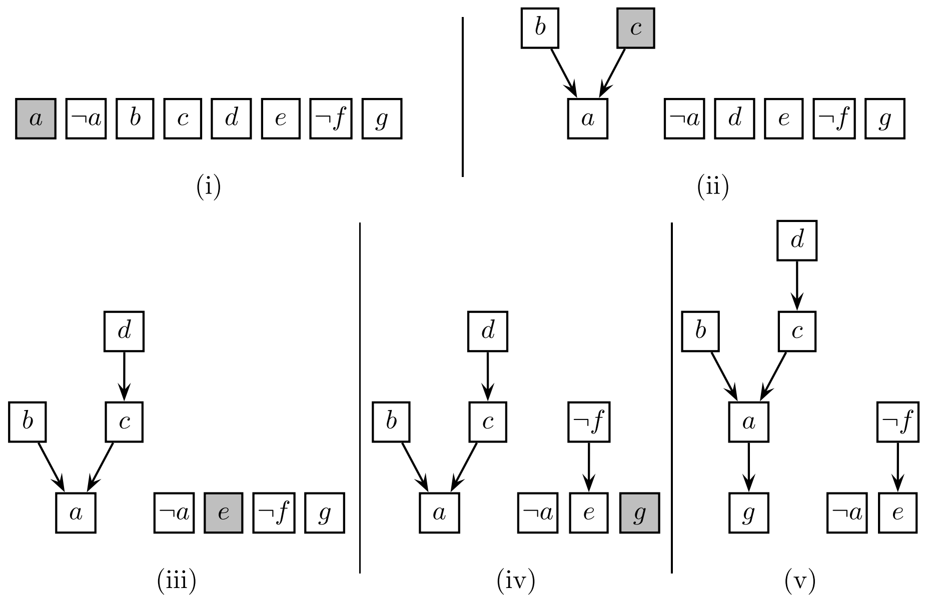

Figure 3.7 — Five steps of building implication trees.

More specifically, for each literal that it encounters, the heuristic checks whether this literal is implied by any other look-ahead literals that are the root of some tree. If so, these are labeled child nodes of the node corresponding to the implied literal. If not already encountered, these child nodes are now recursively checked in the same manner. At the same time, we remove the corresponding elements from the list, so that each literal will be checked exactly once, and will receive a position within exactly one tree.

Figure 3.7 shows the process for a small set of look-ahead literals. The formula for this example contains the binary clause $a \vee \neg b$, $a \vee \neg c$, $\neg a \vee g$, $c \vee \neg d$, and $e \vee f$ amongst other clauses. A gray box in the figure denotes the current position.

Because of the order in which the list is traveled, literals which have received higher predictions are labeled as parent nodes as early as possible. This is important, because it is often possible to extract many different trees from an implication graph, and because every literal should occur in exactly one tree.

Availability of implication trees opens up several possibilities of going beyond resolution. One such possibility is to detect implied literals. Whenever a node has descendants that are complementary, clearly the corresponding literal is implied. By approximation, we detect this for the most important literals, as these should have ended up near the roots of larger trees by the above heuristic. For solvers unable to deduce such implications by themselves, we suggest a simple, linear-time algorithm that scans the trees.

Some intriguing ideas for further research have occurred to us during the development our tree-based look-ahead procedure, but which, we have not been able to pursue due to time constraints. One possible extension would be to add variables that both positively and negatively imply some look-ahead literal as full-fledged look-ahead variables. This way we may discover important, but previously undetected variables to perform look-ahead on and possibly branch upon. Because of the inherent sharing, the overhead will be smaller than without a tree-based look-ahead.

Also, once trees have been created, we could include non-look-ahead literals in the sharing, as well as in the checking of implied literals. As for the first, suppose that literals a and b imply some literal c. In this case we could share not just the propagation of c, but also that of any other shared implications of a and b. Sharing among tree roots could be exploited in the same manner, with the difference that in the case of many shared implications, we would have to determine which trees could best share implications with each other. In general, it might be a good idea to focus in detail on possibilities of sharing.

Table 3.3 — Performance of march_eq on several benchmarks with and without the use of tree-based look-ahead.

	normal look-ahead		**tree-based look-ahead**		
Benchmarks	time(s)	treesize	time(s)	treesize	speed-up
`random_unsat_250` (100)	1.24	3428.5	1.45	3391.7	-16.94 %
`random_unsat_350` (100)	40.57	74501.7	48.78	73357.2	-20.24 %
`stanion/hwb-n20-01`	29.55	184363	23.65	183553	19.97 %
`stanion/hwb-n20-02`	40.93	227237	30.91	222251	24.48 %
`stanion/hwb-n20-03`	25.88	155702	21.70	163984	16.15 %
`longmult8`	332.64	7918	90.80	8149	72.70 %
`longmult10`	1014.09	10861	226.31	11597	77.68 %
`longmult12`	727.01	4654	176.85	5426	75.67 %
`pyhala-unsat-35-4-03`	1084.08	19093	662.93	19517	38.85 %
`pyhala-unsat-35-4-04`	1098.50	19493	659.04	19364	40.01 %
`quasigroup3-9`	8.85	1508	7.97	1495	9.94 %
`quasigroup6-12`	78.75	1339	58.05	1311	26.29 %
`quasigroup7-12`	13.03	268	10.03	256	23.02 %
`zarpas/rule14_1_15dat`	25.62	0	20.70	0	19.20 %
`zarpas/rule14_1_30dat`	192.30	0	186.27	0	3.14 %

3.9 Removal of Inactive Clauses

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

When a variable x is assigned to a certain truth value during the solving phase, all the ternary clauses in which it occurs become inactive in the arrays pointing to clause indices: The clauses in which x occurs positively become satisfied, while those clauses in which it occurs negatively are reduced to binary clauses. These binary clauses are moved to the implication arrays.

Table 3.4 shows that the removal of inactive clauses during the solving phase is useful on all kinds of benchmarks. Although the speed-up is only small on uniform random benchmarks, larger gains are achieved on more structured instances.

Table 3.4 — Performance of march_eq on several benchmarks with and without the removal of inactive clauses on the chosen path.

	without removal		**with removal**		
Benchmarks	time(s)	treesize	time(s)	treesize	speed-up
`random_unsat_250` (100)	1.70	3393.7	1.45	3391.7	14.71 %
`random_unsat_350` (100)	63.38	73371.9	48.78	73357.2	23.04 %
`stanion/hwb-n20-01`	24.92	182575	23.65	183553	5.10 %
`stanion/hwb-n20-02`	33.78	222487	30.91	222251	8.50 %
`stanion/hwb-n20-03`	23.68	164131	21.70	163984	8.36 %
`longmult8`	114.71	8091	90.80	8149	20.84 %
`longmult10`	287.37	11597	226.31	11597	21.25 %
`longmult12`	254.51	6038	176.85	5426	30.51 %
`pyhala-unsat-35-4-03`	783.52	19513	662.93	19517	15.39 %
`pyhala-unsat-35-4-04`	772.59	19378	659.04	19364	14.70 %
`quasigroup3-9`	11.73	1497	7.97	1495	32.05 %
`quasigroup6-12`	136.70	1335	58.05	1311	57.53 %
`quasigroup7-12`	22.53	256	10.03	256	55.48 %
`zarpas/rule14_1_15dat`	29.80	0	20.70	0	30.54 %
`zarpas/rule14_1_30dat`	254.81	0	186.27	0	26.90 %

3.10 Conclusion

Several techniques have been discussed to increase the solving capabilities of a look-ahead SAT solver. Some are essential for solving various specific benchmarks: A range of families can only be solved using equivalence reasoning, and as we have seen, march_eq is able to solve a large `zarpas` benchmark by only adding constraint resolvents.

Other proposed techniques generally result in a performance boost. However, the usefulness of our pre-selection heuristics is as yet undoubtedly subject to improvement and will be subject of future research.

To equal a predecessor,
one must have twice they worth.
Baltasar Gracian

4 March_dl*

In the sequel of Maniac Mansion[13.], you are able to travel through time. Unfortunately, time traveling provides no escape from the maniac, but at each time frame the mansion differs in size layout and furniture. Yet the front door is always locked. You can choose the moment in time for your search, knowing that if a key exists somewhere in time, it exists at any time.

The question arises to which moment in time you want to travel: Is it to the past, when the house was still small but with broken stairways and no electricity? Or in the future, where the mansion will have grown to massive proportions, but its elevators make it easy to move fast? You have to choose between size and mobility. There is no moment in time in which all conditions will be perfect. The optimal choice will depend heavily on how you search.

Back to SAT. Similarly to the above, you have some influence on the representation of a formula (mansion) that needs to be solved. Mostly, the given formula is a problem translated into SAT and frequently the provided translation (current time frame) is far from optimal for your SAT solver. Depending on the solver, you might prefer a totally different representation. In short, reducing size appears useful for most complete solving methods, while mobility suits incomplete solvers [Pre07].

We added an enhanced pre-processor to march_eq. Before solving, this pre-processor reduces the size of the formula. After reduction, constraints are added. Together with the other contributions presented in this chapter this resulted in version march_dl. Besides modification of the formula, this chapter also introduces a new strategy that attempts to solve a problem locally. In the context of the Maniac Mansion example, this strategy works as follows: In the old way, each succeeding room is selected such that it is next to the current one. This might require quite some walking since the door of that room may not be the nearest door around. For instance the door to the room across the hallway may be nearer. Therefore, we improved on this by selecting the succeeding room by nearest closed door instead of the door to the adjacent room.

*This chapter is based on: Marijn J.H. Heule and Hans van Maaren. *March_dl: Adding Adaptive Heuristics and a New Branching Strategy.* Journal on Satisfiability, Boolean Modeling and Computation **2** (2006), 47-59.

13. Maniac Mansion: Day of the Tentacle, LucasArts, 1993

4.1 Introduction

The satisfiability (SAT) competitions of the last years have boosted the development of SAT solvers: Each year, several unsolvable benchmarks (within the given time limit), were easily solved the year after. Modern SAT solver architectures can be split into three divisions: Conflict-driven (minisat, vallst, zChaff), look-ahead (kcnfs, march, OKsolver) and local search (AdaptNovelty+, R+ AdaptNovelty+, unitwalk). All solvers mentioned above won a category in the past competitions [LS03, LS04, LS06, SLH05]. Each architecture outperforms the other two on parts of the spectrum of available CNF instances. For instance, conflict-driven solvers dominate on industrial formulae, look-ahead solvers are very strong on unsatisfiable random formulae, while local-search techniques are unbeatable on large satisfiable random formulae.

As a look-ahead SAT solver, early development of march was focused on fast performance on unsatisfiable uniform random 3-SAT formulas. Frustrated by the poor performance on structured instances, we attempted to increase the speed on this latter kind of benchmarks by additional reasoning and eager data-structures. The resulted solver, march_eq, is described in detail in [HDvZvM04]. Since equivalence reasoning - an important part of march_eq and thus march_dl - is not further developed, we will ignore this aspect of the solver in this chapter.

The march_eq solver was quite successful: It won two (crafted) categories during the SAT 2004 competition [LS04]. However, various benchmarks - relatively easy to solve by conflict-driven solvers - were still unsolvable by march_eq. We developed some enhancements in order to solve several of these instances. These enhancements are the primary focus of this chapter.

The usefulness of each enhancement is illustrated by some experiments. We selected a small set of benchmarks for this purpose, since extensive experiments are beyond the scope of this chapter. Because look-ahead SAT solvers perform relatively well on unsatisfiable uniform random 3-SAT formulae, we generated[14.] 200 of them (100 of 250 variables with 1075 clauses, and 100 of 350 variables with 1500 clauses) as a reference. We added some crafted and structured instances from five families:

- the `connamacher` family contributed by Connamacher to SAT 2004. This family consists of encodings of the generic uniquely extendible constraint satisfaction problem [Con04].
- the `ezfact` family contributed by Pehoushek to SAT 2002 [SLH05]. These benchmarks are encodings of factorization problems.
- the `lksat` family contributed by Anton to SAT 2004 [LS04]. These are random l-clustered k-SAT instances.
- the `longmult` family contributed by Biere. Instances from this family arise from bounded model checking [BCCZ99].
- the `philips` family contributed by Heule to SAT 2004 [LS04]. Encoding of a multiplier circuit provided by Philips.

[14.] using mkcnf available from `http://www.satlib.org`.

All experiments were performed on a system with an Intel 3.0 GHz CPU and 1 Gb of memory running on Fedora Core 4.

The remaining part of this chapter is structured as follows: Section 4.2 provides a short introduction to the look-ahead architecture together with some references to the origin of the techniques. Section 4.3 deals with two small enhancements to the march_dl pre-processor. Two new adaptive heuristics are introduced in Section 4.4, and a new branching strategy is presented in Section 4.5. Finally, Section 4.6 concludes with some results on the overall performance.

4.2 Look-ahead architecture

Since march_dl is a look-ahead SAT solver, we will first provide a brief introduction on its general architecture. This architecture (introduced in [Fre95]) consists of a DPLL search-tree [DLL62] using a LOOKAHEAD procedure to determine a decision variable x_{decision} (see Algorithm 4.1). We refer to a look-ahead on literal l as assigning l to true and performing iterative unit propagation. If a conflict occurs during this unit propagation (the empty clause is generated), then l is called a *failed literal* - forcing l to be fixed on false. The resulting formula after a look-ahead on l is denoted by $\mathcal{F}(l = 1)$.

Algorithm 4.1 DPLL($\mathcal{F}$)

```
1: if $\mathcal{F}$ is empty then
2:     return satisfiable
3: else if empty clause $\in \mathcal{F}$ then
4:     return unsatisfiable
5: end if
6: $\langle \mathcal{F}, x_{\text{decision}} \rangle$ := LookAhead( $\mathcal{F}$ )
7: $\mathbf{B}$ := GetDirection( $x_{\text{decision}}$ )
8: if DPLL( $\mathcal{F}(x_{\text{decision}} \leftarrow \mathbf{B})$ ) = satisfiable then
9:     return satisfiable
10: end if
11: return DPLL( $\mathcal{F}(x_{\text{decision}} \leftarrow \neg\mathbf{B})$ )
```

Five subprocedures of the LOOKAHEAD procedure (see Algorithm 4.2) are common in most modern look-ahead SAT solvers:

- PRESELECT - In general, performing a look-ahead on all unfixed variables is very costly. Therefore, most look-ahead SAT solvers pre-select a subset (denoted by $\mathcal{P}$) of the unfixed variables in each node of the search-tree to enter the LOOKAHEAD procedure. This enhancement was introduced by Li *et al.* [LA97a]. In each node, variables are ranked based on their occurrences in binary and ternary clauses. Variables with the highest ranking are pre-selected. Some modifications to these pre-selection heuristics are discussed in Section 4.4.1.

Algorithm 4.2 LOOKAHEAD($\mathcal{F}$)

```
1:  $\mathcal{P}$ := PRESELECT( $\mathcal{F}$ )
2:  for each variable $x_i$ in $\mathcal{P}$ do
3:      $\mathcal{F}' := \mathcal{F}(x_i = 0)$
4:      if empty clause $\notin \mathcal{F}'$ and $\mathcal{F}'_2 \gg \mathcal{F}_2$ then
5:          $\mathcal{F}'$ := DOUBLELOOK( $\mathcal{F}'$ )
6:      end if
7:      $\mathcal{F}'' := \mathcal{F}(x_i = 1)$
8:      if empty clause $\notin \mathcal{F}''$ and $\mathcal{F}''_2 \gg \mathcal{F}_2$ then
9:          $\mathcal{F}''$ := DOUBLELOOK( $\mathcal{F}''$ )
10:     end if
11:     if empty clause $\in \mathcal{F}'$ and empty clause $\in \mathcal{F}''$ then
12:         return unsatisfiable
13:     else if empty clause $\in \mathcal{F}'$ then
14:         $\mathcal{F} := \mathcal{F}''$
15:     else if empty clause $\in \mathcal{F}''$ then
16:         $\mathcal{F} := \mathcal{F}'$
17:     else
18:         $H(x_i)$ = MIXDIFF( DIFF($\mathcal{F}$, $\mathcal{F}'$), DIFF($\mathcal{F}$, $\mathcal{F}''$) )
19:     end if
20: end for
21: return $x_i$ with greatest $H(x_i)$
```

- DOUBLELOOK - If during the look-ahead on a literal, many new binary clauses are created (denoted by $\mathcal{F}'_2 \gg \mathcal{F}_2$), the resulting formula is frequently unsatisfiable. The DOUBLELOOK procedure attempts to find a conflict in the resulted formula by performing additional look-aheads. Details on this subprocedure are presented in Section 4.4.2.

- DIFF - The look-ahead evaluation function (DIFF) used in march_dl is identical to the one used in march_eq: Newly created binary clauses and reduced equivalence clauses are counted in a weighted fashion. The resulting sum is used as an indicator of the size of the search-tree of the reduced formula. A higher sum suggests a smaller search-tree. For a full description we refer to [HDvZvM04].

- MIXDIFF - Combines the two DIFF numbers. Let $L :=$ DIFF$(\mathcal{F}, \mathcal{F}(x = 0))$ and $R :=$ DIFF$(\mathcal{F}, \mathcal{F}(x = 1))$ then MIXDIFF$(L, R) := 1024 \times LR + L + R$. Motivation for this formula is that an effective decision variable splits the formula into two small search-trees (realized by LR). The $L + R$ addition is used for tie-breaking purposes. The formula is used in most look-ahead SAT solvers (POSIT, kcnfs, march, and satz) and originates from [Fre95].

- GetDirection - Given a decision variable x_{decision}, it pays off (for look-ahead Sat solvers only on satisfiable instances) to choose wisely whether to first enter branch $\mathcal{F}(x_{\text{decision}} = 1)$ or branch $\mathcal{F}(x_{\text{decision}} = 0)$. The first is preferred if $\text{Diff}(\mathcal{F}, \mathcal{F}(x_{\text{decision}} = 1)) < \text{Diff}\ (\mathcal{F}, \mathcal{F}(x_{\text{decision}} = 0))$, otherwise the latter.

4.3 Pre-processor enhancements

The performance of look-ahead Sat solvers is highly related to the size of the formula: In general, large CNF's require much more solving time regardless the complexity of the underlying problem. Both other architectures are not very sensitive to this. Therefore, pre-processing (reducing the size of) the formula is essential for fast performance of a look-ahead Sat solver. march_dl simplifies the formula - like march_eq - by binary equivalence propagation, detection of failed literals and subsumption of clauses [HDvZvM04].

4.3.1 Root look-ahead

Unlike the other solvers participating in the Sat competitions, all march versions use a 3-Sat translator in the pre-processor. march_dl uses the same translator as the one used in march_eq [HDvZvM04]. In the pre-processor, most march versions perform an *iterative full look-ahead* procedure to reduce the formula. This procedure checks for all literals (full) whether unit propagation will result in a conflict. This is iteratively performed until no new failed literals are detected.

In march_eq, this procedure was executed *after* applying the 3-Sat translator. The resulted formula - after the iterative full look-ahead procedure - frequently contained many dummy (introduced by the translation) variables. By executing the procedure *before* the 3-Sat translator, generally, less dummy variables have to be used.

Table 4.1 shows some experimental results of the effects - on solving time and number of generated dummies - on performing a root look-ahead before or after the 3-Sat translator. This table gives quite an accurate illustration of the effect of this enhancement: If swapping the execution order of the root look-ahead and the translator results in less generated dummy variables, then less computational time is required. Otherwise, no difference is noticed in solving times too.

4.3.2 Ternary resolvents

While pre-processing a formula, many resolvents could be added. Addition of all possible resolvents will - in general - significantly increase the size of the problem. Even adding only all resolvents of length two in the preprocessing phase, will increase solving time in most cases. Therefore, adding resolvents in march_eq is restricted to all binary constraint resolvents [HDvZvM04], both in the pre-processor and in the actual solving phase.

Table 4.1 — Performance of march_dl and the number of used dummy variables by applying the root look-ahead before or after the 3-SAT translator.

	before 3-SAT		**after 3-SAT**	
Benchmarks	time(s)	#dummies	time(s)	#dummies
`random-unsat-250` (100)	0.52	0	0.52	0
`random-unsat-350` (100)	15.04	0	15.04	0
`connm-n600-d0.04-sat04-975`	406.52	148	406.52	148
`connm-n600-d0.04-sat04-978`	535.05	142	535.05	142
`connm-n600-d0.04-sat04-981`	220.78	141	220.78	141
`ezfact48-1`	**8.61**	1803	13.34	1850
`ezfact48-2`	**8.84**	1792	14.76	1846
`ezfact48-3`	**19.93**	1817	28.86	1857
`lksat-n1000-k3-l5-sat04-930`	26.47	0	26.47	0
`lksat-n1000-k3-l5-sat04-931`	25.41	0	25.41	0
`lksat-n1000-k3-l5-sat04-932`	6.97	0	6.97	0
`longmult8`	**35.62**	195	51.14	377
`longmult10`	**117.50**	246	140.72	471
`longmult12`	**218.92**	293	352.17	565
`philips`	292.81	0	292.81	0

In march_eq, we already implemented a prototype procedure adding some ternary resolvents. This procedure is now efficiently implemented in march_dl and adds - just after the 3-SAT translation - only ternary resolvents of a certain type: All ternary resolvents are added to the formula that could be created by resolving two ternary clauses:

$$(x_i \vee x_j \vee x_r) \otimes_{x_j} (x_i \vee \neg x_j \vee x_s) = (x_i \vee x_r \vee x_s) \qquad \lfloor 4.1$$

In this equation, $\otimes_{x_j}$ refers to the resolution operator on variable x_j. Notice that added ternary resolvents could be used to create other ternary resolvents using the same equation.

The motivation to add these resolvents is first observed in [BS92]. On uniform random 3-SAT formulae, their addition in the pre-processor reduces on average the computational costs by about 10%. We experimented with the addition of these resolvents on various structured benchmarks. Within our experimental domain, this addition appeared to have either a favorable influence or no influence at all regarding the required computation time.

Table 4.2 shows the number of ternary clauses after the 3-SAT translator (denoted by $\#T_{\text{trans}}$) and the number of ternary resolvents that - using (1) - could be added (denoted by $\#T_{\text{resolve}}$). The last two columns show the computational cost of march_dl with and without this addition. The table convincingly shows the usefulness of adding these ternary resolvents on a wide scale of benchmarks. In only one case the performance is slightly decreased. Since, on structured benchmarks, the far majority of the clauses has length two, this performance boost can be realized by the addition of relatively few clauses.

Table 4.2 — Performance of march_dl on several benchmarks with and without adding ternary resolvents during the pre-processing phase.

Benchmarks	$\#T_{\text{trans}}$	$\#T_{\text{resolve}}$	with	without
`random-unsat-250` (100)	1075	92.7	**0.52**	0.55
`random-unsat-350` (100)	1500	89.3	**15.04**	16.13
`connm-n600-d0.04-sat04-975`	7640	16840	**406.52**	> 2000
`connm-n600-d0.04-sat04-978`	7292	17908	**535.05**	> 2000
`connm-n600-d0.04-sat04-981`	7242	16888	**220.78**	> 2000
`ezfact48-1`	8086	4292	**8.61**	> 2000
`ezfact48-2`	8063	3969	**8.84**	> 2000
`ezfact48-3`	8104	4596	**19.93**	> 2000
`lksat-n1000-k3-l5-sat04-930`	3629	1080	**26.47**	392.91
`lksat-n1000-k3-l5-sat04-931`	3602	715	**25.41**	875.20
`lksat-n1000-k3-l5-sat04-932`	3634	926	**6.97**	233.51
`longmult8`	1638	48	**35.62**	37.48
`longmult10`	2142	57	117.50	**111.85**
`longmult12`	2670	62	**218.92**	233.85
`philips`	896	0	292.81	292.81

4.4 Adaptive heuristics

Most heuristics used in look-ahead SAT solvers are heavily optimized towards fast performance on uniform random formulae. These heuristics are partly the cause of mediocre performance on structured instances. By developing heuristics that adapt towards each specific instance, we tried to do well 'across the board'.

4.4.1 Pre-selection heuristics

The main differences between the four march versions submitted to the SAT 2004 competition (march_001, march_007, march_eq_010, and march_eq_100) is the number of variables pre-selected to enter the LOOKAHEAD procedure. Each version pre-selects a fixed number of variables determined as a percentage of the original number of variables. The last suffix (_xxx) denotes this percentage. For instance, while solving a CNF with 1234 initial variables, march_eq_010 will pre-select 123 variables in each node of the DPLL search-tree to enter the LOOKAHEAD procedure. If, in a certain node, there are less than 123 variables unfixed, all remaining variables will be pre-selected. Hence, deeper in the search-tree relative more unfixed variables are pre-selected.

The motivation to use a different percentage in each of the submitted versions originates from the observation that the optimal percentage is benchmark dependent [HDvZvM04]. Therefore, we decided to use more dynamic pre-selection heuristics in march_dl. We also observed that - within our experimental domain - the optimal percentage was closely related to the frequency of detected failed literals: When relatively many failed literals were detected, higher percentages appeared optimal. Let $\#\text{failed}_i$ be the number of detected failed literals in node i. We tried to exploit the correlation mentioned above by using the average number of detected failed literals as an indicator for the maximum size of the pre-selected set in node n (denoted by $\mathcal{P}^n_{\max}$):

$$\mathcal{P}^n_{\max} := \mu + \frac{\gamma}{n} \sum_{i=1}^{n} \#\text{failed}_i \qquad [4.2$$

In the above, parameter μ refers to lower bound of $\mathcal{P}_{\max}$ in each node (namely when the average tends to zero) and γ is a parameter modeling the importance of failed literals. During small scale experiments on various structured and random instances, with $\mu := 5$ and $\gamma := 7$, resulted in favorable performance on most instances. Notice that these adaptive pre-selection heuristics are heavily influenced by the branching strategy - which in turn affects by these heuristics.

In most nodes $|\mathcal{P}| = \mathcal{P}_{\max}$. Only when the number of unfixed variables in the formula is smaller than $\mathcal{P}_{\max}$, then all variables are pre-selected and resulting in $|\mathcal{P}| < \mathcal{P}_{\max}$. It could happen that all variables in $\mathcal{P}$ are forced - due to the detection of failed literals - during the LookAhead procedure. In these cases the procedure is restarted with the reduced formula.

Table 4.3 — Performance of march_dl and three modified versions $\text{march\_dl}^*_{\text{xxx}}$ (in short $\text{m\_dl}^*_{\text{xxx}}$) with a constant number of pre-selected variables. Subscript xxx denotes the used percentage of the number of variables.

Benchmarks	march_dl	m_dl^*_{001}	m_dl^*_{010}	m_dl^*_{100}
`random-unsat-250` (100)	**0.52**	0.94	0.54	0.71
`random-unsat-350` (100)	**15.04**	33.62	15.80	25.31
`connm-n600-d0.04-sat04-975`	406.52	1150.86	**389.65**	584.96
`connm-n600-d0.04-sat04-978`	535.05	**517.55**	661.29	707.19
`connm-n600-d0.04-sat04-981`	220.78	814.78	**210.31**	291.60
`ezfact48-1`	8.61	**7.72**	8.62	8.67
`ezfact48-2`	8.84	**8.52**	9.02	9.05
`ezfact48-3`	19.93	**17.64**	19.78	19.89
`lksat-n1000-k3-l5-sat04-930`	26.47	39.18	**23.83**	52.85
`lksat-n1000-k3-l5-sat04-931`	25.41	40.77	**23.94**	47.50
`lksat-n1000-k3-l5-sat04-932`	6.97	9.87	**6.72**	11.65
`longmult8`	**35.62**	36.01	43.68	44.21
`longmult10`	117.50	**102.43**	107.65	111.87
`longmult12`	218.92	**130.54**	153.24	165.43
`philips`	292.81	**282.16**	432.84	441.83

The effect of using these adaptive pre-selection heuristics on the performance is shown in Table 4.3. As a reference, three columns are added with the computational costs of modified versions of march_dl with static percentages. Although the adaptive variant rarely results in the fastest performance; in general, its performance is relatively - compared to the references - close to optimal. Since these adaptive heuristics are still in an experimental phase, we expect to achieve even better results by further optimizing the settings.

4.4.2 Double look-ahead

The DoubleLook procedure (see Algorithm 4.3) checks whether a formula resulting from a look-ahead is unsatisfiable. It does so by performing additional unit-propagations. Since the computational costs of the DoubleLook procedure are high, it should not be called after every look-ahead. In the ideal case, one would only call it when the procedure would detect that the input formula is unsatisfiable. This could be done by an indicator expressing the usefulness of a DoubleLook call.

Li [Li99] suggests that the number of newly created binary clauses found during a look-ahead is an effective indicator whether or not to call the DoubleLook procedure: Many newly created binary clauses during a look-ahead increases the chance that DoubleLook will detect a conflicting formula. Li's solver satz calls DoubleLook if the number of new binary clauses in the reduced formula (after a look-ahead) is larger than a certain constant. We refer to this constant asΔ_{trigger}. In satz, $\Delta_{\text{trigger}} := 65$ is used.

Algorithm 4.3 DoubleLook($\mathcal{F}$)

```
1: $\mathcal{P}$ := PreSelect( $\mathcal{F}$ )
2: for each variable $x_i$ in $\mathcal{P}$ do
3:     $\mathcal{F}' := \mathcal{F}(x_i = 0)$
4:     $\mathcal{F}'' := \mathcal{F}(x_i = 1)$
5:     if empty clause $\in \mathcal{F}'$ and empty clause $\in \mathcal{F}''$ then
6:         return $\mathcal{F}'$
7:     else if empty clause $\in \mathcal{F}'$ then
8:         $\mathcal{F} := \mathcal{F}''$
9:     else if empty clause $\in \mathcal{F}''$ then
10:        $\mathcal{F} := \mathcal{F}'$
11:    end if
12: end for
13: return $\mathcal{F}$
```

Dubois and Dequen use a slight variation of the above setting in their solver kcnfs [DD]. Here, the DoubleLook procedure is triggered when the number of new binary clauses is larger than $\Delta_{\text{trigger}} := 0.175n + 5$ (n refers to the initial number of variables). Both settings of Δ_{trigger} result from optimizing this parameter towards the performance on uniform random 3-Sat formulae. On these instances they appear quite effective. However, on structured formulae - industrial and crafted - these settings are far from optimal: On some families, practically none of the look-aheads generate enough new binary to trigger DoubleLook. Even worse, on many other structured instances both Δ_{trigger} settings result in a pandemonium of calls of the DoubleLook procedure, which will come down hard on the computational costs to solve these instances.

To counter these unfavorable effects, march_dl uses a more abstract parameter $\mathrm{DL}_{\mathrm{success}}$. This parameter refers to the aimed ratio of successful calls on the DoubleLook procedure. A DoubleLook call is successful if it detects that the input formula is unsatisfiable. For instance, $\mathrm{DL}_{\mathrm{success}} := \frac{3}{4}$ means that the solver tries to call the DoubleLook procedure in such a way that three out of four calls are successful.

We achieve this success ratio by using a dynamic $\Delta_{\mathrm{trigger}}$ parameter: Depending on the success of a certain DoubleLook call, $\Delta_{\mathrm{trigger}}$ is updated using $\mathrm{DL}_{\mathrm{update}}$ (see Equation 4.3). Under the assumption that the number of newly created binary clauses is an effective indicator for the success probability of a DoubleLook call, it is expected that $\Delta_{\mathrm{trigger}}$ will converge to a certain value. In practice either it stabilizes or $\Delta_{\mathrm{trigger}}$ reaches early in the solving phase a high value such that DoubleLook is never triggered in a later stadium.

$$\mathrm{DL}_{\mathrm{update}} := \begin{cases} -\dfrac{1 - \mathrm{DL}_{\mathrm{success}}}{\mathrm{DL}_{\mathrm{success}}} & \text{if } \textsc{DoubleLook}(\mathcal{F}) \text{ is successful} \\ 1 & \text{otherwise} \end{cases} \quad \lfloor 4.3$$

Experiments show that a wide range of settings of $\mathrm{DL}_{\mathrm{success}}$ (0.7 to 0.95) result in a similar (fast) performance. In march_dl we choose $\mathrm{DL}_{\mathrm{success}} := \frac{9}{10}$. We initialized $\Delta_{\mathrm{trigger}} := 0.1n$. A full description with large-scale experiments to analyze and explain the above parameters and the behavior of the evolving sequence of $\Delta_{\mathrm{trigger}}$, will be the subject of Chapter 5.

Table 4.4 — Four different settings of the parameter $\Delta_{\mathrm{trigger}}$ implemented in march_dl: (a) adaptive (march_dl); (b) $\Delta_{\mathrm{trigger}} := 65$ (satz); (c) $\Delta_{\mathrm{trigger}} := 0.175\,n + 5$ (kcnfs); and (d) turning it OFF, so $\Delta_{\mathrm{trigger}} := \infty$.

Benchmarks	adaptive	à la satz	à la kcnfs	OFF
`random-unsat-250` (100)	0.52	**0.51**	0.52	0.59
`random-unsat-350` (100)	15.04	**14.68**	14.88	17.94
`connm-n600-d0.04-sat04-975`	**406.52**	823.37	815.69	517.41
`connm-n600-d0.04-sat04-978`	**535.05**	1808.77	1816.78	1205.27
`connm-n600-d0.04-sat04-981`	**220.78**	1149.71	1112.08	729.69
`ezfact48-1`	**8.61**	111.04	9.30	10.25
`ezfact48-2`	**8.84**	117.33	10.86	10.96
`ezfact48-3`	**19.93**	223.86	22.21	21.29
`lksat-n1000-k3-l5-sat04-930`	**26.47**	50.49	26.81	32.63
`lksat-n1000-k3-l5-sat04-931`	**25.41**	47.61	25.48	31.00
`lksat-n1000-k3-l5-sat04-932`	**6.97**	14.97	7.04	8.41
`longmult8`	35.62	73.38	34.83	**34.68**
`longmult10`	117.50	231.99	120.42	**111.67**
`longmult12`	218.92	241.80	207.14	**198.12**
`philips`	218.92	239.47	218.31	**215.87**

Table 4.4 shows the performance (in seconds) of four different approaches: (1) Our proposed adaptive heuristics; (2) the one used in satz; (3) the one used in kcnfs; and (4) no DOUBLELOOK at all (used in march_eq). The adaptive heuristics are the best option within our experimented domain. The down sides of the heuristics used in satz and kcnfs are clearly visible: On average the OFF setting performs better than both these random-instance-motivated methods.

4.5 Local branching

The look-ahead evaluation heuristic H (see Algorithm 4.2) has an unfavorable effect: The selected decision variables only have a high MIXDIFF in common. On structured instances, decision variables could be scattered all over the structure. For example, this could make it difficult to resolve local conflicts.

By branching only on variables occurring in reduced clauses, we try to counter this effect. Clearly, this is not applicable for the first node, because the initial formula has no reduced clauses. We refer to the above branching strategy as *local branching*. Recall that march_dl uses a 3-SAT translator in the pre-processing, thus clauses are either binary or ternary. So, local branching in this special case means that march_dl only branches on variables that occur in binary clauses that originated from ternary clauses in the initial formula.

This new branching strategy is realized by modifying the PRESELECT procedure: Instead of performing the pre-selection heuristics on the whole formula in a certain node, we discard all clauses that also occur in the initial formula (denoted by $\mathcal{F}_{\text{initial}}$). So, only variables occurring in reduced clauses are pre-selected. The resulted procedure is called LOCALPRESELECT and is shown in Algorithm 4.4. Notice that LOCALPRESELECT does not only pre-selects different variables to enter the look-ahead phase (compared to PRESELECT), it also could select less variables: The number of variables occurring in reduced clauses in the formula of node n could be smaller the $\mathcal{P}^{n}_{\max}$.

Algorithm 4.4 LOCALPRESELECT($\mathcal{F}$)

```
1: $\mathcal{F}_{\text{reduced}} := \mathcal{F} \setminus \mathcal{F}_{\text{initial}}$
2: if $\mathcal{F}_{\text{reduced}}$ is empty then
3:     $\mathcal{F}_{\text{initial}} := \mathcal{F}$
4:     restart
5: end if
6: return PRESELECT( $\mathcal{F}_{\text{reduced}}$ )
```

On some families - satisfiable and unsatisfiable - using local branching resulted in large speed-ups. Two examples of this kind are (1) the `ferry` family (all satisfiable) contributed by Maris to the SAT 2003 competition [LS03] and (2) the `homer` family (all unsatisfiable) contributed by Aloul to the SAT 2002 competition [SLH05]. Table 4.5 shows the performance of the march versions submitted to the SAT 2004 and 2005 competition on small instances of these families. Clearly, march_dl is the only solver that - due to local branching -

solves these instances. On formulae where the new branching strategy did not realize such a speed-up, no significant performance gains or losses were noticed.

Table 4.5 — Performance of different march versions on instances of the `ferry` and `homer` families. The march_eq_xxx solvers are abbreviated as m_eq_xxx.

Benchmarks	march_dl	march_001	march_007	m_eq_010	m_eq_100
`ferry8.sat03-384`	**2.18**	> 2000	> 2000	> 2000	> 2000
`ferry8u.sat03-385`	**2.54**	> 2000	> 2000	> 2000	> 2000
`ferry9.sat03-386`	**1.70**	> 2000	> 2000	> 2000	> 2000
`ferry9u.sat03-387`	**1.37**	> 2000	> 2000	> 2000	> 2000
`fpga10-11-uns-rcr`	**118.93**	> 2000	> 2000	> 2000	> 2000
`fpga10-12-uns-rcr`	**136.21**	> 2000	> 2000	> 2000	> 2000
`fpga10-13-uns-rcr`	**154.78**	> 2000	> 2000	> 2000	> 2000

The original motive to implement local branching was to increase the chance of finding *autarkies* [Kul00] - partial assignments that satisfy all clauses that they “touch”. The remaining formula, after removing all satisfied clauses by an autarky, is satisfiability equivalent to the original formula. A pure literal is an example of an autarky. Especially on unsatisfiable benchmarks, detection of autarkies is useful: Unsatisfiability of the remaining formula yields unsatisfiability of the original formula, resulting in a smaller search-tree.

This aspect is also shown in Algorithm 4.4: Whenever the formula $\mathcal{F}$ in a node (except from the rootnode) does not contain reduced clauses - compared to the initial formula $\mathcal{F}_{\text{initial}}$ - an autarky in detected. So, $\mathcal{F}$ is satisfiability equivalent to $\mathcal{F}_{\text{initial}}$. To reduce the computational cost to solve $\mathcal{F}_{\text{initial}}$, we restart the DPLL procedure with $\mathcal{F}$. Although many autark assignments were found in various families, none of these detections resulted in significant performance gains. This disappointing result could be explained by the fact that nearly all autarkies were found on satisfiable instances.

4.6 Results and conclusions

Five enhancements are presented which were developed to increase the overall performance of march. All were illustrated using some experimental results showing their contribution of reducing the computational costs. For comparisons with other solvers we refer to the SAT competition pages[15.].

The resulted version - march_dl - participated in the SAT 2005 competition. It was awarded with three silver and two bronze medals [LS06]. Unlike previous competitions, march_dl performed relatively good on industrial benchmarks too: It ended midway in the final ranking in that category. However, much progress is still required to make look-ahead based solvers competitive on these kind of structured instances.

[15.] `www.satcompetition.org`

Everything should be made as simple as possible, but not simpler.
Albert Einstein

5 Adaptive Heuristics[*]

Let us continue the Maniac Mansion example where we left off in the previous chapter. You need to find the front door key to get out of the mansion. You decided that a nerd is probably the best avatar to search the house thanks to his gadgets and engineering skills. The question we will try to answer is when to use his gadgets.

First and foremost, the use of gadgets is recommended somewhere during the search. Otherwise, another avatar would probably be a better option. To maximize the gain realized by gadgets, we therefore should use them right from the beginning. Future use should be motivated by actual gain.

Besides owning some gadgets, the nerd also has some knowledge about the appropriate situation to use them. For instance, with binoculars at hand, he can search beyond normal sight. He can look "long distance" where otherwise a search is required. So, by using binoculars in a large room you may save quite some time. However, in a small room you do not need to enhance your vision. In fact, binoculars will be very impractical.

Whether or not to use a gadget does generally not depend on the properties of the mansion, but on the properties of each room. E.g. in a mansion with spacious rooms, you probably want to use binoculars. But if most rooms are filled with boxes that block your sight, that might not be such a good idea.

Based on these considerations, we advice – regardless the layout of the mansion – to fully use gadgets in the beginning of the search. And, as soon as a gadget appears to be completely worthless, to radically decrease the frequency to use it. Depending on the knowledge when to use a gadget, while considering the proposed frequency, determines whether a gadget should be used. To guaranty that gadgets are used every once in a while, the frequency is gradually raised in time.

Using this advice, we developed an adaptive algorithm to control the application of an additional reasoning technique (gadget) called the DoubleLook procedure. Alternative static heuristics determine the usefulness of this procedure based on properties of the formula (mansion).

[*]This chapter is based on: Marijn J.H. Heule and Hans van Maaren. Effective Incorporation of Double Look-Ahead Procedures. SAT 2007 Springer LNCS **4501** (2007), pp 258-271.

5.1 Introduction

Nowadays state-of-the-art satisfiability (SAT) solving shows two main solving architectures: Conflict-driven and look-ahead driven. As tuned by the SAT competitions over the last years these two architectures seem to perform in an almost complementary way. The conflict-driven solvers dominate the so called industrial flavored problems (industrial category) while the look-ahead architecture dominates on random problems and problems with an intrinsic combinatorial hardness (part of crafted category). This chapter deals with an engineering type of solver optimization with respect to one of the ingredients of look-ahead SAT solving.

The look-ahead architecture of (SAT) solvers has two important features: (1) It selects decision variables that result in a balanced search-tree; and (2) it detects failed literals to reduce the size of the search-tree. Many enhancements have been proposed for this architecture in recent years. One of the enhancements for look-ahead SAT solvers is the DOUBLELOOK procedure, which was introduced by Li [Li99]. The usefulness of this procedure is straight forward: By also performing look-ahead on a second level of propagation, more failed literals could be detected, resulting in an even smaller search-tree.

By always performing additional look-aheads on the reduced formula, the computational costs rise drastically. One would like to restrict this enhancement in such a way that the overall computational time will decrease. Early implementations rely on restrictions based on static heuristics. Although these implementations significantly reduce the time to solve `random 3-`SAT formulae, they yield a clear performance slowdown on many structured instances.

We designed an algorithm for the DOUBLELOOK procedure that adapts towards the (reduced) CNF formula. Our algorithm has some key advantages: 1) Existing DOUBLELOOK implementations require only minor changes; 2) only one magic constant is used, which makes it easy to optimize the algorithm for a specific solver; and 3) this algorithm appears to outperform existing approaches.

In this chapter, Section 5.2 provides a general overview of the look-ahead architecture and zooms in on the DOUBLELOOK procedure. Section 5.3 deals with static heuristics for this procedure and their effect on the performance. Our algorithm is introduced in Section 5.4 together with an alternative by Li. It offers detailed descriptions and motivates the decisions made regarding its design. Section 5.5 illustrates the usefulness and the behavior of the algorithm by experimental results and adaptation plots. Finally, we draw some conclusions in Section 5.6.

5.2 Preliminaries

The look-ahead SAT architecture (introduced in [Fre95]) consists of a DPLL search-tree [DLL62] using a LOOKAHEAD procedure to reduce the formula and to determine a decision variable x_{decision}(see Algorithm 5.1). We refer to a look-ahead on literal l as assigning l to true and performing iterative unit propagation. If a conflict occurs during this unit propagation (the empty clause is generated), then l is called a *failed literal* - forcing l to be fixed on false. The resulting formula after a look-ahead on l is denoted by $\mathcal{F}(l = 1)$.

Algorithm 5.1 DPLL($\mathcal{F}$)

```
1:  if $\mathcal{F}$ is empty then
2:      return satisfiable
3:  else if empty clause $\in \mathcal{F}$ then
4:      return unsatisfiable
5:  end if
6:  $\langle \mathcal{F}, x_{\text{decision}} \rangle$ := LookAhead( $\mathcal{F}$ )
7:  $\mathbf{B}$ := GetDirection( $x_{\text{decision}}$ )
8:  if DPLL( $\mathcal{F}(x_{\text{decision}} \leftarrow \mathbf{B})$ ) = satisfiable then
9:      return satisfiable
10: end if
11: return DPLL( $\mathcal{F}(x_{\text{decision}} \leftarrow \neg\mathbf{B})$ )
```

The effectiveness of the LOOKAHEAD procedure (see Algorithm 5.2) depends heavily on the LOOKAHEADEVALUATION function which should favor variables that yield a small and balanced search-tree. Detection of failed literals could further reduce the size of the search-tree. Additionally, several enhancements are developed to boost the performance of SAT solvers based on this architecture.

One of these enhancements is the PRESELECT procedure, which preselects a subset of the variables (denoted by $\mathcal{P}$) to enter the look-ahead phase. By performing look-ahead only on variables in $\mathcal{P}$ the computational costs of the LOOKAHEAD procedure are reduced. However, this may result in less effective decision variables and less detected failed literals. All three solvers discussed in this chapter, march_dl, satz, and kcnfs, use a PRESELECT procedure. Yet, their implementation of this procedure is different.

Another enhancement is the DOUBLELOOK procedure (see Algorithm 5.3), which was introduced by Li [Li99]. This procedure checks whether a formula resulting from a look-ahead on l is unsatisfiable - it detects l as a failed literal by performing additional look-aheads on the reduced formula. Since the computational costs of these extra unit-propagations are high, this procedure should not be performed on each reduced formula. In the ideal case, one would want to apply it only when the reduced formula could be detected to be unsatisfiable. This requires an indicator expressing the likelihood to observe a conflict.

Let $\mathcal{F}_2^*$ denote the set of binary clauses in the reduced formula. Li [Li99] suggests that the number of newly created binary clauses (denoted by $|\mathcal{F}_2^* \setminus \mathcal{F}|$) in the reduced formula is an effective indicator whether or not to perform additional look-aheads: If *many* new binary clauses are created during the look-ahead on a literal, the resulting formula is often unsatisfiable. In Algorithm 5.3 the additional look-aheads are triggered when the number of newly created binary clauses exceeds the value of Δ_{trigger}. The optimal value of this parameter is the main topic of this chapter.

Algorithm 5.2 LookAhead($\mathcal{F}$)

```
1: $\mathcal{P}$ := Preselect( $\mathcal{F}$ )
2: for all variables $x_i \in \mathcal{P}$ do
3:     $\mathcal{F}'$ := DoubleLook( $\mathcal{F}(x_i = 0)$, $\mathcal{F}$ )
4:     $\mathcal{F}''$ := DoubleLook( $\mathcal{F}(x_i = 1)$, $\mathcal{F}$ )
5:     if empty clause $\in \mathcal{F}'$ and empty clause $\in \mathcal{F}''$ then
6:         return $\langle \mathcal{F};\ * \rangle$
7:     else if empty clause $\in \mathcal{F}'$ then
8:         $\mathcal{F} := \mathcal{F}''$
9:     else if empty clause $\in \mathcal{F}''$ then
10:        $\mathcal{F} := \mathcal{F}'$
11:    else
12:        H($x_i$) = LookAheadEvaluation( $\mathcal{F}$, $\mathcal{F}'$, $\mathcal{F}''$ )
13:    end if
14: end for
15: return $\langle \mathcal{F};\ x_i$ with greatest H($x_i$) $\rangle$
```

5.3 Static Heuristics

The DoubleLook procedure has been implemented in two look-ahead Sat solvers. Initially, Li proposed a static value for Δ_{trigger} [Li99]: In the first implementation in `satz` the DoubleLook procedure was triggered using $\Delta_{\text{trigger}} := 65$. (The latest version of `satz` uses a dynamic algorithm which will be discussed in the next section.) Dubois and Dequen use a variation in their solver kcnfs [DD]: In their implementation, the DoubleLook procedure is triggered depending on the original number of variables (denoted by n): $\Delta_{\text{trigger}} := 0.18n$.

Both settings of Δ_{trigger} result from optimizing this parameter towards the performance on `random 3-`Sat formulae. On these instances they appear quite effective. However, on structured formulae - industrial and crafted - these settings are far from optimal: On some families, practically none of the look-aheads generate enough new binary clauses to trigger additional look-aheads. Even worse, on many other instances both Δ_{trigger} settings result in a pandemonium of additional look-aheads, which come down hard on the computational costs.

We selected a set of benchmarks from a wide range of families to illustrate these effects. We generated 20 `random 3-`Sat formulae with 350 variables with

Algorithm 5.3 DOUBLELOOK($\mathcal{F}^*$, $\mathcal{F}$)

```
if empty clause $\in \mathcal{F}^*$ then
    return $\mathcal{F}^*$
end if
if $|\mathcal{F}_2^* \setminus \mathcal{F}| > \Delta_{\text{trigger}}$ then
    for all variables $x_i \in \mathcal{P}$ do
        $\mathcal{F}' := \mathcal{F}^*(x_i = 0)$
        $\mathcal{F}'' := \mathcal{F}^*(x_i = 1)$
        if empty clause $\in \mathcal{F}'$ and empty clause $\in \mathcal{F}''$ then
            return $\mathcal{F}'$
        else if empty clause $\in \mathcal{F}'$ then
            $\mathcal{F}^* := \mathcal{F}''$
        else if empty clause $\in \mathcal{F}''$ then
            $\mathcal{F}^* := \mathcal{F}'$
        end if
    end for
end if
return $\mathcal{F}^*$
```

1491 clauses (10 satisfiable and 10 unsatisfiable formulae) and used 10 `random 3color` instances from the SAT 2002 competition [SLH05]. Additionally, we added some crafted and structured instances from various families:

- the `connamacher` family (generic uniquely extendible CSPs) contributed by Connamacher to SAT 2004 [Con04]. We selected $n = 600$, $d = 0.04$;
- the `ezfact` family (factoring problems) contributed by Pehoushek. We selected the first three benchmarks of 48 bits from SAT 2002 [SLH05];
- the `lksat` family, subfamily `l5k3` (random l-clustered k-SAT instances) contributed by Antont to SAT 2004 [LS04]. We selected all unsatisfiable instances;
- the `longmult` family contributed by Biere [BCCZ99]. These instances arise from bounded model checking. We used instances of size 8, 10 and 12;
- the `philips` family. An encoding of a multiplier circuit contributed by Heule to SAT 2004 [LS04];
- a `pigeon hole` problem (`phole10`) from `www.satlib.org`;
- the `pyhala braun` family (factoring problems) contributed by Pyhala Braun to SAT 2002 [SLH05]. We selected the `unsat-35-4-03` and `unsat-35-4-04`, the two smallest instances from this family not solved during SAT 2002;
- the `stanion/hwb` family (equivalence checking problems) contributed by Stanion. We selected all three benchmarks of size 24 from SAT 2003 [LS03];
- SAT-encodings of `quasigroup` instances contributed by Zhang [ZS00] We selected the harder unsatisfiable instances - `qg3-9`, `qg5-13`, `qg6-12`, and `qg7-12`.

Besides the random instances, all selected benchmarks are unsatisfiable to realize relatively stable performances. On most these families, the performance of look-ahead SAT solvers is strong[16.] (compared to conflict-driven SAT solvers). We performed two tests: One that used constant numbers for Δ_{trigger} - analogue to early satz - and another used values depending on the original number of variables - analogue to kcnfs. For both tests we used the march_dl SAT solver[17.]. All experiments were performed on a system with an Intel 3.0 GHz CPU and 1 Gb of memory running on Fedora Core 4. The results of the first test are shown in Table 5.1 and 5.2, for the low and high values of Δ_{trigger}, respectively.

Table 5.1 — Performance of march_dl using various static (low) values for Δ_{trigger}.

family	0	10	30	65	100	150
`3color` (10)	118.69	39.91	**31.50**	62.87	67.96	70.42
`anton` (5)	276.74	269.00	184.73	119.99	80.31	**62.39**
`connamacher` (3)	5352.55	5407.50	4426.95	4373.89	4559.49	4852.63
`ezfact48` (3)	650.01	451.55	287.67	321.70	264.79	187.93
`longmult` (3)	886.51	578.08	452.34	278.93	**219.35**	255.99
`philips` (1)	595.43	547.54	391.43	323.97	**273.99**	306.71
`pigeon`(1)	246.62	**140.05**	141.65	141.45	140.53	140.37
`pyhala-braun`(2)	4000.0	3024.49	2415.46	2019.37	1481.09	1224.92
`quasigroup` (4)	2351.98	2102.62	1649.78	1437.39	1362.80	1327.79
`stanion` (3)	2102.21	1661.80	**941.59**	971.29	964.34	972.18
`random-sat` (10)	157.12	136.95	96.04	**71.01**	75.44	86.09
`random-uns` (10)	322.68	285.80	199.03	**143.04**	156.70	178.00

Table 5.2 — Performance of march_dl using various static (high) values for Δ_{trigger}.

family	250	400	600	850	1150	1500
`3color` (10)	67.26	70.26	70.24	70.49	72.21	73.52
`anton` (5)	64.09	73.28	75.02	75.07	77.22	78.98
`connamacher` (3)	4353.03	**2633.67**	2642.37	2861.83	4258.05	4099.12
`ezfact48` (3)	69.87	**47.54**	55.78	57.16	54.56	51.91
`longmult` (3)	272.15	291.85	249.99	243.81	278.86	303.99
`philips` (1)	313.98	317.23	320.84	325.41	328.31	336.90
`pigeon`(1)	140.61	141.01	140.86	141.38	142.36	142.73
`pyhala-braun`(2)	1145.64	941.32	607.76	577.75	449.59	**428.26**
`quasigroup` (4)	1225.14	1011.26	849.64	507.18	455.84	**358.97**
`stanion` (3)	968.60	963.49	985.46	983.51	988.12	997.59
`random-sat` (10)	92.53	92.24	93.55	93.20	92.33	91.71
`random-uns` (10)	186.74	187.64	187.72	189.34	190.04	190.43

[16.] based on the results of the SAT competitions, see `http://www.satcompetition.org`
[17.] available from `http://www.st.ewi.tudelft.nl/sat/`

Recall that satz uses $\Delta_{\text{trigger}} := 65$ - as a result of experiments on `random 3-SAT` instances. As expected, setting $\Delta_{\text{trigger}} := 65$ boosts performances on this family. However, instances from the `pyhala-braun` and `quasigroup` are hard to solve with this parameter setting: On these families the computational time can be reduced by 80% by changing the setting to $\Delta_{\text{trigger}} := 1500$. In general, we observe that a parameter setting which results in optimal performance for a specific family, yields far-from-optimal performances on other families.

Table 5.3 offers the results of the second test. On `random 3-SAT` optimal performance is realized by $\Delta_{\text{trigger}} := .20n$: Indeed close to the setting used in kcnfs. However, none of the parameter settings result in close-to-optimal performances on all families. Moreover, the optimal performances on the families `3color`, `connamacher`, and `quasigroup` measured during the first test are about twice as fast as the optimal performances of the second test. So, all parameter settings used in the second test are far from optimal - at least for these families.

Table 5.3 — Performance of march_dl using various static values for Δ_{trigger}. These static values are based on the original number of variables (denoted by n).

family	.05 n	.10 n	.15 n	.20 n	.25 n	.30 n
`3color` (10)	**59.08**	67.98	70.19	67.08	68.06	65.87
`anton` (5)	146.13	83.24	62.67	**59.40**	64.19	67.15
`connamacher` (3)	4627.01	**4387.70**	4392.15	5078.09	4841.21	4807.81
`ezfact48` (3)	324.14	202.17	61.19	50.75	**43.85**	47.64
`longmult` (3)	**205.46**	247.89	308.71	285.71	265.31	267.09
`philips` (1)	288.72	**285.43**	311.09	312.46	323.28	311.15
`pigeon`(1)	158.59	147.60	**142.02**	142.99	143.96	142.15
`pyhala-braun`(2)	1173.64	1095.74	753.08	590.00	546.79	**484.66**
`quasigroup` (4)	1473.65	1201.45	1035.91	1069.18	951.36	**837.54**
`stanion` (3)	1885.25	1110.04	**938.94**	949.83	956.62	973.57
`random-sat` (10)	118.50	88.57	72.86	**70.61**	71.55	75.97
`random-uns` (10)	254.60	185.96	155.18	**142.56**	150.69	165.46

5.4 Adaptive DoubleLook

We developed an adaptive algorithm to control the DOUBLELOOK procedure. This algorithm updates Δ_{trigger} after each look-ahead in such fashion, that it adapts towards the characteristics of the (reduced) formula. This section deals with the decisions made regarding the algorithm. First and foremost - for both elegance and practical testing - we focused on using only one magic constant.

The algorithm has three components: (i) The Δ_{trigger} initial value, (ii) an increment strategy TRIGGERINCREASE and (iii) a decrement strategy TRIGGERDECREASE to update Δ_{trigger}. Both strategies consist of two parts: The location within the DOUBLELOOK procedure and the size of the update value.

Regarding the first component: An effective initial value for Δ_{trigger} is probably as hard to determine as an effective global value for this parameter. Therefore, the algorithm should work on many initial values - even on zero, the most costly value at the root node. Hence our decision to initialize $\Delta_{\text{trigger}} := 0$.

The first aspect of the increment strategy is rather straight-forward: Assuming a strong correlation between the value of Δ_{trigger} and the detection of a conflict by the DoubleLook procedure, Δ_{trigger} should always be increased when the procedure fails to meet this objective. Algorithm 5.4 shows an adaptive variant of the DoubleLook procedure with the increment strategy located at line 17, the first position following a failure.

The largest reasonable increment of Δ_{trigger} appears to make this parameter equal to the number of newly created binary clauses: Since no conflict was observed, Δ_{trigger} should be at least the number of new binary clauses ($|\mathcal{F}_2^* \setminus \mathcal{F}|$) - which would have prevented the additional computational costs. The smallest value of the increment is a value close to zero and would result in a slow adaptation. The optimal value will probably be somewhere in between. We prefer a radical adaptation. For this reason we use the largest reasonable value:

$$\textsc{TriggerIncrease}() \ : \ \Delta_{\text{trigger}} := |\mathcal{F}_2^* \setminus \mathcal{F}| \tag{5.1}$$

Algorithm 5.4 AdaptiveDoubleLook($\mathcal{F}^*$, $\mathcal{F}$)

```
1: if empty clause $\in \mathcal{F}^*$ then
2:     return $\mathcal{F}^*$
3: end if
4: if $|\mathcal{F}_2^* \setminus \mathcal{F}| > \Delta_{\text{trigger}}$ then
5:     for all variables $x_i \in \mathcal{P}$ do
6:         $\mathcal{F}' := \mathcal{F}^*(x_i = 0)$
7:         $\mathcal{F}'' := \mathcal{F}^*(x_i = 1)$
8:         if empty clause $\in \mathcal{F}'$ and empty clause $\in \mathcal{F}''$ then
9:             TriggerSuccess( )
10:            return $\mathcal{F}'$
11:        else if empty clause $\in \mathcal{F}'$ then
12:            $\mathcal{F}^* := \mathcal{F}''$
13:        else if empty clause $\in \mathcal{F}''$ then
14:            $\mathcal{F}^* := \mathcal{F}'$
15:        end if
16:    end for
17:    TriggerIncrease( )
18: else
19:    TriggerDecrease( )
20: end if
21: return $\mathcal{F}^*$
```

Within the DOUBLELOOK procedure, two events could suggest that Δ_{trigger} should be decreased[18.] : (1) The detection of a conflict and (2) the number of newly created binary clauses is less than Δ_{trigger}. The first event seems the most logical: If the DOUBLELOOK procedure detects a conflict, this is a strong indication that a slightly decreased Δ_{trigger} could increase the number of detected failed literals by this procedure. However, this may result in a deadlock situation: The increment strategy could update Δ_{trigger} such that no additional look-ahead will be executed, thereby making it impossible to decrease this parameter.

Placing the decrement strategy after the second event would guarantee that additional look-aheads will be executed every once in a while. Assuming that the computational time could diminish on all benchmarks by the DOUBLELOOK procedure, then this location (Algorithm 5.4 line 19) seems a more appealing choice.

How much should Δ_{trigger} be decreased if after a look-ahead the number of newly created binary clauses is less than this parameter? It seems hard to provide a motivated answer for this question. Therefore, we decided to obtain an effective value for the decrement using experiments.

These experiments were based on two considerations: First, the tests on static heuristics (see Section 5.3) showed that effective parameter settings for Δ_{trigger} ranged from 10 to 1500. Therefore, the decrement should not be absolute but relative. So, it should be of the form $\Delta_{\text{trigger}} := c \times \Delta_{\text{trigger}}$ for some $c \in [0, 1]$.

Second, the size of preselected set $\mathcal{P}$ could vary significantly over different nodes. Therefore, the maximum decrement of Δ_{trigger} in each node depends on the size of $\mathcal{P}$. We believe this dependency is not favorable, so we decided to "neutralize" it. Notice that at most $2|\mathcal{P}|$ times in each node Δ_{trigger} could be decreased. Now, let parameter $\text{DL}_{\text{decrease}}$ denote the maximum relative decrement of Δ_{trigger} in a certain node. Then, combining these considerations, the decrement strategy could be formulated as follows:

$$\text{TRIGGERDECREASE}() \ : \ \Delta_{\text{trigger}} := \sqrt[2|\mathcal{P}|]{\text{DL}_{\text{decrease}}} \times \Delta_{\text{trigger}} \qquad \lfloor 5.2$$

The "optimal" value for parameter $\text{DL}_{\text{decrease}}$ is discussed in Section 5.5.1.

The latest version of `satz` (2.15.2) also uses an adaptive algorithm: (*i*) It initializes $\Delta_{\text{trigger}} := .167n$; (*ii*) it increases the Δ_{trigger} using the same TRIGGERINCREASE() placed at the same location. The important difference lies in the location and size of (*iii*) the decreasing strategy: The algorithm realized the decrement at line 9 instead of line 19 of Algorithm 5.4 - so Δ_{trigger} is only reduced after a successful DOUBLELOOK call instead of slowly decrease after each look-ahead.

$$\text{TRIGGERSUCCESS}() \ : \ \Delta_{\text{trigger}} := .167n \qquad \lfloor 5.3$$

A drawback of this approach is that Δ_{trigger} could never be reduced to a value smaller than $.167n$ - although we noticed from the experiments on static heuris-

[18.] Δ_{trigger} could also be decreased after lines 12 and 14 of Algorithm 5.4: Each new forced literal on a second level of propagation increases the chance of hitting a conflict.

tics that significant smaller values are optimal in some cases (see Table 5.3). When a high value of Δ_{trigger} is optimal this approach might frequently alter between a relative low value ($\Delta_{\text{trigger}} := .167n$) and a relative high value ($\Delta_{\text{trigger}} := |\mathcal{F}_2^* \setminus \mathcal{F}|$) or result in the deadlock situation mentioned above.

5.5 Results

The adaptive algorithm as described above has been implemented in all look-ahead SAT solvers that contain a DOUBLELOOK procedure: march_dl, satz, and kcnfs. First, we show the effect of parameter $\text{DL}_{\text{decrease}}$ on the computational time. For this purpose, we use the modified march_dl. Second, the performance is compared between the original versions and the modified variants of satz and kcnfs. Third, the behavior of the algorithm is illustrated by adaptation plots. During the experiments we used the benchmarks as described in Section 5.3.

5.5.1 The magic constant

The only undetermined parameter of the adaptive algorithm is $\text{DL}_{\text{decrease}}$. The computational times resulting from various settings for this parameter are shown in Table 5.4. The data shows the effectiveness of the adaptive algorithm:

- Different settings for $\text{DL}_{\text{decrease}}$ result in comparable performances - generally close to the optimal values from the experiments using static heuristics.
- We observe that, for $\text{DL}_{\text{decrease}} := 0.85$, performances are realized for the `anton` and `philips` family that are nearly optimal, while on all the other families this setting outperforms all results using static heuristics.
- The optimal performances achieved by the adaptive heuristics are, on average, about 20% faster than those that are the result of static heuristics.

Table 5.5 shows the average values of Δ_{trigger} for various settings of $\text{DL}_{\text{decrease}}$. The average for each family is the mean of the averages of its instances, while for each instance the average is the mean of the averages over all nodes. Because these values are not very accurate, we present only rounded integers.

Parameter $\text{DL}_{\text{decrease}}$ seems to have little impact on these average values. Note that - except for `pyhala-braun` and `quasigroup` instances - the average values of Δ_{trigger} are very close to the optimal values shown in Tables 5.1 and 5.2. In Section 5.5.3 we provide a possible explanation for the two exceptions.

Table 5.4 — Influence of parameter $\text{DL}_{\text{decrease}}$ on the computational time.

family	.75	.80	.85	.90	.95	.99
`3color` (10)	25.77	**25.39**	25.60	28.98	32.79	44.36
`anton` (5)	69.22	67.66	64.99	**63.26**	63.60	66.41
`connamacher` (3)	2258.59	2723.14	**1742.62**	3038.68	2872.84	4431.91
`ezfact48` (3)	39.00	**35.18**	37.87	38.66	38.68	46.08
`longmult` (3)	**197.29**	197.70	203.03	210.12	241.75	258.90
`philips` (1)	307.22	288.10	286.31	**267.17**	280.81	299.71
`pigeon`(1)	**99.31**	99.77	103.47	110.91	113.81	115.28
`pyhala-braun`(2)	369.49	**365.51**	372.98	366.89	376.89	405.05
`quasigroup` (4)	162.38	161.95	157.24	154.59	**150.63**	162.01
`stanion` (3)	**941.94**	946.38	950.44	965.30	984.20	1010.71
`random-sat` (10)	70.04	70.71	**69.21**	69.95	69.74	74.32
`random-uns` (10)	147.40	147.19	**145.95**	148.30	149.17	159.90

Table 5.5 — Influence of parameter $\text{DL}_{\text{decrease}}$ on the average value of Δ_{trigger}.

family	.75	.80	.85	.90	.95	.99
`3color` (10)	23	24	25	28	33	42
`anton` (5)	129	134	141	162	176	220
`connamacher` (3)	538	575	589	527	462	292
`ezfact48` (3)	324	332	357	370	420	538
`longmult` (3)	76	78	80	90	100	127
`philips` (1)	99	102	107	110	117	142
`pigeon`	7	7	8	8	9	9
`pyhala-braun`(2)	105	108	112	117	127	148
`quasigroup` (4)	537	530	516	489	529	664
`stanion` (3)	21	22	23	25	29	36
`random-sat` (10)	57	58	62	67	77	98
`random-uns` (10)	57	59	62	67	78	98

5.5.2 Comparison

To test the general application of the adaptive algorithm, we also implemented it in both other SAT solvers that use a DOUBLELOOK procedure: satz and kcnfs. We modified the latest version of the source codes[19.]. All three components were made according to the proposed adaptive algorithm: First, initialization is changed to $\Delta_{\text{trigger}} := 0$. Second - only for kcnfs - a line is added to increase Δ_{trigger} when no conflict is detected. Analogue to the march_dl and satz, $\Delta_{\text{trigger}} := |\mathcal{F}_2^* \setminus \mathcal{F}|$.

The third modification is implemented slightly differently, because in satz and kcnfs the size of the pre-selected set $\mathcal{P}$ is computed "on the fly". Therefore, $\sqrt[2|\mathcal{P}|]{\text{DL}_{\text{decrease}}}$ would not be a constant value in each LOOKAHEAD procedure. As a workaround, we decided to use the average value of march_dl for $\sqrt[2|\mathcal{P}|]{\text{DL}_{\text{decrease}}}$ instead. Additionally, from satz the decrement strategy TRIGGERSUCCESS is removed. While using $\text{DL}_{\text{decrease}} := 0.85$, this average appeared approximately 0.9985, which was used for an alternative decrement strategy:

$$\text{TRIGGERDECREASE}() \;:\; \Delta_{\text{trigger}} := 0.9985 \times \Delta_{\text{trigger}} \qquad \lfloor 5.4$$

Notice that using value 1.0 instead of 0.9985 would drastically reduce the number of additional look-aheads, because Δ_{trigger} would never be decreased.

Table 5.6 — Comparison between performances of the original and the modified versions of satz, kcnfs and march_dl.

	satz		kcnfs		march_dl	
family	original	modified	original	modified	prelim	final
`3color` (10)	52.71	**36.91**	37.89	**27.88**	72.51	**25.60**
`anton` (5)	183.97	**123.16**	3433.39	**2382.96**	80.75	**64.99**
`connamacher` (3)	> 6000	> 6000	4707.51	**4705.23**	4134.85	**1742.62**
`ezfact48` (3)	39.96	**32.98**	> 6000	> 6000	54.22	**37.87**
`longmult` (3)	2411.36	**1582.85**	440.34	**413.19**	265.88	**203.03**
`philips` (1)	1126.38	**710.75**	750.75	**443.27**	428.52	**286.31**
`pigeon`(1)	**23.72**	24.12	43.39	**40.25**	145.38	**103.47**
`pyhala-braun`(2)	1247.46	**881.91**	644.84	**466.92**	380.57	**372.98**
`quasigroup` (4)	172.40	**171.54**	230.59	**173.86**	351.85	**157.24**
`stanion` (3)	**3657.49**	3810.53	**3834.31**	3863.13	993.89	**950.44**
`random-sat` (10)	93.82	**92.56**	**79.63**	80.33	91.63	**69.21**
`random-uns` (10)	**260.13**	266.81	139.67	**138.22**	189.75	**145.95**

The performances of the original and the modified versions of satz, kcnfs, and march_dl are shown in Table 5.6. The proposed adaptive algorithm generally outperforms the one in satz: On most instances from our test, the performance was improved up to 30%, while on the others only small losses were

[19.] For satz we used version 215.2 (with the adaptive algorithm) which is available at `http://www.laria.u-picardie.fr/~cli/satz215.2.c` and for kcnfs we used the version available at `http://www.laria.u-picardie.fr/~dequen/sat/kcnfs.zip` measured. Significant performance boosts are also observed in kcnfs, although the stanion/hwb instances are solved slightly slower. Since we did not optimize the magic constant, additional progress could probably be made.

The double look-ahead is the latest feature of march resulting in version march_dl. The preliminary version used has all features except the DoubleLookAhead procedure. The addition of this feature - using the proposed adaptive algorithm - boost the performance on the complete test set.

5.5.3 Adaptation plots

We selected four benchmarks (due to space limitations) to illustrate the behavior of the adaptive algorithm. For each benchmark, the first 10.000 (non-leaf) nodes of the DPLL-tree - using march_dl with $\Delta_{\text{trigger}} := .85$ - are plotted with a colored dot. Nodes are numbered in the (depth-first) order they are visited - so for the first few nodes their number equals their depth. The color is based on the depth of the node in the DPLL-tree. The horizontal axis shows the number of a certain node and the vertical axis shows the average value of parameter Δ_{trigger} in this node. These *adaptation plots* are shown in Figures 5.1, 5.2, 5.3 and 5.4.

In general, we observed that each family has its own kind of adaptation plot, while strong similarities between instances from different families were rare. For none of the tested instances Δ_{trigger} converged to a certain value, which is probably due to the design of the algorithm.

For half of the families, the value of Δ_{trigger} tends to be above average at nodes near the root of the search-tree and / or tends to be below average at nodes near the leafs (see Figures 5.1 and 5.4). For the other half of the families the opposite trend was noticed (see Figures 5.2 and 5.3).

Recall that for pyhala-braun and quasigroup instances the average value for Δ_{trigger} was much lower than the optimum based on static heuristics. Figure 5.4 offers a possible explanation: Notice that nodes near the root use $\Delta_{\text{trigger}} \approx 1100$ while on average nodes use $\Delta_{\text{trigger}} \approx 100$. Adaptation plots for quasigroup instances showed a similar gap. A low static value for Δ_{trigger} will probably result in many additional look-aheads at the nodes near the root which could ruin the overall performance.

5.6 Conclusions

We presented an adaptive algorithm to control the DoubleLook procedure, which uses - like the static heuristic - only one magic constant. The algorithm has been implemented in all look-ahead Sat solvers that use a DoubleLook procedure. As a result of this modification, all three solvers showed a performance improvement on a wide selection of benchmarks. On macro level we observed that for most instances this algorithm approximates the family specific "optimal" static strategy, while on micro level the algorithm adapts to the (reduced) formula in each node of the search-tree.

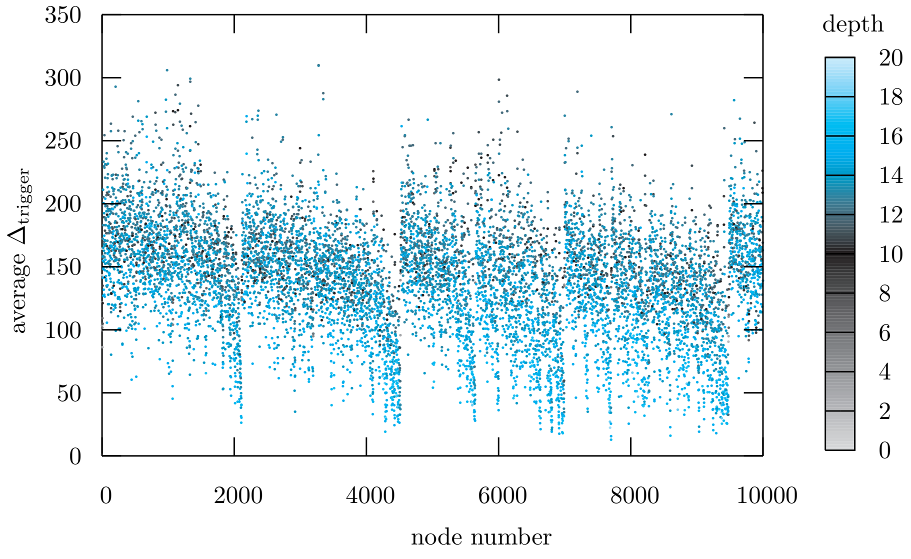

Figure 5.1 — Adaptation plot of the `philips` benchmark

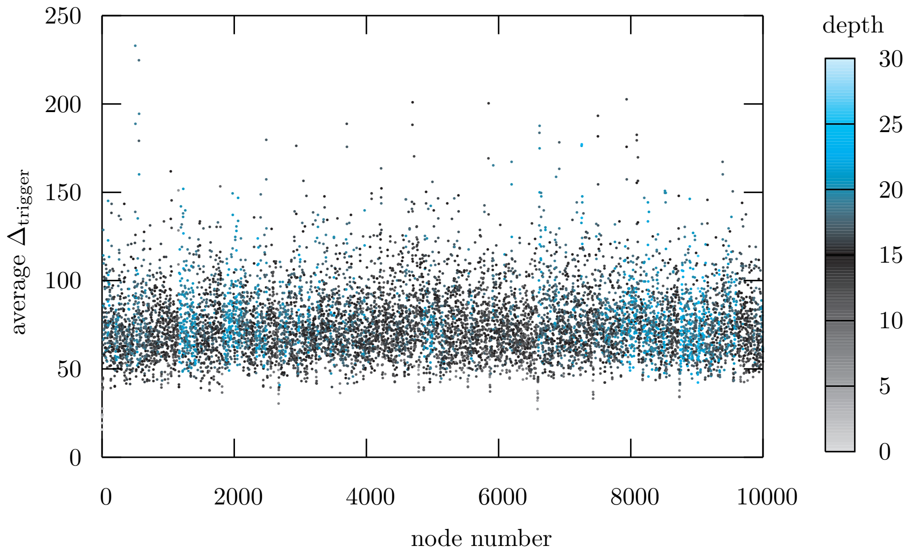

Figure 5.2 — Adaptation plot of a `random` 3-SAT formula with $n = 350$, $\rho = 4.26$

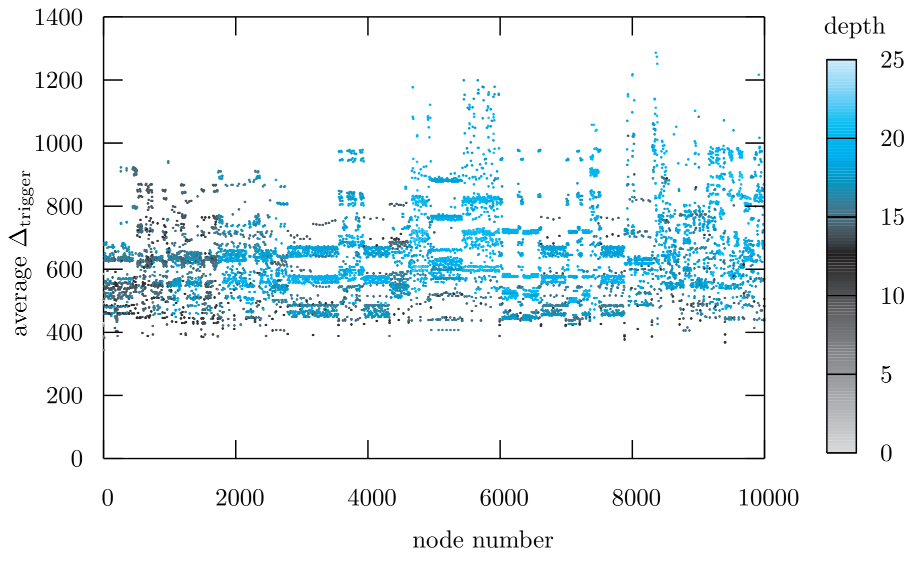

Figure 5.3 — Adaptation plot of `connm-ue-csp-sat-n600-d0.04-s1211252026`

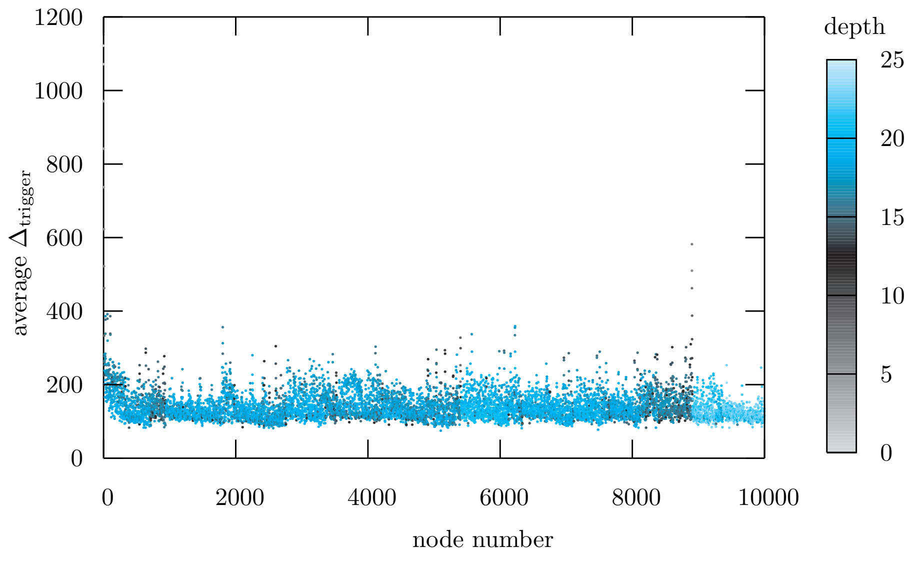

Figure 5.4 — Adaptation plot of `pyhala-braun-unsat-35-4-03`

I am not left,
I am not right,
I am straight-forward.
Rita Verdonk, dutch MP

6 Direction heuristics[*]

Intuition about where you should search can be a very powerful tool to solve problems. Recall the Maniac Mansion example and consider yourself a deity with perfect, flawless intuition. Searching for the key is now much easier: Just follow your intuition and if the key is not in the first room you look for it, there is none. Otherwise, just help yourself out.

However, there are no humans with perfect intuition. So, imagine that you are a woman: You have some intuition, albeit not perfect. This too could help the search, although less substantially. Again, the best place to look first is the room where your intuition leads you. But, if the key is not there, how to continue the search process?

Current search strategies use intuition to get to the first room and continue searching in the nearby rooms. Yet, this might not fully exploit the power of your intuition: Say, while walking to the first ”most intuitive” room, there was a crossing where your intuition was not clear. Turning left and turning right at that crossing were both appealing. We propose not to continue searching the nearby rooms, but to return to that particular crossing and this time take the opposite direction. Although such a strategy is more costly – the next preferred room may be quite some distance away – you may find the key much faster.

Applying this idea to Sat solving, we made two important contributions. First, we studied the intuition observed while solving different formulae (mansions) by different Sat solvers (women). Second, to capitalize on the observed intuition, we formalized a search strategy which selects the most appealing prior crossing to return to in case the key was not found in a certain room.

[*]This chapter is based on: Marijn J.H. Heule and H. van Maaren. *Whose side are you on? Finding solutions in a biased search-tree.* Submitted to the Journal on Satisfiability, Boolean Modeling and Computation.

6.1 Introduction

Various state-of-the-art satisfiability (SAT) solvers use *direction heuristics* to predict the sign of the decision variables: These heuristics choose, after the selection of the decision variable, which Boolean value is examined first. Direction heuristics are in theory very powerful: If always the correct Boolean value is chosen, satisfiable formulae would be solved without backtracking. Moreover, existence of perfect direction heuristics (computable in polynomial time) would prove that $\mathcal{P} = \mathcal{NP}$.

On some families these heuristics bias the location of solutions in the search-tree. Given a large family with many satisfiable instances, this bias can be measured on small instances. The usefulness of this depends on what we call the *bias-extrapolation property*: Given the direction heuristics of a specific solver, the observed bias on smaller instances extrapolates to larger ones. Notice that this notion depends on the *action* of a particular solver on the family involved: I.e. a solver with random direction heuristics may also satisfy the bias extrapolation property, but not in a very useful way - probably, there is no bias at all. In case the estimated bias shows a logical pattern, it could be used to consider a jumping strategy that adapts towards the distribution of the solutions. We refer to this strategy as *distribution jumping*.

Other jump strategies have been developed for SAT solvers. The most frequently used technique is the *restart strategy* [GSC97]: If after some number of backtracks no solution has been found, the solving procedure is restarted with a different decision sequence. This process is generally repeated for an increasing number of backtracks. This technique could fix an ineffective decision sequence. A disadvantage of restarts is its potential slowdown of performance on unsatisfiable instances, especially on look-ahead SAT solvers. However, conflict driven SAT solvers on average improve their performance using restarts.

Another jumping strategy is the *random jump* [Zha06]. Instead of jumping all the way to the root of the search-tree, this technique jumps after some backtracks to a random level between the current one and the root. This technique could fix a wrongfully chosen sign of some of the decision variables. By storing the subtrees that have been visited the performance on unsatisfiable instances is only slightly reduced for look-ahead SAT solvers.

Both these techniques are designed to break free from an elaborate subtree in which the solver is "trapped". Our proposed technique not only jumps out of such a subtree but also towards a subtree with a high probability of containing a solution. Unlike restart strategies, random jumping and distribution jumping only alter the signs of decision variables. Therefore, for look-ahead SAT solvers, the performance on unsatisfiable formulae does not influence costs, besides some minor overhead.

The outline is as follows: Section 6.2 introduces the direction heuristics used in complete (DPLL based) SAT solvers. In each step, these solvers select a decision variable (decision heuristics). Whether to first visit the *positive branch* (assigning the decision variable to true) or the *negative branch* (assigning the decision variable to false) is determined by these direction heuristics. These heuristics can heavily influence the performance on satisfiable benchmarks. In case conflict clauses are added, the performance on unsatisfiable instances is affected, as well.

Section 6.3 studies the influence of existing direction heuristics. More specific, we ask ourselves the question: Given a (complete) SAT solver and a benchmark family, is the *distribution of solutions* in the search-tree biased (not uniform)? A possible bias is caused by the direction heuristics used in the SAT solver. We offer some tools to visualize, measure and compare the possible bias for different SAT solvers on hard random k-SAT formulae. We selected this family of formulae because it is well studied and one can easily generate many hard instances for various sizes.

The bias we observe, can intuitively be explained: Since the direction heuristics discussed in this paper try to select the heuristically most satisfiable subtree (referred to as the *left branch*) first, a bias towards the left branches is expected and observed. Near the root of the search-tree, the considered (reduced) formulae are larger and more complex compared to those lower in the tree. Therefore, it is expected that the effectiveness of direction heuristics improves (and thus the bias towards the left branches increases) in nodes deeper in the tree - which is also observed.

Section 6.4 discusses the possibilities to capitalize on a given / observed bias. We focus on the observed bias of look-ahead SAT solvers on random k-SAT formulae. We developed a new jump strategy, called *distribution jumping*, which visits subtrees in decreasing (observed) probability of containing a solution. We show that using the proposed (generalized) order of visiting subtrees - compared to chronological order - results in a significant speed-up in theory.

We implemented this new technique in march_ks and Section 6.5 offers the results. These results show performance gains on random k-SAT formulae. On many satisfiable structured benchmarks improvements were observed, as well. Due to this technique, the look-ahead SAT solver march_ks won the satisfiable crafted family of the SAT 2007 competition. Finally some conclusions are drawn in Section 6.6.

6.2 Direction heuristics

All state-of-the-art complete SAT solvers are based on the DPLL architecture [DLL62]. This recursive algorithm (see Algorithm 6.1) first simplifies the formula by performing unit propagation (see Algorithm 6.2) and checks whether it hits a leaf node. Otherwise, it selects a decision variable x_{decision} and splits the formula into two subformulae where x_{decision} is forced - the *positive branch* (denoted by $\mathcal{F}(x_{\text{decision}} = 1)$) and the *negative branch* ($\mathcal{F}(x_{\text{decision}} = 0)$).

Algorithm 6.1 DPLL($\mathcal{F}$)

```
1: $\mathcal{F}$ := UnitPropagation( $\mathcal{F}$ )
2: if $\mathcal{F}$ is empty then
3:     return satisfiable
4: else if empty clause $\in \mathcal{F}$ then
5:     return unsatisfiable
6: end if
7: $x_{\text{decision}}$ := DecisionHeuristics( $\mathcal{F}$ )
8: $\mathbf{B}$ := DirectionHeuristics( $x_{\text{decision}}$ )
9: if DPLL( $\mathcal{F}(x_{\text{decision}} \leftarrow \mathbf{B})$ ) = satisfiable then
10:     return satisfiable
11: else
12:     return DPLL( $\mathcal{F}(x_{\text{decision}} \leftarrow \neg\mathbf{B})$ )
13: end if
```

Algorithm 6.2 UnitPropagation($\mathcal{F}$)

```
1: while $\mathcal{F}$ does not contain an empty clause and unit clause $y$ exists do
2:     satisfy $y$ and simplify $\mathcal{F}$
3: end while
4: return $\mathcal{F}$
```

Two important heuristics emerge for splitting: *Variable selection heuristics* (in the DecisionHeuristics procedure) and *direction heuristics* (in the DirectionHeuristics procedure). Variable selection heuristics aim at selecting a decision variable in each recursion step yielding a relatively small search-tree. Direction heuristics try to find a satisfying assignment as fast as possible by choosing which subformula - $\mathcal{F}(x_{\text{decision}} = 0)$ or $\mathcal{F}(x_{\text{decision}} = 1)$ - to examine first. We will refer to the *left branch* as the subformula that is visited first. Consequently, the *right branch* refers to the one examined later. In theory, direction heuristics could be very powerful: If one always predicts the correct direction, all satisfiable formulae will be solved in a linear number of decisions.

The search-tree of a DPLL-based Sat solver can be visualized as a binary search-tree. Figure 6.1 shows such a tree with decision variables drawn in the internal nodes. Edges show the type of each branch. A black leaf refers to an unsatisfiable dead end, while a white leaf indicates that a satisfying assignment has been found. An internal node is colored black in case both its children are black, and white otherwise. For instance, at depth 4 of this search-tree, 3 nodes are colored white. This means that at depth 4, 3 subtrees contain a solution.

Traditionally, Sat research tends to focus on variable selection heuristics. Exemplary of the lack of interest in direction heuristics is its use in the conflict-driven Sat solver minisat [ES03]: While this solver is the most powerful on a wide range of instances, it always branches negatively. An explanation for the effectiveness of this heuristic may be found in the general encoding of most (structural) Sat formulae. Also, these direction heuristics are more sophisticated then they appear: Choosing the same sign consequently is - even on random formulae - much more effective than a random selection [MvVW07].

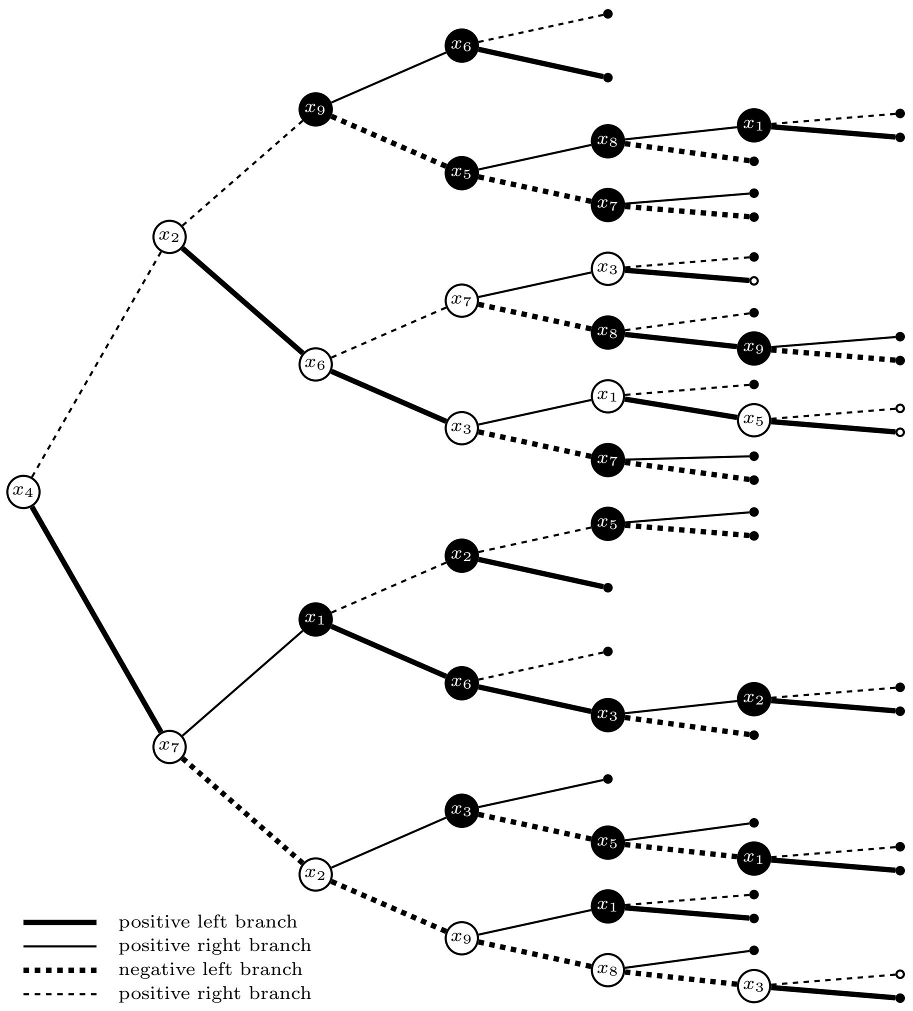

Figure 6.1 — Complete binary search-tree (DPLL) for a formula with nine variables ($x_1, \ldots, x_9$). The decision variables are shown inside the internal nodes. A node is colored black if all child nodes are unsatisfiable, and white otherwise. The type of edge shows whether it is visited first (left branch), visited last (right branch), its decision variable is assigned to true (positive branch), or its decision variable is assigned to false (negative branch).

Throughout this chapter, we will discuss only SAT solving techniques that do not add global constraints such as a conflict clauses. So, only chronological backtracking is considered. Also, given a SAT solver and a certain formula, the complete search-tree will always be identical: Visiting (leaf)nodes in a different order will not affect the binary representation as shown in Figure 6.1. In case the formula is satisfiable, the order in which (leaf)nodes are visited only influences the fraction of the search-space that has to be explored to find a (first) solution.

We will focus on the direction heuristics used in look-ahead SAT solvers. This choice is motivated by the strong performance of these solvers on random k-SAT formulae (the ones we selected for our experiments). Also, they do not add conflict clauses which disturb the pure binary search-tree representation. The additional reasoning used in look-ahead SAT solvers is performed on line 7 of Algorithm 6.1: While computing the decision variable, it searches for implied (forced) variables to reduce the current formula.

Another simple direction heuristic is used in the look-ahead SAT solver kc-nfs [DD]. It aims at selecting the most satisfiable branch. It compares the difference of occurrences between x_{decision} and $\neg x_{\text{decision}}$. The larger of these two satisfies more clauses in which x_{decision} occurs and is therefore preferred.

The look-ahead SAT solver march_ks bases its direction heuristics on the reduction caused by the decision variable [HvM06]. The reduction from $\mathcal{F}$ to $\mathcal{F}(x_{\text{decision}} = 0)$ and from $\mathcal{F}$ to $\mathcal{F}(x_{\text{decision}} = 1)$ is measured by the number of clauses that are reduced in size without being satisfied. In general, the stronger this reduction the higher the probability the subformula is unsatisfiable. Therefore, march_ks branches first on the subformula with the smallest reduction.

Oliver Kullmann proposes direction heuristics (used in his look-head OK-solver) to select the subformula with the lowest probability that a random assignment will falsify a random formula of the same size [Kul02]. Let $\mathcal{F}_k$ denote the set of clauses in $\mathcal{F}$ of size k. It prefers either $\mathcal{F}(x_{\text{decision}} = 0)$ or $\mathcal{F}(x_{\text{decision}} = 1)$ for which the following is smallest:

$$\sum_{k \geq 2} -|\mathcal{F}_k| \cdot \ln(1 - 2^{-k}) \qquad \lfloor 6.1$$

6.3 Observed bias on random k-SAT formulae

This section studies the effectiveness of existing direction heuristics of SAT solvers based on the DPLL architecture. Here we will provide a large study of different solvers on random k-SAT formulae with different sizes and densities. The main motivation to use these formulae is that one can easily create many instances of different sizes and hardness. Therefore, this family of formulae seems an obvious candidate to test whether the direction heuristics used in some SAT solvers satisfy the bias-extrapolation property. We focus on the hard random k-SAT instances - near the (observed) phase transition density. The concepts introduced in this section are developed to offer some insights in the effectiveness of direction heuristics.

6.3.1 Distribution of solutions

We determined the bias of the distribution of solutions amongst the various subtrees using the following experiment: Consider all the subtrees $T_{d,i}$ which are at depth d. Assuming that the search-tree is big enough, there are 2^d of these subtrees. Given a set of satisfiable formulae, what is the probability that a certain subtree contains a solution? Let the left branch in a node denote the subformula - either $\mathcal{F}(x_i = 0)$ or $\mathcal{F}(x_i = 1)$ - which a solver decides to examine first. Consequently, we refer to the right branch as the latter one.

Subtrees are numbered from left to right starting with $T_{d,0}$ (see Figure 6.2 for an example with $d = 3$). We generated sets of random k-SAT formulae for various sizes of the number of variables (denoted by n) and for different densities (clause-variable ratio, denoted by ρ). For each set, 10.000 formulae (satisfiable and unsatisfiable) were generated from which we discarded the unsatisfiable instances.

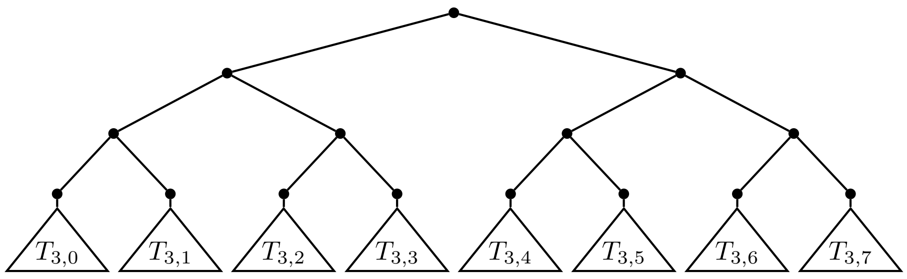

Figure 6.2 — A search-tree with jump depth 3 and 8 subtrees T_i

Definition: The *satisfying subtree probability* $P_{\text{sat}}(d, i)$ denotes - for a given SAT solver and a set of satisfiable benchmarks - the fraction of satisfiable instances that have at least one solution in $T_{d,i}$.

Definition: The *satisfying subtree mean* $\mu_{\text{sat}}(d)$ denotes - for a given SAT solver and a set of satisfiable benchmarks - the average number of subtrees at depth d that have at least one solution.

We compute $\mu_{\text{sat}}(d)$ as

$$\mu_{\text{sat}}(d) = \sum_{i=0}^{2^d-1} P_{\text{sat}}(d, i) \qquad \lfloor 6.2$$

By definition $P_{\text{sat}}(0, 0) = 1$ and $\mu_{\text{sat}}(0) = 1$. Because formulae could have solutions in both $T_{d,2i}$ and $T_{d,2i+1}$, $\mu_{\text{sat}}(d)$ is increasing. Or more formal:

$$\begin{aligned} P_{\text{sat}}(d, 2i) + P_{\text{sat}}(d, 2i + 1) &\geq P_{\text{sat}}(d - 1, i) \text{ , and thus} && \lfloor 6.3 \\ \mu_{\text{sat}}(d) &\geq \mu_{\text{sat}}(d - 1) && \lfloor 6.4 \end{aligned}$$

Given a SAT solver and a set of satisfiable benchmarks, we can estimate for all 2^d subtrees $T_{d,i}$ the probability $P_{\text{sat}}(d, i)$. A histogram showing the $P_{\text{sat}}(12, i)$ values using march_ks on the test set with $n = 350$ and $\rho = 4.26$ is shown in Figure 6.3. We refer to such a plot as to the *solution distribution histogram*. The horizontal axis denotes the subtree index i of $T_{12,i}$, while the vertical axis provides the satisfying subtree probability.

Colors visualize the number of right branches (denoted as $\#RB$) required to reach a subtree: $\#RB(T_{d,0}) = 0$, $\#RB(T_{d,1}) = 1$, $\#RB(T_{d,2}) = 1$, $\#RB(T_{d,3}) = 2$,$\#RB(T_{d,4}) = 1$ etc. The figure clearly shows that the distribution is biased towards the left branches: The highest probability is $P_{\text{sat}}(12, 0)$ (zero right branches), followed by $P_{\text{sat}}(12, 2048)$, $P_{\text{sat}}(12, 1024)$, and $P_{\text{sat}}(12, 256)$ - all reachable by one right branch.

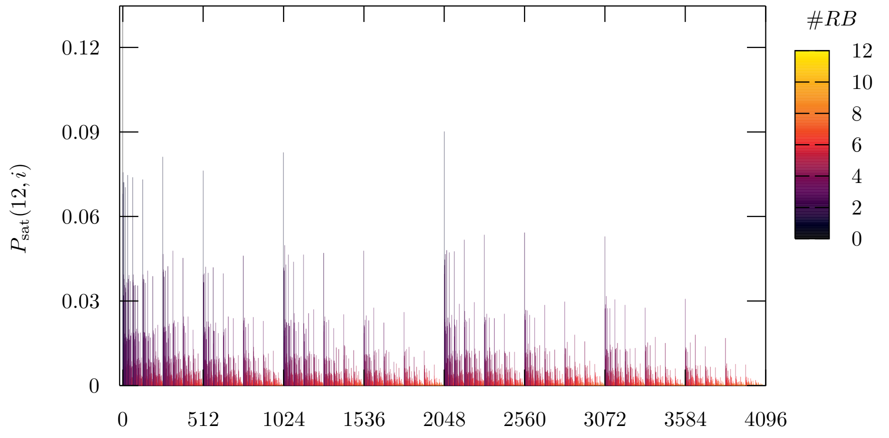

Figure 6.3 — Solution distribution histogram showing $P_{\text{sat}}(12, i)$ using march_ks on 10.000 random 3-SAT formulae with $n = 350$ and $\rho = 4.26$. For this experiment, $\mu_{\text{sat}}(12) = 19.534$.

A solution distribution histogram of $P_{\text{sat}}(12, i)$ using kcnfs on the same benchmark set is shown in Figure 6.4. Similar to the histogram using march_ks, the $P_{\text{sat}}(12, i)$ values are higher if $T(12, i)$ can be reached in less right branches. However, the high (peak) $P_{\text{sat}}(12, i)$ values are about 50% higher for march_ks than for kcnfs, while the $\mu_{\text{sat}}(10)$ values for both solvers does not differ much. So, the low $P_{\text{sat}}(12, i)$ values must be higher for kcnfs. This can be observed in the more dense part down in the histogram. Since the same test set is used, based on the lower peak $P_{\text{sat}}(12, i)$ values we can conclude that the direction heuristics of kcnfs result in a smaller bias to the left branches on these instances.

Again, we see (Figure 6.5) a bias towards the left branches if we look at the solution distribution histogram of march_ks on random 4-SAT formulae near the phase transition density (in this case $\rho = 9.9$). Also, the high $P_{\text{sat}}(12, i)$ values for march_ks on random 3-SAT are much higher than on random 4-SAT formulae.

However, the $\mu_{\text{sat}}(12)$ value is much smaller too. So, the lower peaks might be caused by the lower number of satisfiable subtrees. Therefore, we cannot easily conclude that the direction heuristics used in march_ks result in a larger bias on random 3-SAT formulae.

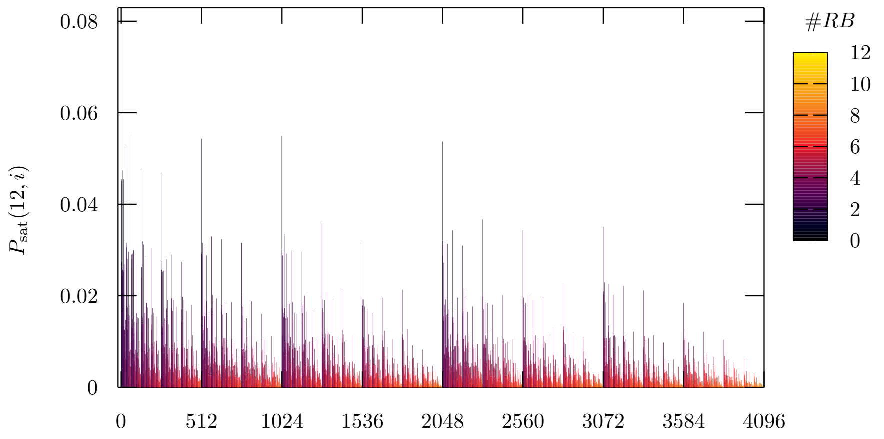

Figure 6.4 — Solution distribution histogram showing $P_{\text{sat}}(12, i)$ using kcnfs on 10.000 random 3-SAT formulae with $n = 350$ and $\rho = 4.26$. For this experiment, $\mu_{\text{sat}}(12) = 18.021$.

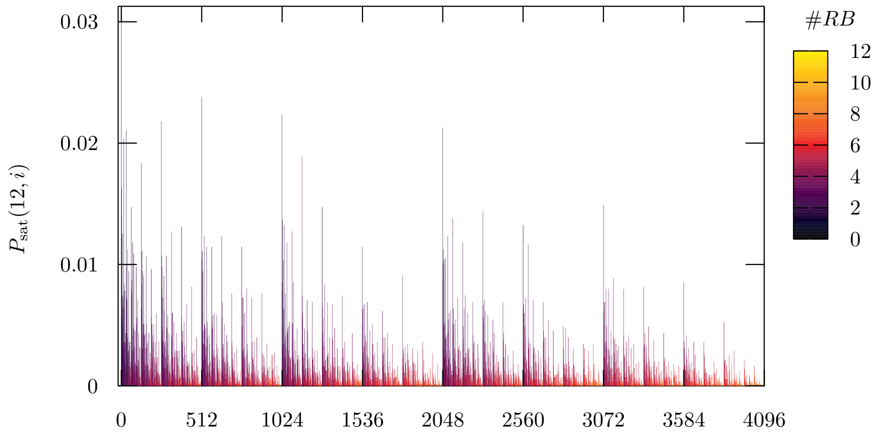

Figure 6.5 — Solution distribution histogram showing $P_{\text{sat}}(12, i)$ using march_ks on 10.000 random 4-SAT formulae with $n = 120$ variables and $\rho = 9.9$. For this experiment, $\mu_{\text{sat}}(12) = 4.817$.

Appendix 6.A shows solution distribution histograms for various sizes and densities of random 3-SAT formulae obtained using march_ks. First and foremost, we see in all solution distribution histograms the characteristic bias towards the left branches. From this observation the claim seems justified that the actions of march_ks on random 3-SAT formulae satisfy the bias extrapolation property. Another similarity is that the peak $P_{\text{sat}}(10, i)$ values are about the same size for test sets with the same density. Regarding $\mu_{\text{sat}}(10)$ values, we observe that $\mu_{\text{sat}}(10)$ is larger if the number of variables is larger. Also, amongst formulae with the same ρ, $\mu_{\text{sat}}(10)$ is higher while the the peak $P_{\text{sat}}(10, i)$ values are comparable. So, the $\mu_{\text{sat}}(10)$ values are higher because of higher low $P_{\text{sat}}(10)$ values. This can be observed in the histograms by the more dense lower sections of the plots.

6.3.2 Satisfying subtree bias

We observed that the distribution of solutions (while experimenting with random k-SAT formulae) is biased towards the left branches due to the direction heuristics used in some SAT solvers. Although there is a great deal of resemblance between the various solution distribution histograms, the magnitude of the $P_{\text{sat}}(d, i)$ values differs. Yet, this magnitude does not easily translate into a comparable bias. To express the differences, we introduce a measurement called the *satisfying subtree bias*.

Definition: The satisfying subtree bias $B_{\text{sat}}(d, i)$ denotes - for a given SAT solver and a set of satisfiable benchmarks - the normalized fraction of all formulae which have a solution in $T_{d-1,\frac{i}{2}}$ that also have a solution in $T_{d,i}$.

The satisfying subtree bias $B_{\text{sat}}(d, i)$ is computed as:

$$B_{\text{sat}}(d, i) = \frac{P_{\text{sat}}(d, i)}{P_{\text{sat}}(d, 2\lfloor\frac{i}{2}\rfloor) + P_{\text{sat}}(d, 2\lfloor\frac{i}{2}\rfloor + 1)} \qquad \lfloor 6.5$$

For even i, we say that the branch towards $T_{d,i}$ is biased to the left if $B_{\text{sat}}(d, i) > 0.5$. In case $B_{\text{sat}}(d, i) < 0.5$ we call these branches biased to the right. For odd i the complement holds. The larger the difference between $B_{\text{sat}}(d, i)$ and 0.5, the larger the bias.

To compare the effectiveness of direction heuristics of SAT solvers on random k-SAT formulae, we measured on various depths the average bias towards the left branches. This bias for depth d is computed as the sum of all $B_{\text{sat}}(d, i)$ values with i even, divided by the number of nonzero $B_{\text{sat}}(d-1, i)$ values.

Figure 6.6 shows the average bias and offers some interesting insights: First, we see that, for both march_ks and kcnfs, the average bias towards the left branches is comparable for small d and in particular $d = 1$. This can be explained by the fact that the direction heuristics used in both solvers is similar for formulae with no or few binary clauses. For small d on random k-SAT formulae this is the case. Second, the larger d, the larger the average bias towards the left branches, for all solver / test set combinations. This supports the intuition that direction heuristics are more effective lower in the search-tree, because the formula is reduced and therefore less complex. Third, the direction heuristics used in march_ks are clearly more effective on the experimented random k-SAT test sets.

Another visualization comparing the effectiveness of direction heuristics are $P_{\text{sat}}/B_{\text{sat}}$ trees. The root of such a tree contains all satisfiable formulae in the given test set. The number in the vertices show the fraction of formulae that still have a solution (the $P_{\text{sat}}(d, i)$ values), while the edges are labeled with the $B_{\text{sat}}(d, i)$ values.

Appendix 6.B shows $P_{\text{sat}}/B_{\text{sat}}$ trees for march_ks and kcnfs on random 3-SAT and random 4-SAT formulae. Figure 6.13 shows such a tree for random 3-SAT with $n = 350$ and $\rho = 4.26$ using march_ks. Again, we see for all d, $P_{\text{sat}}(d, i)$ values are higher if $T(d, i)$ can be reached with less right branches. Also, again we see that the $B_{\text{sat}}(d, i)$ towards the left increases, while d increases. So, the used direction heuristics seem to "guess" the branch containing a solution lower in the search-tree more accurately. Both observations above are also supported by the other $P_{\text{sat}}/B_{\text{sat}}$ trees.

6.3.3 Finding the first solution

The observed distribution of solutions as well as the bias provide some insight in the effectiveness of the used direction heuristics. Yet, for satisfiable instances we are mainly interested in the usefulness of these heuristics to find the *first* solution quickly.

In order to naturally present the effect of distribution jumping, we first (this subsection) consider the march_ks solver *without* this feature and refer to it as march_ks$^-$. More precisely, march_ks$^-$ visits subtrees in chronological (or depth first) order. We denote chronological order at jump depth d by $\pi_{\text{chro},d}$ $= (0, 1, 2, 3, \ldots, 2^d - 1)$. Since other (jump) orders will be discussed later, all definitions use an arbitrary order at depth d called $\pi_{j,d}$.

Definition: The *first solution probability* $P_{\text{first}}(\ \pi_{j,d},\ i\ )$ denotes - for a given SAT solver and a set of satisfiable benchmarks - the probability that while visiting subtrees at depth d using jump order $\pi_{j,d}$, the solver needs to explore more than i subsequent subtrees to find the first solution (satisfying assignment).

For any $\pi_{j,d}$ holds $P_{\text{first}}(\pi_{j,d},0) = 1$, $P_{\text{first}}(\pi_{j,d},2^d) = 0$, and $P_{\text{first}}(\pi_{j,d},i) \geq P_{\text{first}}(\pi_{j,d},i{+}1)$. A *first solution probability plot* for a given SAT solver and benchmark family shows the probabilities $P_{\text{first}}(\pi_{j,d}, i)$ with $i \in \{0, \ldots, 2^d\}$. Figure 6.7 shows the probability plot for various random 3-SAT formulae solved using march_ks$^-$. The order of the test sets in the legend represents the order of the probability plots. The shape for the various size and densities have many similarities.

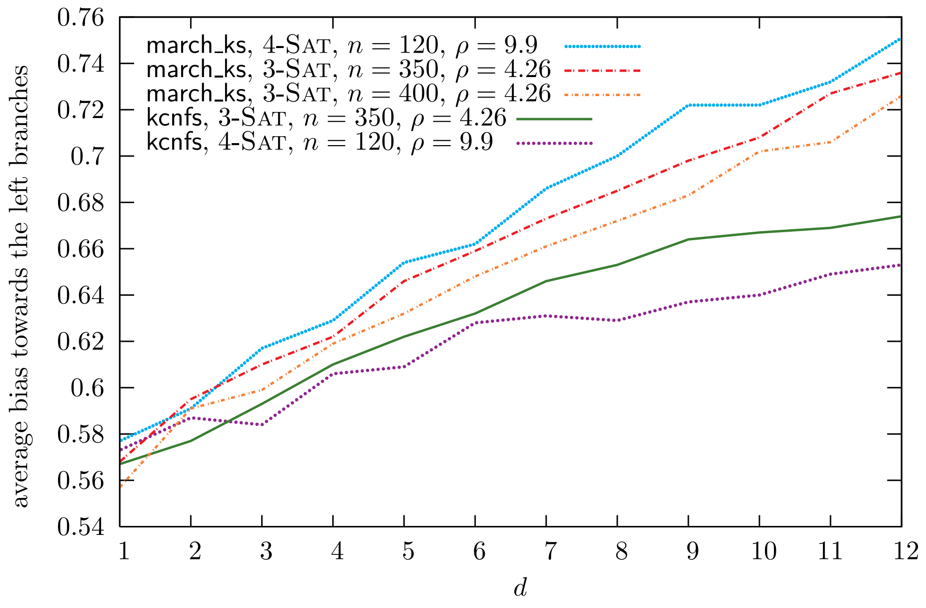

Figure 6.6 — The average (for various d) bias towards the left branches using march_ks and kcnfs on random k-SAT formulae.

Definition: The *expected tree size* $E_{\text{size}}(\pi_{j,d})$ denotes - for a given SAT solver and a set of satisfiable benchmarks - the expected size of the search-tree required to solve an instance while visiting subtrees at depth d using jump order $\pi_{j,d}$.

$E_{\text{size}}(\ \pi_{j,d}\ )$ is computed as:

$$E_{\text{size}}(\pi_{j,d}) = 2^{-d} \sum_{i \in \{0,\ldots,2^d-1\}} P_{\text{first}}(\pi_{j,d}, i) \qquad \lfloor 6.6$$

Notice that $E_{\text{size}}(\pi_{j,d})$ correlates with the size of the surface below the first solution probability plot with solver and jump depth d. Based on Figure 6.7, we can state that for march_ks^{-} the expected tree size $E_{\text{size}}(\pi_{\text{chro},12})$ increases if the density increases. This was expected because formulae with a higher density have on average fewer solutions; it takes longer to find the first solution. For test sets with the same density, $E_{\text{size}}(\pi_{\text{chro},12})$ is slightly smaller for those formulae with more variables. Based on these data, no conclusions can be drawn regarding the computational time: Exploring the whole search-tree requires much more effort for hard random formulae with increased size.

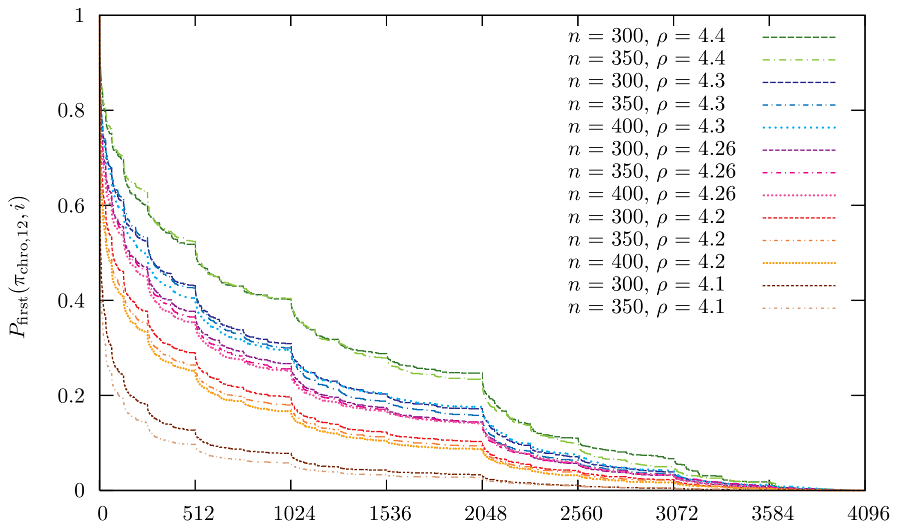

Figure 6.7 — First solution probability plot showing $P_{\text{first}}(\pi_{\text{chro},12}, i)$ on random 3-SAT formulae using march_ks.

6.4 Distribution jumping

The idea behind *distribution jumping* arises naturally from the observations made in the prior section: While visiting the subtrees in chronological (or depth first) order, the first solution probability plots show some characteristic angels - hinting that the expected tree size is far from optimal. So, could we construct an alternative jump order which visits the subtrees in a (near) optimal order? First we will discuss the possibilities to optimize the jump order, followed by some thoughts on the optimal jump depth.

6.4.1 Optimizing the jump order

This section deals with the question how to construct a jump order with the (near) minimal expected tree size. First, we motivate why we focus on the minimization of the expected tree size. Second, we determine the (close to) minimal $E_{\text{size}}(\pi_{j,12})$ for various random k-SAT formulae using a greedy algorithm. Third, we construct a generalized jump order that can be implemented into a SAT solver.

Theoretical speed-up

Why shall we try to optimize the expected tree size? The reason is closely related to the *theoretical speed-up*. The performance gain realized by distribution jumping can be approximated if the following two restrictions are met:

R_1) the size of the subtrees is about equal;
R_2) the jump depth is much smaller than the size of subtrees.

Due to R_1 the expected computational costs of each subtree is equal and R_2 marginalizes the overhead costs - getting from one subtree to another - of the distribution jumping technique. Using march_ks, the search-trees for hard random k-SAT appear to be quite balanced (satisfying R_1). Given a relatively small jump depth (satisfying R_2) the speed-up could be computed. However, R_1 and R_2 are probably hard to meet for more structured formulae. Given a large set of satisfiable benchmarks, we can compute the theoretical speed-up caused by distribution jumping $\pi_{j,d}$:

$$S(\pi_{j,d}) := \frac{E_{\text{size}}(\pi_{\text{chro},d})}{E_{\text{size}}(\pi_{j,d})} \qquad \lfloor 6.7$$

Greedy jump order

For jump depth d there exist $2^d!$ jump orders in which subtrees can be visited. This makes it hard to compute the *optimal jump order*. We denote the optimal jump order $\pi_{\text{opt},d}$(test set), the $\pi_{j,d}$ for which $E_{\text{size}}(\pi_{j,d})$ is minimal for the test set using a given SAT solver. Because $\pi_{\text{opt},d}$(test set) is hard to compute, we focused on an approximation.

Any $\pi_{\text{opt},d}$(test set) has the property that it is *progressive*: Given a SAT solver and benchmark set, we call $\pi_{j,d}$ a progressive jump order if the probability that the first solution is found at the i-th visited subtree according to $\pi_{j,d}$ ($P_{\text{first}}(\pi_{j,d}, i+1) - P_{\text{first}}(\pi_{j,d}, i)$) is decreasing.

A $\pi_{j,d}$ that is not progressive can easily be improved with respect to a smaller expected tree size: If the probability is not decreasing for a certain i, then swap the order in which the i-th and $i+1$-th subtrees are visited according to that $\pi_{j,d}$ in order to obtain a ${\pi_{j,d}}^*$ which has a smaller expected tree size. Any $\pi_{j,d}$ can be made progressive by applying a finite number of those swaps.

Out of the $2^d!$ possible jump orders, probably only a small number is progressive. We assume that other progressive jump orders of a given test set have an expected tree size close to the minimal one. Therefore, we developed a greedy algorithm which computes for a given SAT solver and test set a progressive jump order called $\pi_{\text{greedy},d}$(test set). Consider the following algorithm:

1. Start with an empty jump order and a set of satisfiable benchmarks.
2. Select the subtree $T_{d,i}$ in which most formulae have at least one solution. In case of a tie-break we select the one with the smallest i. The selected subtree is the next to be visited in the greedy jump order.
3. All formulae that have at least one solution in the selected subtree (of step 2) are removed from the set.
4. Repeat the steps 2 and 3 until all formulae of the set are removed.

We computed the greedy jump order for various random k-SAT test sets (denoted by $R_{k,n,\rho}$) using march_ks. The results are shown in Table 6.1. Columns two to five show the order in which $T_{12,i}$'s should be visited according to the greedy orders based on different hard random k-SAT formulae. To clearly present also the $\#RB(T_{12,i})$ values within the order, only the indices i are shown. These are printed in binary representation with bold 1's (right branches). E.g., in Table 6.1 $T_{12,1024}$ is shown as 0**1**0000000000.

Table 6.1 — $\pi_{\text{greedy},12}(k, n, \rho)$ computed for random 3-SAT ($n \in \{300, 305, 400\}$, $\rho = 4.26$) and 4-SAT ($n = 120$, $\rho = 9.9$). The index of the greedy jump orders is shown in binary representation.

$\pi_{\text{chro},12}$	$\pi_{\text{greedy},12}$ $(R_{3,300,4.26})$	$\pi_{\text{greedy},12}$ $(R_{3,350,4.26})$	$\pi_{\text{greedy},12}$ $(R_{3,400,4.26})$	$\pi_{\text{greedy},12}$ $(R_{4,120,9.9})$
0	000000000000	000000000000	000000000000	000000000000
1	**1**00000000000	**1**00000000000	**1**00000000000	00**1**000000000
2	0**1**0000000000	0**1**0000000000	00**1**000000000	0**1**0000000000
3	00**1**000000000	000**1**00000000	000**1**00000000	000**1**00000000
4	0000**1**0000000	00**1**000000000	0**1**0000000000	**1**00000000000
5	000000**1**00000	000000**1**00000	0000**1**0000000	0**1**00**1**0000000
6	000**1**00000000	0000**1**0000000	0000000**1**0000	000000**1**00000
7	00000**1**000000	**1**0**1**000000000	00000**1**000000	0000**1**0000000
8	**11**0000000000	00000000**1**000	**1**0000**1**000000	0000000**1**0000
9	**1**0000**1**000000	**1**00**1**00000000	**1**0**1**000000000	**11**0000000000
10	**1**000**1**0000000	0000000**1**0000	**1**0000**1**000000	**1**00**1**00000000
11	0000000**1**0000	**11**0000000000	000000**1**00000	**1**0**1**000000000
12	0**1**0**1**00000000	0**11**000000000	**11**0000000000	**1**000**1**0000000
13	00000000**1**000	0000000000**1**0	00000000**1**000	000000000**1**00
14	**1**0000**1**000000	**1**000**1**0000000	0**1**00**1**0000000	0**1**0**1**00000000
15	00**11**00000000	00000**1**000000	00**11**00000000	0**1**000**1**000000
16	**1**0**1**000000000	**1**000000**1**0000	00**1**0**1**0000000	000**11**0000000
17	**1**00**1**00000000	**1**0000**1**000000	**1**00**1**00000000	00**11**00000000
18	000**11**0000000	0**1**000**1**000000	0**11**000000000	00000**1**000000
19	00000000000**1**	00**1**0000**1**0000	0000**1**0**1**00000	0**1**0000**1**00000
...	...	...	...	...

An interesting trait that can be observed from the $\pi_{\text{greedy},12}(R_{k,n,\rho})$ jump orders is that they spoil the somewhat perfect pictures presented in Section 6.3. Recall that in the solution distribution histograms all subtrees which can be reached in a single right branch show higher peaks than all subtrees reachable with two (or more) right branches. Here, we observe a similar tendency. Yet, $T_{12,1}$, $T_{12,2}$, $T_{12,4}$, $T_{12,8}$ seem far less important than for instance $T_{12,3072}$ (**11**0000000000). So, based on the greedy jump orders we can conclude that for optimal performance, $T_{d,i}$ should not solely be visited in increasing number of right branches. In other words, the $P_{\text{sat}}(d, i)$ values are not a perfect tool for constructing the ideal jump order.

Generalized jump order

Since we only computed $\pi_{\text{greedy},12}$ (only for jump depth 12) and the order is slightly different for the different experimented data-sets, a more generalized permutation is required for the actual implementation. Although the greedy jump orders are our best approximation of the optimal jump order, we failed to convert them to a generalized jump order. Instead, we created a generalized jump order based on two prior observations: First, subtrees reachable in less right branches have a higher probability of containing a solution (see Section 6.3.1). Second, among subtrees which are reachable in the same number of right branches, those with right branches near the root have a higher probability of containing a solution (see Section 6.3.2).

Based on these observations, we propose the following jump order: First, visit $T_{d,0}$, followed by all $T_{d,i}$'s that can be reached in only one right branch. Continue with all T_i's that can be reached in two right branches, etc. All $T_{d,i}$'s that can be reached in the same number of right branches are visited in decreasing order of i. We refer to this permutation as $\pi_{\text{left},d}$.

Table 6.2 shows the visiting order of the subtrees with $\pi_{\text{chro},4}$ and $\pi_{\text{left},4}$. There are only a few similarities between $\pi_{\text{chro},d}$ and $\pi_{\text{left},d}$: The first and the last subtree ($T_{d,0}$ and $T_{d,2^d-1}$) have the same position in both jump orders. Also, both $\pi_{\text{chro},d}$ and $\pi_{\text{left},d}$ are symmetric: If subtree $T_{d,i}$ has position j in the order than subtree $T_{d,2^d-1-j}$ has position $2^d - 1 - j$ in that order.

Notice that $\pi_{\text{chro},d}$ and $\pi_{\text{left},d}$ are in some sense complementary: $\pi_{\text{chro},d}$ starts by visiting subtrees that have right branches near the leaf nodes, while $\pi_{\text{left},d}$ starts by visiting subtrees having right branches near the root. Since the next subtree selected by $\pi_{\text{chro},d}$ is the nearest subtree, "jumping" is very cheap. A motivation for using $\pi_{\text{left},d}$ is that direction heuristics are more likely to fail near the root of the search tree because at these nodes it is harder to predict the "good" direction. Therefore, subtrees with few right branches near the root of the search-tree have a higher probability of containing a solution.

Using the data from the experiments on random k-SAT formulae to compute the distribution of solutions, we obtained[20.] the $P_{\text{first}}(\pi_{\text{left},d}, i)$ and $E_{\text{size}}(\pi_{\text{left},d}, i)$ values. Figure 6.8 shows the first solution probability plots based on this data for random 3-SAT test sets of different sizes and densities using march_ks$^-$ and $\pi_{\text{left},12}$. The lines decrease much more rapidly compared to those of Figure 6.7, also indicating that the expected tree size is much smaller using this jump order. The sequence (from top to bottom) of these lines is about the same for both figures: The larger ρ, the higher the line in the sequence. However, in Figure 6.8 a larger n does not always result in a lower line.

Table 6.3 summarizes the results of these tests on all the experimented data. Shown is $\mu_{\text{sat}}(d)$, $E_{\text{size}}(\pi_{\text{chro},d})$, $E_{\text{size}}(\pi_{\text{left},d})$ and the theoretical speed-up which is computed using these values. The speed-up using march_ks$^-$ with $\pi_{\text{left},d}$ is

[20.] The data was gathered using march_ks$^-$ while visiting the subtree in chronological order. Because march_ks$^-$ uses adaptive heuristics, actual jumping according to $\pi_{\text{left},d}$ might result in a slightly different search-tree and thus may influence the $P_{\text{first}}(\pi_{\text{left},d}, i)$ and $E_{\text{size}}(\pi_{\text{left},})$ values. Further discussion in Section 6.5.1

Table 6.2 — Jump orders in which subtrees can be visited with jump depth 4.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi_{\text{chro},4}$	$T_{4,0}$	$T_{4,1}$	$T_{4,2}$	$T_{4,3}$	$T_{4,4}$	$T_{4,5}$	$T_{4,6}$	$T_{4,7}$	$T_{4,8}$	$T_{4,9}$	$T_{4,10}$	$T_{4,11}$	$T_{4,12}$	$T_{4,13}$	$T_{4,14}$	$T_{4,15}$
$\pi_{\text{left},4}$	$T_{4,0}$	$T_{4,8}$	$T_{4,4}$	$T_{4,2}$	$T_{4,1}$	$T_{4,12}$	$T_{4,10}$	$T_{4,9}$	$T_{4,6}$	$T_{4,5}$	$T_{4,3}$	$T_{4,14}$	$T_{4,13}$	$T_{4,11}$	$T_{4,7}$	$T_{4,15}$

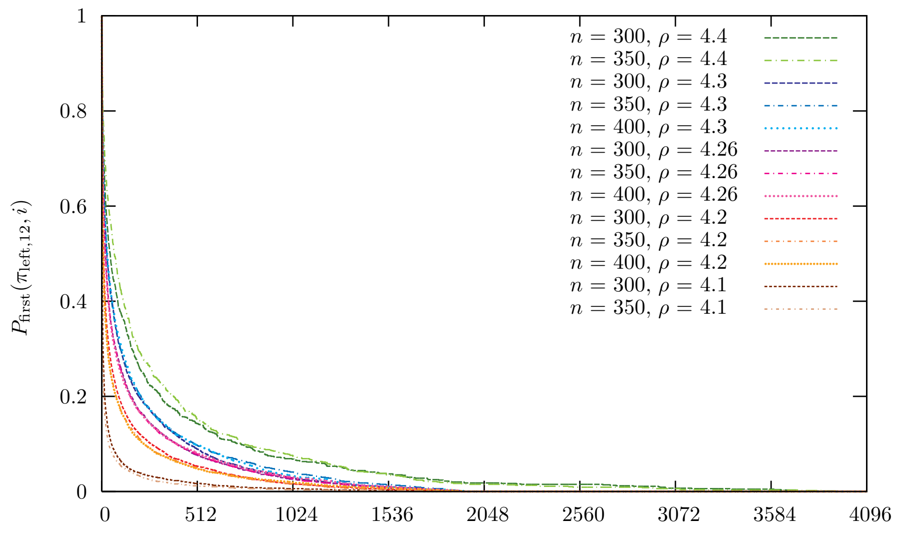

Figure 6.8 — First solution probability plot showing $P_{\text{first}}(\pi_{\text{left},12}, i)$ on random 3-SAT formulae using march_ks$^-$. Please compare to Figure 6.7.

about a factor 4. On formulae below the phase transition density, the speed-up is even greater, while above the phase transition density it is less. Also, formulae with a higher $\mu_{\text{sat}}(d)$ value have a greater speed-up - although the correlation is much less clear. The speed-up realized by kcnfs with $\pi_{\text{left},d}$ is smaller than by march_ks$^-$. A possible explanation is that the direction heuristics used in kcnfs result in a smaller bias towards the left. Using march_ks$^-$ with $\pi_{\text{left},d}$, the speed-up on the 4-SAT test set is the smallest. This may be caused by the relatively small $\mu_{\text{sat}}(12)$ value.

6.4.2 Optimizing the jump depth

Under the assumption that $\pi_{\text{left},d}$ is an effective jump order, we now face the question: When to jump? Or more precise: What is the optimal jump depth? For different values of jump depth d, the leaf nodes of the search-tree are visited in a different order - in contrast to jumping using $\pi_{\text{chro},d}$. Which d is optimal? First, we will try to answer this question from a theoretical viewpoint followed by some practical difficulties.

Table 6.3 — Expected size and speed-up for random k-SAT formulae.

solver	**family**	n	ρ	$\mu_{\text{sat}}(12)$	$E_{\text{size}}(\pi_{\text{chro},12})$	$E_{\text{size}}(\pi_{\text{left},12})$	$S(\pi_{\text{left},12})$
march_ks$^-$	3-SAT	300	4.1	95.803	0.04965	0.00993	5.00
march_ks$^-$	3-SAT	300	4.2	29.575	0.11863	0.02727	4.35
march_ks$^-$	3-SAT	300	4.26	15.775	0.15673	0.03804	4.12
march_ks$^-$	3-SAT	300	4.3	10.843	0.18026	0.04365	4.13
march_ks$^-$	3-SAT	300	4.4	5.379	0.23408	0.05456	4.29
march_ks$^-$	3-SAT	350	4.1	147.933	0.03934	0.00811	4.85
march_ks$^-$	3-SAT	350	4.2	40.199	0.10919	0.02510	4.35
march_ks$^-$	3-SAT	350	4.26	19.534	0.15406	0.03822	4.03
kcnfs	3-SAT	350	4.26	18.021	0.16990	0.06113	2.78
march_ks$^-$	3-SAT	350	4.3	12.925	0.17409	0.04569	3.81
march_ks$^-$	3-SAT	350	4.4	5.871	0.22971	0.06399	3.59
march_ks$^-$	3-SAT	400	4.2	52.906	0.10237	0.02572	3.98
march_ks$^-$	3-SAT	400	4.26	24.461	0.15047	0.04056	3.71
march_ks$^-$	3-SAT	400	4.3	15.281	0.17682	0.04715	3.75
march_ks$^-$	4-SAT	120	9.9	4.817	0.23514	0.07864	2.99

Notice that

$$\begin{aligned} P_{\text{first}}(\pi_{\text{chro},d}, 2i) &= P_{\text{first}}(\pi_{\text{chro},d-1}, i) \qquad \text{and} \\ P_{\text{first}}(\pi_{\text{chro},d-1}, i+1) \leq P_{\text{first}}(\pi_{\text{chro},d}, 2i+1) &\leq P_{\text{first}}(\pi_{\text{chro},d-1}, i) \end{aligned}$$

So,

$$\begin{aligned} \frac{2E_{\text{size}}(\pi_{\text{chro},d-1}) - 2^{1-d}P_{\text{first}}(\pi_{\text{chro},d-1}, 0)}{2} &\leq E_{\text{size}}(\pi_{\text{chro},d}) \leq \frac{2E_{\text{size}}(\pi_{\text{chro},d-1})}{2} \\ E_{\text{size}}(\pi_{\text{chro},d-1}) - 2^{-d} &\leq E_{\text{size}}(\pi_{\text{chro},d}) \leq E_{\text{size}}(\pi_{\text{chro},d-1}) \end{aligned}$$

In other words, $E_{\text{size}}(\pi_{\text{chro},d})$ is decreasing for increasing d and converges fast. These properties are independent of the used SAT solver[21.] and the used benchmark family. Figure 6.9 shows how $P_{\text{first}}(\pi_{\text{chro},d}$, i) (and thus $E_{\text{size}}(\pi_{\text{chro},d})$ evolves for increasing d based on 10.000 random 3-SAT formulae with $n = 400$ and $\rho = 4.26$ using march_ks. For these formulae and solver, $E_{\text{size}}(\pi_{\text{chro},d})$ converges to 0.150.

Both properties - decrease and convergence - may not hold for $E_{\text{size}}(\pi_{\text{left},d})$ for some solver / benchmark family combinations. The influence of the jump depth on $E_{\text{size}}(\pi_{\text{left},d})$ depends heavily on the direction heuristics used in a solver. Using march_ks on random 3-SAT formulae, we observed that $E_{\text{size}}(\pi_{\text{left},d})$ is decreasing for increasing d. However (fortunately), $E_{\text{size}}(\pi_{\text{left},d})$ is still visibly decreasing for increasing d during our experiments (using $d \in \{0, \ldots, 12\}$). This observation is visualized in Figure 6.10. Consequently the theoretical speed-up

[21.] Assuming that subtrees are visited in chronological order and no restarts are performed of using march_ks with distribution jumping based on $\pi_{\text{left},d}$ instead of march_ks$^-$ improves while increasing d.

This is also the main conclusion of Table 6.4 - showing the influence of the jump depth on $E_{\text{size}}(\pi_{\text{chro},d})$, $E_{\text{size}}(\pi_{\text{left},d})$ and the theoretical speed-up. Notice also that $E_{\text{size}}(\pi_{\text{chro},0}) = E_{\text{size}}(\pi_{\text{left},0})$ and $E_{\text{size}}(\pi_{\text{chro},1}) = E_{\text{size}}(\pi_{\text{left},1})$. While using march_ks on random 3-SAT formulae with $n = 400$ and $\rho = 4.26$, $E_{\text{size}}(\pi_{\text{left},d}) < E_{\text{size}}(\pi_{\text{chro},d})$ during the experiments (i.e. for $d \in \{2, \ldots, 12\}$).

Table 6.4 — The influence of the jump depth denoted as d on $E_{\text{size}}(\pi_{\text{chro},d})$, $E_{\text{size}}(\pi_{\text{left},d})$ and the expected speed-up $S(\pi_{\text{left},d})$ on random 3-SAT formulae with 400 variables and density 4.26.

d	$E_{\text{size}}(\pi_{\text{chro},d})$	$E_{\text{size}}(\pi_{\text{left},d})$	$S(\pi_{\text{left},d})$
0	1.000	1.000	1.000
1	0.571	0.571	1.000
2	0.357	0.347	1.029
3	0.252	0.231	1.089
4	0.200	0.165	1.215
5	0.175	0.128	1.373
6	0.163	0.102	1.589
7	0.156	0.085	1.837
8	0.153	0.072	2.125
9	0.152	0.062	2.440
10	0.151	0.054	2.790
11	0.151	0.048	3.173
12	0.151	0.041	3.713

So, based on the theoretical speed-up, the optimal value for the jump depth is probably $d = \infty$. In other words, theoretically jumping between the leaf nodes of the search-tree results in the largest performance gain.

However, in practice, two kinds of overhead exist: First, the cost of jumping. While solving, march_ks spends most time to determine for each node the decision variable and to detect forced variables. Compared to these calculations, the cost of jumping (backtracking to the nearest parent node and descending in the tree) for one subtree to another is relatively cheap. Therefore, the overhead of jumping is marginal.

On the other hand, because the cost of obtaining the decision variable and forced literals is huge, one would like to remember this information. Therefore, these data should be stored for nodes that will be visited more than once (i.e. those nodes at depth $\leq d$). That will cost a lot of memory and time if d is large.

To reduce overhead, we therefore decided to use a jump depth in such way that only a fraction of the nodes needs to be stored: In the first phase of solving (before the first jump) the average depth of the search-tree is estimated. The jump depth is set to be 7 levels above this average. Using this heuristic only about 1 in 100 nodes is stored.

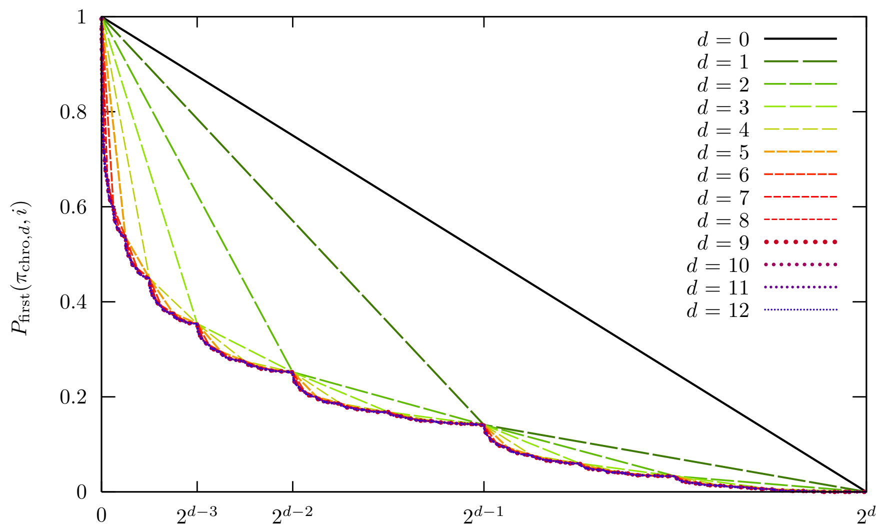

Figure 6.9 — Influence of the jump depth d on the first solution probability $P_{\text{first}}(\pi_{\text{chro},d}, \text{i})$ using march_ks on 10.000 random 3-SAT formulae with $n = 400$ and $\rho = 4.26$.

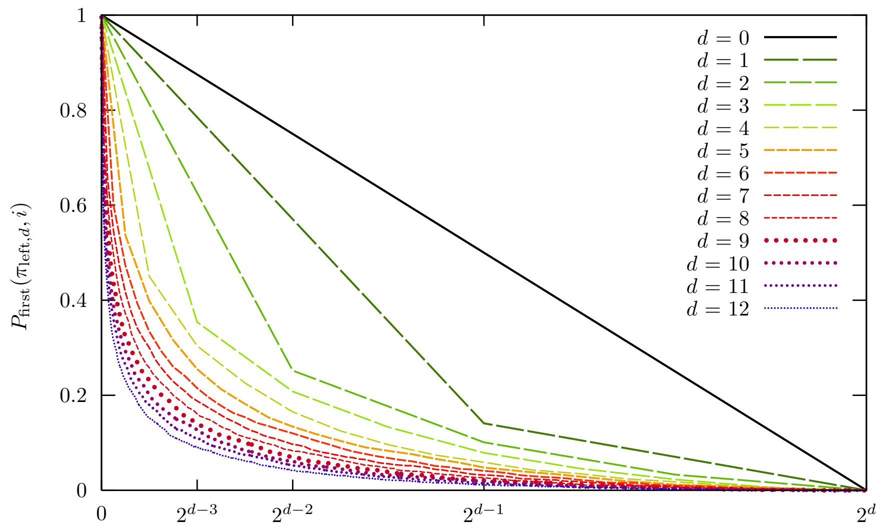

Figure 6.10 — Influence of the jump depth d on the first solution probability $P_{\text{first}}(\pi_{\text{left},d}, \text{i})$ using march_ks on 10.000 random 3-SAT formulae with $n = 400$ and $\rho = 4.26$.

6.5 Results

Distribution jumping is one of the main new features in the march SAT solver resulting in version march_ks which participated at the SAT 2007 competition: The technique is implemented as discussed above. First, using the dynamic jump depth heuristic d is computed. Second, the solver jumps using $\pi_{\text{left},d}$.

6.5.1 Random 3-SAT

To examine the actual speed-up resulting from distribution jumping, we run both march_ks$^-$ and march_ks on 10.000 random 3-SAT formulae with $n = 400$ and $\rho = 4.26$. The results are shown in Figure 6.11. On most of these instances, the adaptive jump depth for march_ks was set to 15. On satisfiable formulae, the performance of march_ks is much better. Although march_ks$^-$ is faster on several instances, on average it takes about 3.5 times more effort to compute the first solution. Regarding the extremes: The largest performance gain on a single instance is a factor 744 (from 81.94 to 0.11 seconds), while the largest performance loss is a factor 37 (from 35.52 to 0.95 seconds).

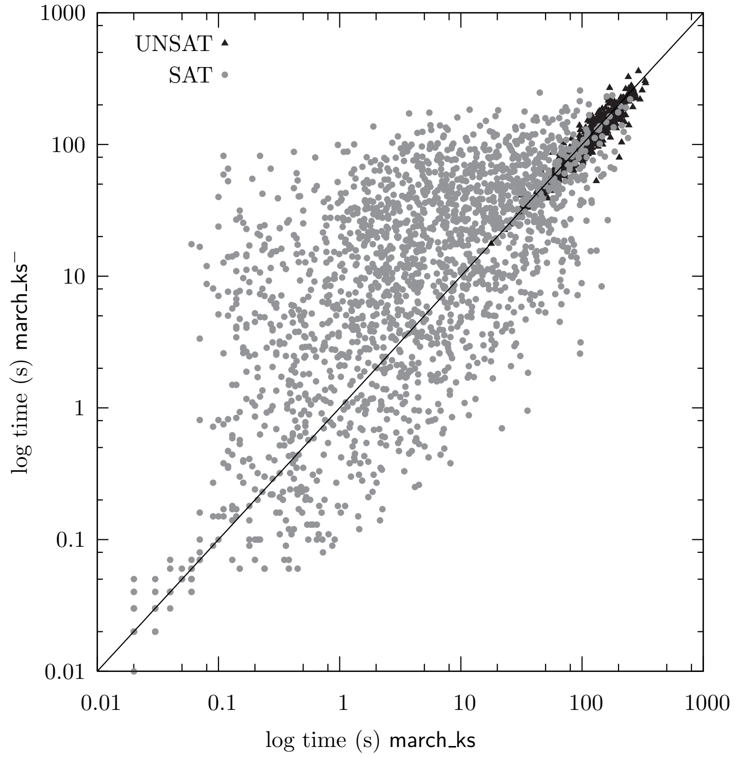

Figure 6.11 — Performance comparison (logscale) between march_ks$^-$ and march_ks on 10.000 random 3-SAT formulae with $n = 400$ and $\rho = 4.26$

The performance of march_ks$^-$ and march_ks on unsatisfiable instances is comparable. Due to the use of some adaptive heuristics in these SAT solvers, they may not explore exactly the same search-tree. It appears that this could result in a computational cost difference of about 10 % in favor of either march_ks$^-$ or march_ks. The overhead costs in march_ks are only marginal.

Large random 3-SAT formulae with density 4.26 appeared still hard for our current implementation. So, for our experiments we generated random 3-SAT formulae for four different sizes - 600 and 700 variables both at density 4.0 and 4.1 - to test the improvement of march_ks. We compared the performance of our solvers with R$^+$AdaptNovelty$^+$ [APSS05] - an incomplete (local search) solver which appeared the strongest on these kinds of formulae during the SAT 2005 competition [LS06]. In general, incomplete SAT solvers are dominant on satisfiable random formulae, but they fail to compete with complete SAT solvers on most structured satisfiable instances.

Table 6.5 — Number of solved (#SAT) and unknown (#UNK) instances within a timeout of 600 seconds

		march_ks$^-$		march_ks		R$^+$AdaptNovelty$^+$	
n	ρ	#SAT	#UNK	#SAT	#UNK	#SAT	#UNK
600	4.0	86	14	100	0	100	0
700	4.0	75	25	100	0	100	0
600	4.1	73	27	100	0	100	0
700	4.1	15	85	90	10	100	0

Table 6.5 shows the results of this experiment. The progress from march_ks$^-$ to march_ks is clear. However, R$^+$AdaptNovelty$^+$ is still more successful on these instances.

6.5.2 SAT Competition 2007

March_ks participated at the SAT Competition 2007. It won the satisfiable crafted category and several other awards. Especially in the former category, distribution jumping contributed to the success of the solver. Without this feature it would not have won.

Four crafted benchmarks `connm-ue-csp-sat-n800-d-0.02` (connamacher), `QG7a-gensys-ukn005`, `QG7a-gensys-brn004`, and `QG7a-gensys-brn100` (quasigroup) are not solved by march_ks without distribution jumping in 12 hours. However, the competition version which includes this feature solves these instances in 306, 422, 585, 3858 seconds, respectively. The dynamic jump depth heuristic selects $d = 12$ for the connamacher instance and $d = 26$ for the quasigroup instances. The solutions were found with only one or two right branches (above the jump depth). Various other solvers were able to solve the quasigroup instances within the 1200 seconds timeout. However, the connamacher instance was only solved by march_ks and satzilla.

Two other hard crafted benchmarks, ezfact64-3 and ezfact64-6, both factorization problems, where solved by march_ks in the second round in 3370 and 2963 seconds, respectively. Only minisat was also able to solve ezfact64-3, while ezfact64-6 was exclusively solved by march_ks. Without distribution jumping, solving these instances requires about twice the computational time. Since the timeout in the second round was 5000 seconds, march_ks$^-$ would not have solved them. Therefore, if march_ks$^-$ would have participated in the SAT 2007 competition - instead of march_ks - it would not have won the crafted satisfiable category, because it would not have solved these six benchmarks.

6.6 Conclusions

We observed that both march_ks and kcnfs bias on random k-SAT formulae - due to the used direction heuristics - the distribution of solutions towards the left branches. We introduced a measurement called the satisfying subtree bias B_{sat} to quantify the bias. Using B_{sat} the bias of the direction heuristics used in different solvers can be compared.

To capitalize on these observations we developed the new jumping strategy *distribution jumping*. While alternative jump strategies examine a random new part of the search-tree, our proposed method jumps towards a subtree that has a high probability of containing a solution.

Distribution jumping has been implemented in march_ks. With this new feature, the SAT solver can solve significantly more satisfiable random k-SAT formulae without hurting its performance on unsatisfiable instances. Despite the progress, march_ks is still outperformed by incomplete solvers on these benchmarks.

Yet, also the performance of march_ks is improved on satisfiable structured formulae. Apparently, distribution jumping is applicable outside the experimented domain. Thanks to this new technique, march_ks won the satisfiable crafted category of the SAT 2007 Competition.

The usefulness of distribution jumping could be even further increased by better direction heuristics: The more biased the distribution of solutions, the larger the expected speed-up. Also, the results of the greedy jump orders suggest that there is an opportunity to improve the generalized jump order.

6.A Solution distribution histograms

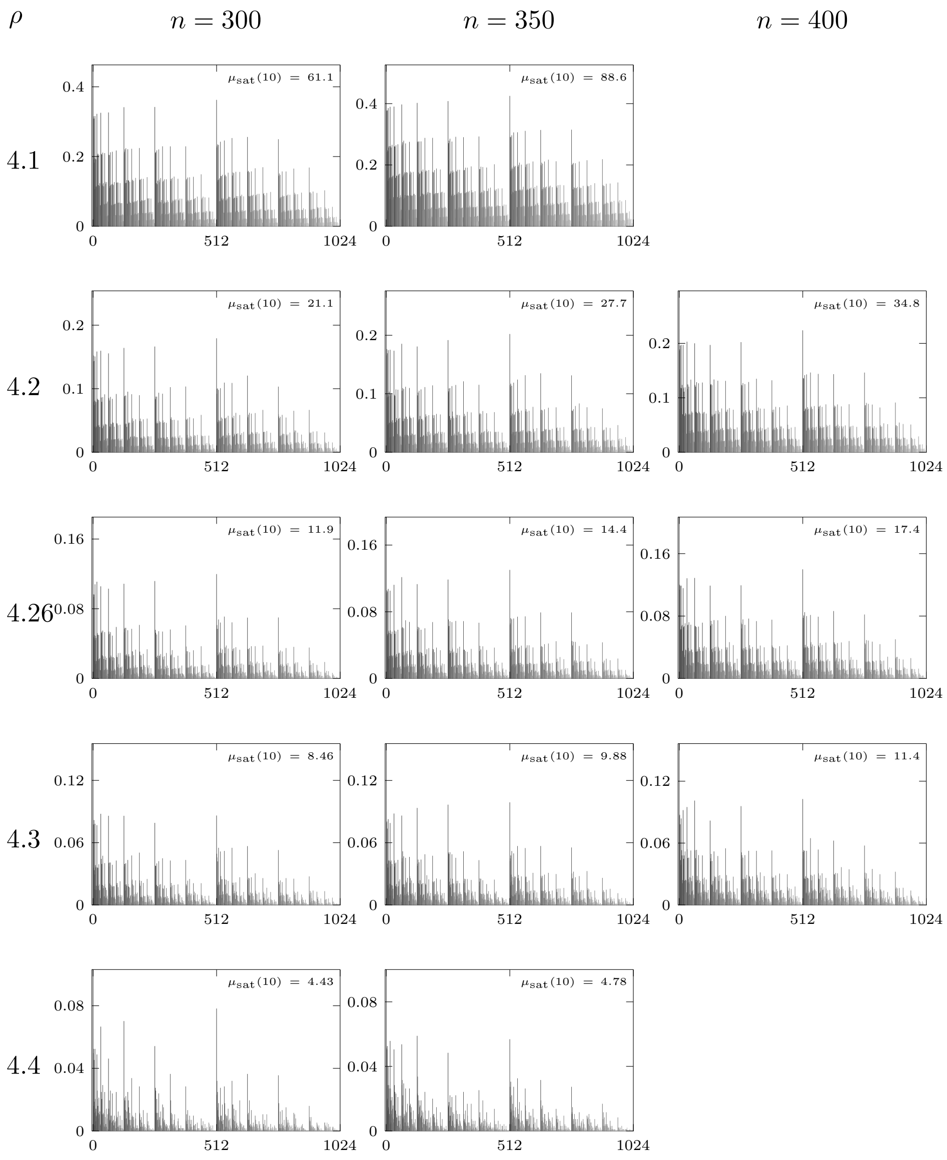

Figure 6.12 — Solution distribution plots showing $P_{\text{sat}}(10, i)$ (y-axis) of random 3-SAT formulae with $n \in \{300, 350, 400\}$ and $\rho \in \{4.1, 4.2, 4.26, 4.3, 4.4\}$ using march_ks.

6.B P_{sat}/ B_{sat} trees

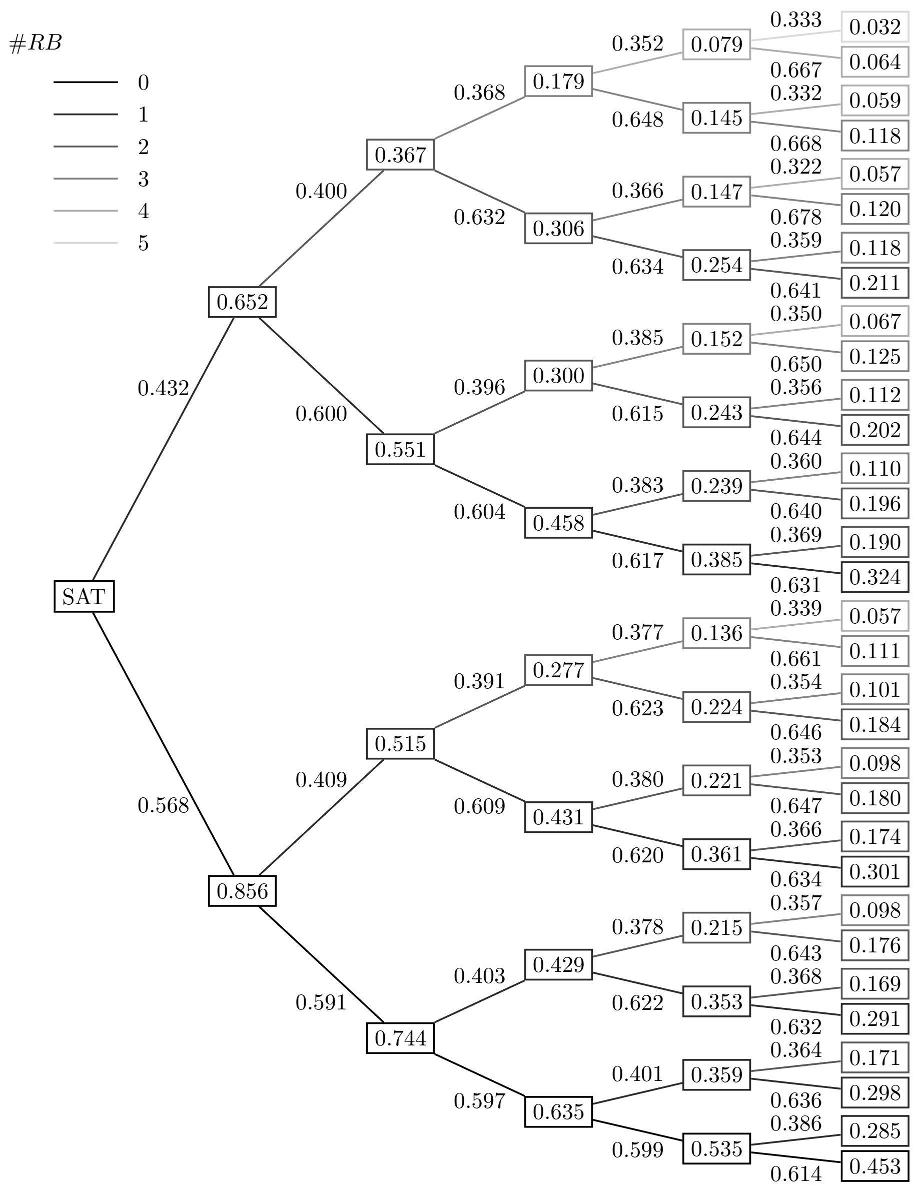

Figure 6.13 — P_{sat} / B_{sat} tree of march_ks$^-$ running on random 3-SAT with $n = 350$ and $\rho = 4.26$. $P_{\text{sat}}(d, i)$ values are shown in the vertices and $B_{\text{sat}}(d, i)$ values are shown on the edges.

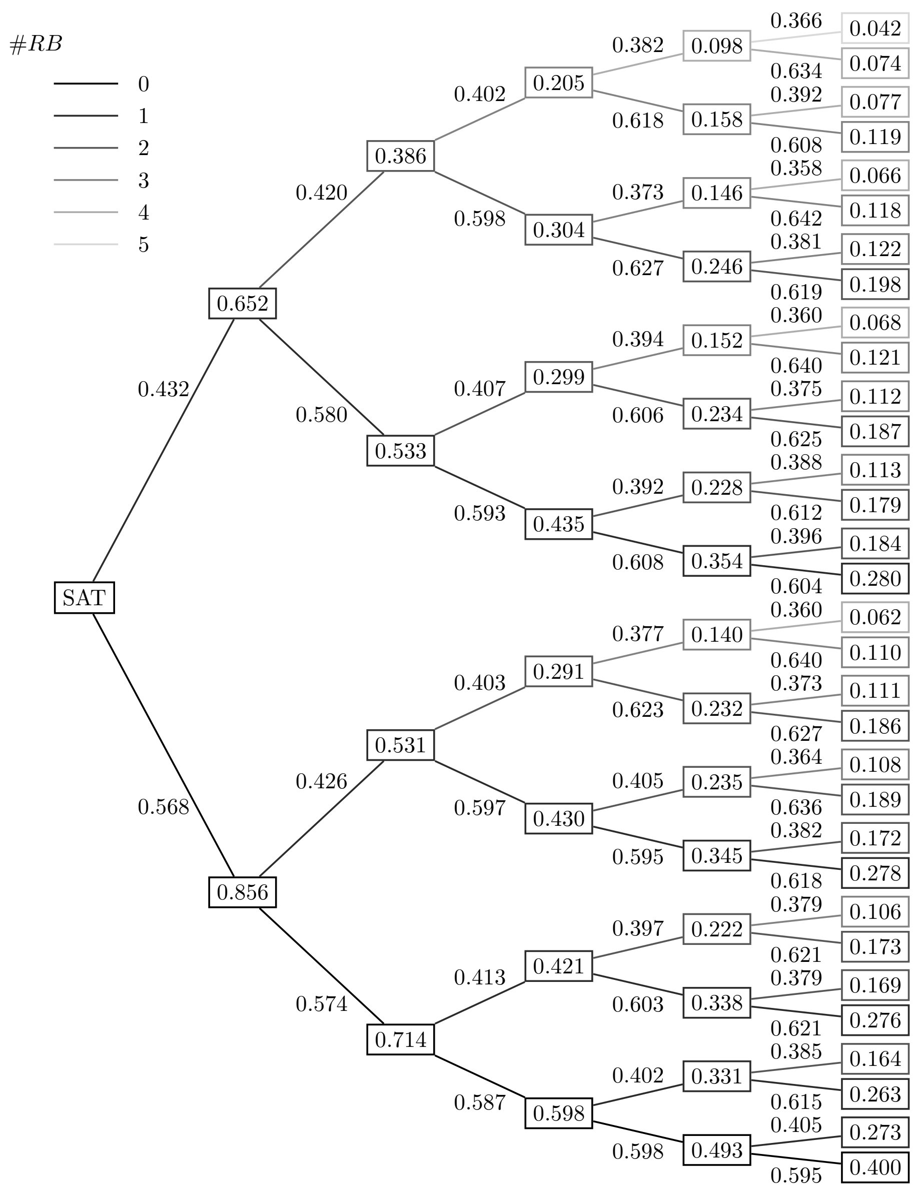

Figure 6.14 — $P_{\text{sat}}/B_{\text{sat}}$ tree of kcnfs running on random 3-SAT with $n = 350$ and $\rho = 4.26$. $P_{\text{sat}}(d, i)$ values are shown in the vertices and $B_{\text{sat}}(d, i)$ values are shown on the edges.

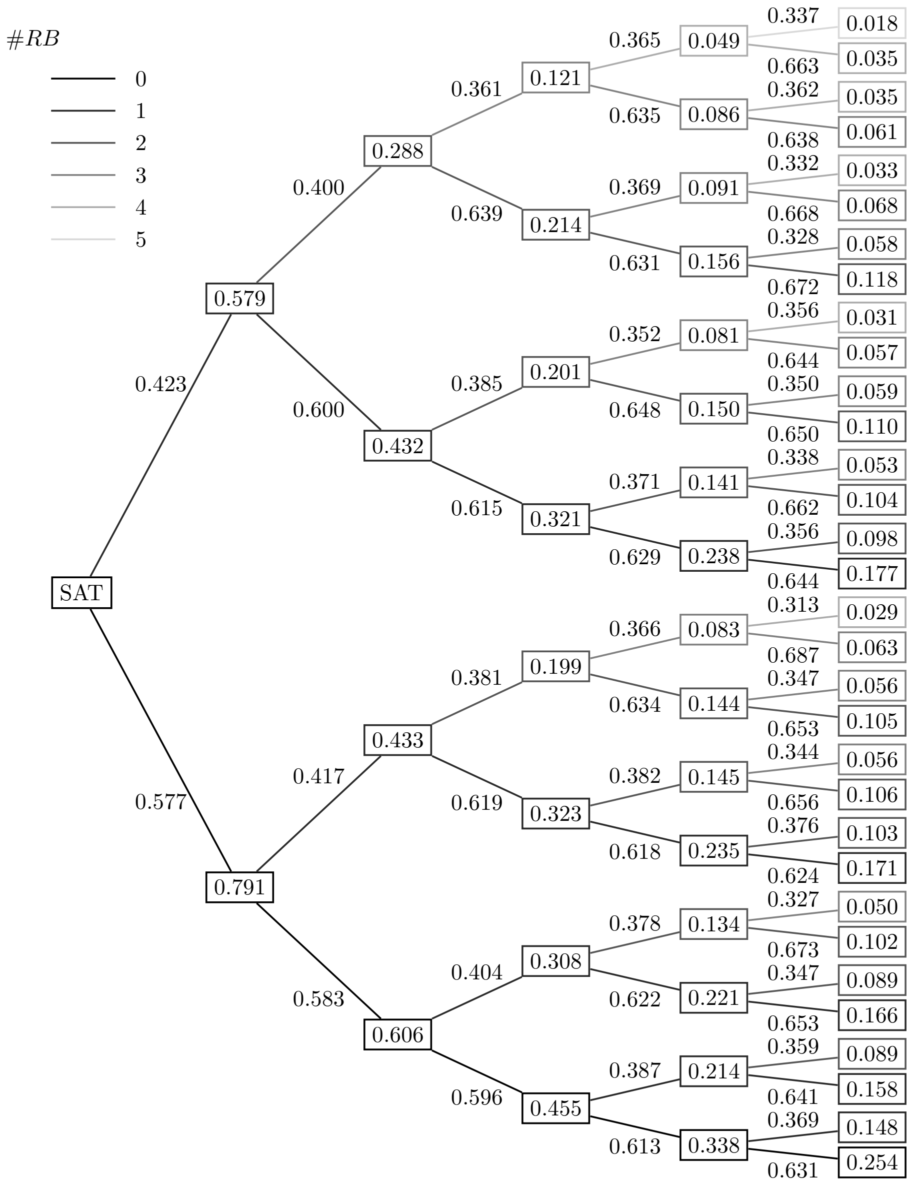

Figure 6.15 — $P_{\text{sat}}/B_{\text{sat}}$ tree of march_ks$^-$ running on random 4-SAT with $n = 120$ and $\rho = 9.9$. $P_{\text{sat}}(d, i)$ values are shown in the vertices and $B_{\text{sat}}(d, i)$ values are shown on the edges.

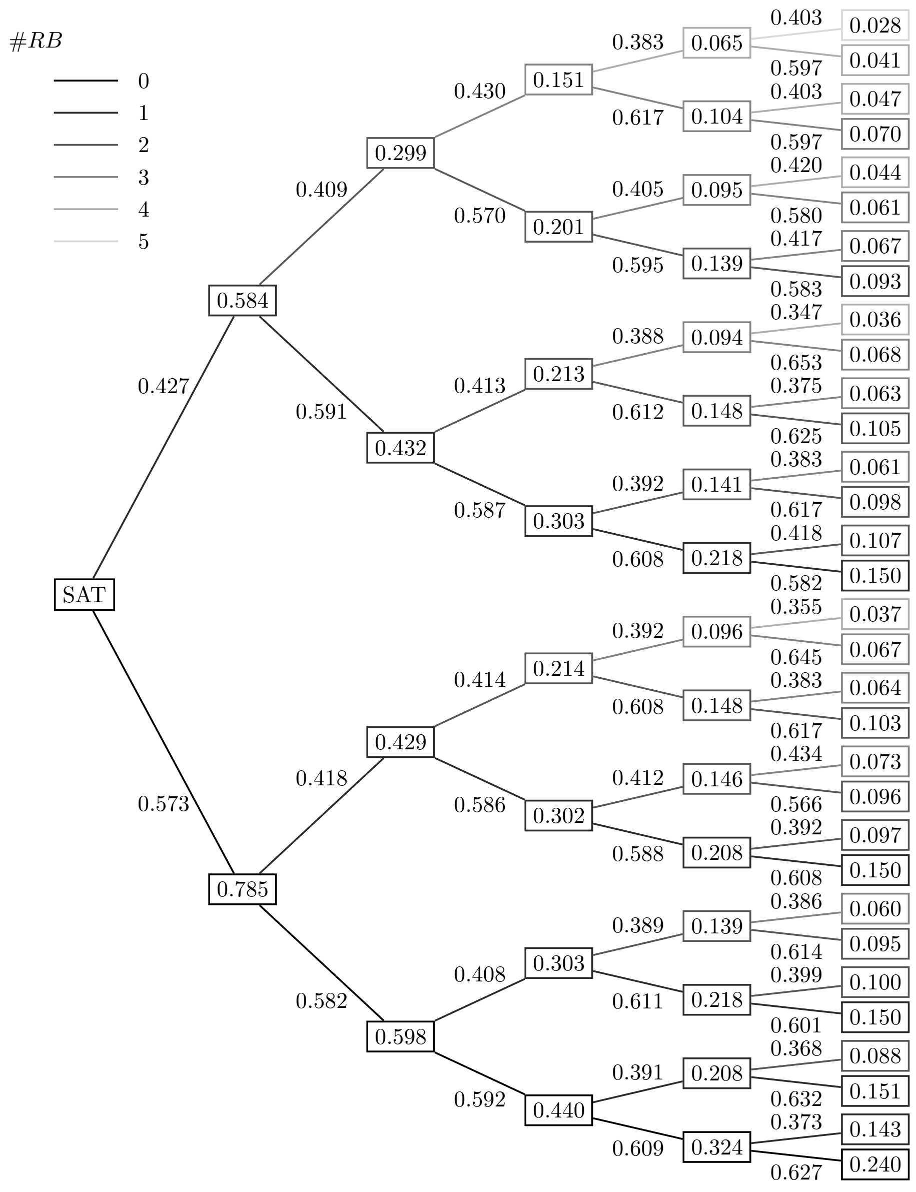

Figure 6.16 — $P_{\text{sat}}/B_{\text{sat}}$ tree of kcnfs running on random 4-SAT with $n = 120$ and $\rho = 9.9$. $P_{\text{sat}}(d, i)$ values are shown in the vertices and $B_{\text{sat}}(d, i)$ values are shown on the edges.

Gutta cavat lapidem,
non vi, sed saepe cadendo

A water drop hollows a stone,
not by the force,
but by falling often.

Ovid

7 UnitMarch[*]

Consider a variation of the Maniac Mansion example, where you search with several people at the same time - as in the computer game. The potential advantage of being with a group comes with two major disadvantages: First, all members of the group are lazy. They can search rooms simultaneously, but nobody wants to do more than the others. So, they wait to search the next room until everybody has finished their current room. Especially, when rooms differ a lot in size, the advantage of being with a bunch of people decreases significantly.

The second disadvantage is group dynamics: As soon as two or more people are together in the same room, the ambiance becomes more social and the work morale decreases. Regardless of the number of people in the room, the total work effort reduces to that of a single person. Moreover, once people are together, they stick together. With a bit of bad luck, all may end up in the same room.

To avoid this, two forms of communication are allowed. First, as soon as two people enter the same room, one is forced to get out and continue his search in a random other room; this counters group dynamics. Second, imagine that you require multiple keys to unlock the front door. This mean that there are many "subproblems" to be solved. If you search alone, this does not really affect your strategy because you just continue to search until all necessary keys are found. But, searching with a group now becomes even more effective: Different keys can be found by different people. Here it is crucial that each person reports is results, so that everybody will stay up-to-date.

We developed an incomplete SAT solver UnitMarch based on the UNITWALK algorithm. A disadvantage of this algorithm is that each step (room) is very expensive. To reduce the burden of the search costs, we parallelized the algorithm such that it can search multiple steps (rooms) simultaneously. However, the problems that arise are similar to the disadvantages of the lazy group. Yet by communicating if searches occur in the same space and by reporting a solution for a subproblem, the resulting parallel algorithm is faster than the original one.

[*]This chapter is based on: Marijn J.H. Heule and H. van Maaren. *Parallel SAT Solving using Bit-level Operations*, Accepted for the Journal on Satisfiability, Boolean Modeling and Computation. This article is an extended version of Marijn J.H. Heule and Hans van Maaren. *From Idempotent Generalized Boolean Assignments to Multi-bit Search.* SAT 2007 Springer LNCS **4501** (2007), pp 134-147.

7.1 Introduction

State-of-the-art satisfiability (SAT) solvers can be divided into complete (solving both satisfiable and unsatisfiable formulae) and incomplete (solving only satisfiables) ones. The former class of solvers uses fast data-structures and reasoning techniques on partial assignments to solve problems. Surprisingly, they also dominate performance of incomplete solvers on most satisfiable structured instances[22.]. Incomplete SAT solvers, mostly based on local search, mainly perform modifications on a (full) assignment using "randomized" flipping decisions. In general, these solvers are less complex. Incomplete solvers are very strong on satisfiable random benchmarks.

Todays 32/64 bit architecture enables computers to perform 32 or 64 of the familiar Boolean operations within a single clock cycle. Since assignment modifications can be considered Boolean operations, multiple of those modifications can be parallelized. Incomplete SAT solvers seem the most likely candidates to apply this technique, because they do not use reasoning techniques and because assignment modifications are an important aspect of the used algorithms.

Current SAT solvers do not make use of the opportunity of a p-bit processor to simulate parallel 1-bit (Boolean) search on p 1-bit processors. Conventional parallel SAT solving [BSK03, BS96, ZBH96] differs from the proposed method in Section 7.3: The former gains performance by dividing the workload over multiple processors and by some minor changes to the solving algorithm, while the latter uses a single processor and requires significant modifications to the algorithm. The most closely related work [IKM$^+$02] also parallelizes a SAT solver (GSAT), on a single processor. However, they use a vector processor (used in most supercomputers), instead of scalar processor (used in most desktop computers).

SAT solvers that use integer type of heuristics frequently (counters for instance), are not very suitable for modification in this respect. However, SAT solvers whose computational "center of gravity" consists of propagating truth values (or other 1-bit operations) may profit from this opportunity. One of such solvers is the state-of-the-art local search SAT solver UnitWalk [HK05]. We show that UnitWalk can be upgraded using a single p-bit processor. This results in a considerable speed-up.

7.2 Big Boolean Algebras

A Boolean Algebra is a six tuple $(B, \wedge, \vee, \neg, \mathbf{0}, \mathbf{1})$ in which set B contains all elements in the Algebra, $\wedge$ and $\vee$ are two binary operators, $\neg$ is a unary operator and $\mathbf{0}$ and $\mathbf{1}$ are constants. For a Boolean Algebra all the Boolean laws are obeyed.

[22.] Based on the results on the SAT competitions. See `www.satcompetition.org` for details.

The Boolean laws are (for $x, y, z \in B$):

- Commutative law: $x \vee y = y \vee x$, $x \wedge y = y \wedge x$
- Associative law: $x \vee (y \vee z) = (x \vee y) \vee z$, $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
- Distributive law: $x \vee (y \wedge z) = (x \wedge y) \vee (x \wedge z)$, $x \wedge (y \vee z) = (x \vee y) \wedge (x \vee z)$
- Absorption law: $x \vee (x \wedge y) = x$, $x \wedge (x \vee y) = x$
- Complement law: $x \vee \neg x = \mathbf{1}$, $x \wedge \neg x = \mathbf{0}$

The conventional Boolean Algebra B consists only of the two constants $\mathbf{0}$ and $\mathbf{1}$. The operators $\wedge$, $\vee$, $\neg$ are defined as logical `AND`, `OR` and `NOT` operators, respectively. We refer to a *big Boolean Algebra* if B consists of more than the two constants and the operators are defined such that all the Boolean axioms are obeyed. A Boolean Algebra can be extended with other constants (e.g. 3, 4) or with variables (e.g. x, y) as long as the operators are properly defined.

7.2.1 p-bit Boolean Algebras

Throughout this chapter, we will focus on a specific type of big Boolean Algebras, which we refer to as *p-bit Boolean Algebras*.

Definition The p-bit Boolean Algebra is a Boolean Algebra with $B = \{0,1\}^p$, $\mathbf{0} = (0, 0, \ldots, 0)$, $\mathbf{1} = (1, 1, \ldots, 1)$. The operators $\wedge$, $\vee$, $\neg$ are defined as *bitwise* logical `AND`, `OR` and `NOT` operators, respectively.

The p-bit Boolean Algebra can be seen as a free product of copies of the classical two valued one.

Example 7.1

Consider the 3-bit Boolean Algebra. We abbreviate multi-bit Booleans (the elements of p-bit Boolean Algebras): $(0, 1, 0)$ will be represented by `010`. Let $\mathcal{F}$ be the formula

$$\neg(x \rightarrow (y \vee (x \wedge z))) \qquad \lfloor 7.1$$

which is equivalent to *Conjunctive Normal Form* (CNF)

$$x \wedge \neg y \wedge (\neg x \vee \neg z) \qquad \lfloor 7.2$$

and assigning $x :=$ `101`, $y :=$ `001` and $z :=$ `111`, we calculate

$$\texttt{101} \wedge \neg\texttt{001} \wedge (\neg\texttt{101} \vee \neg\texttt{111}) = \mathbf{0} \qquad \lfloor 7.3$$

By assigning $x :=$ `101`, $y :=$ `001` and $z :=$ `011` however, $\mathcal{F}$ evaluates to the value `100`, as the reader may verify. All non-zero multi-bit Boolean outcomes verify that the given formula is satisfiable (so called “completeness”of Boolean Algebras) [Bro90].

At this point, the reader should realize that if we deal with CNF representations of formulae a big Boolean Algebra looses a bit of its interest: If a certain clause gets a `0` in some bit position (by some partial multi-bit assignment) there is no possibility to extract a satisfying assignment from this bit position, because the multiplications (the `AND`'s of the CNF) in this bit position can never undo this "being zero"!

Example 7.2

Consider the 2-bit Boolean Algebra and the formula $x \wedge y$. The reader may check that there are 16 possible 2-bit Boolean assignments of which 7 evaluate to a non-zero multi-bit Boolean. Drawing *multi-bit Boolean assignments* (MBA) randomly, the probability of hitting a non-zero multi-bit Boolean outcome is $\frac{7}{16}$, while in the conventional Boolean situation this probability is $\frac{1}{4}$. In general, the probability is $1 - (\frac{3}{4})^p$ using the p-bit Boolean Algebra.

The above example shows that probability to hit a solution using random sampling MBA's increases using a larger big Boolean Algebra. In case multi-bit Booleans can be used in approximately the same computational time as normal Booleans, solutions can be found faster. This is done in [KKM96], where Boolean "patterns" (rather than Booleans) are propagated through a circuit to increase the probability of hitting a solution - indicating an error in their application.

Although this random sampling can be considered a rather straight forward parallelism, we claim that efficient multi-bit propagation for SAT solving is not straight forward at all: In [KKM96], at each step, variables are either unassigned or assigned a *full* Boolean pattern, while in the proposed propagation variables can also be assigned a *partial* Boolean assignment.

Example 7.3

The sets of idempotents in finite rings of integers form a big Boolean Algebra. This Boolean Algebra has some exotic properties: The operators $x \wedge y$ and $\neg z$ are defined as the arithmetical operations $x \times y$ and $1 - z$, respectively. See for details [HvM07].

7.2.2 Generic MBA's

For any formula with n variables, there exist 2^n different Boolean assignments. Using the 2^n bit Boolean Algebra, we can associate each of these 2^n Boolean assignments to some bit position of a MBA. We call such a MBA, a *generic multi-bit Boolean assignment.* An example of a generic MBA for 3 variables is:

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

Let $\mathcal{F}$ be a formula on n variables and consider the 2^n bit Boolean Algebra. From the definition follows that this Algebra contains 2^{2^n} elements. This is the same number as the number of logically independent Boolean functions on n variables. In fact, it is not hard to demonstrate that in the above situation, a MBA to the variables exists such that each formula on n variables evaluates to its associated multi-bit Boolean, each multi-bit Boolean representing an equivalence class of Boolean functions.

Example 7.4

Consider the Boolean functions with 2 variables and the 4-bit Boolean Algebra. A generic MBA is for instance $x = \texttt{0011}$, $y = \texttt{0101}$. In this case $x \wedge \neg y$ evaluates to `0010`, $\neg(\neg x \wedge y)$ to `1011`, $x \leftrightarrow y$ to `1001`, $\neg(x \leftrightarrow y)$ to `0110` and $(x \vee y) \leftrightarrow \neg y$ to `0010`. In this case, every formula on 2 variables can be checked on feasibility by propagating the values $x = \texttt{0011}$ and $y = \texttt{0101}$, and only the outcome **0** (or `0000`) reflects a contradiction. That $(\texttt{0011}, \texttt{0101})$ is generic follows from the fact that all four possible assignments to 2 variables - (0,0), (0,1), (1,0) and (1,1) - are represented in the four bit positions of the MBA. Because $x \wedge \neg y$ and $(x \vee y) \leftrightarrow \neg y$ evaluate to the same multi-bit Boolean (using the same generic MBA), we can conclude that these two Boolean functions are logically equivalent. Table 7.1 shows the multi-bit Boolean identifiers for all Boolean functions using this generic MBA.

Table 7.1 — Multi-bit Boolean identifiers for Boolean functions with two variables using generic MBA x = `0011` and y = `0101`.

Boolean function	multi-bit Boolean
equivalence class	identifier
$\mathbf{0}$	0000
$x \wedge y$	0001
$x \wedge \neg y$	0010
x	0011
$\neg x \wedge y$	0100
y	0101
$\neg x \leftrightarrow y$	0110
$x \vee y$	0111
$\neg x \wedge \neg y$	1000
$x \leftrightarrow y$	1001
$\neg y$	1010
$x \vee \neg y$	1011
$\neg x$	1100
$x \vee \neg y$	1101
$\neg x \vee \neg y$	1110
$\mathbf{1}$	1111

A fantasy application of the above could be:

- Suppose a secret formula is kept of which we only know that it contains n variables;
- Suppose arbitrarily size multi-bit Booleans are allowed as input for the variables;
- Suppose that we are allowed to make one single guess on its feasibility by assigning a value to each of its input variables.

In this situation, the input of a generic MBA reveals the feasibility by checking whether the outcome is a non-zero multi-bit Boolean. If we allow large multi-bit Booleans as input to the propagation process, formulae can be solved on feasibility in one run. Notice that in this case we just exchange time for space.

Working with MBA's can be beneficial in situations when the multiple Boolean operations involved can be performed in a single clock cycle - as in modern computers. More specifically: If a 32-bit processor is available, formulae with up to 5 variables can be resolved in one propagation run using generic MBA's in about the same time an ordinary Boolean assignment is propagated.

7.3 Multi-Bit Unit Propagation

This section describes the use of MBA's to parallelize a SAT solving algorithm. However, this differs from conventional parallelism: Modifications of MBA's can be processed in parallel, while, for instance, operations on counters cannot. In general, only 1-bit operations can be parallelized. Therefore, algorithms that potentially benefit from MBA's should have their computational "center of gravity" on assignment modifications.

A widely used procedure for assignment modifications is *unit propagation*: Given a formula $\mathcal{F}$ and an assignment φ. If φ applied to $\mathcal{F}$ (denoted by $\varphi \circ \mathcal{F}$) contains *unit clauses* (clauses of size 1) then the remaining literal in each unit clause is forced to be true - thereby expanding φ. This procedure continues until there are no unit clauses left in $\varphi \circ \mathcal{F}$. This section describes a SAT solving algorithm that uses unit propagation at its computational "center of gravity".

The UnitWalk algorithm.

For a possible application we focused on local search (incomplete) SAT solvers. In contrast to complete SAT solvers, they are less complicated and work with full assignments. A generic structure of local search SAT solvers is as follows: An assignment φ is generated, earmarking a random Boolean value to all variables. By flipping the truth values of variables, φ can be modified to satisfy as many clauses as possible of the formula at hand. If after a multitude of flips φ still does not satisfy the formula, a new random assignment is generated.

Most local search SAT solvers use counting heuristics to flip the truth value of the variables in a turn-based manner. These heuristics appear hard to parallelize on a single processor. However, the UNITWALK algorithm [HK05] is an exception. Instead of counting heuristics, it uses unit propagation to flip variables. The UnitWalk SAT solver - based on this algorithm - is the fastest local search SAT solver on many structured instances and won the SAT 2003 competition in the category *All random SAT* [LS03].

The UNITWALK algorithm (see Algorithm 7.1) flips variables in so-called *periods*: Each period starts with an initial assignment (referred to as master assignment φ_{master}), an empty assignment φ_{active} and a random order of the variables π. First, unit propagation is executed on the empty assignment. Second, the first unassigned variable in π is assigned to its value in φ_{master}, followed by unit propagation of this value. A period ends when all variables are assigned a value in φ_{active}. Notice that *conflicts* - clauses with all literals assigned to false - are more or less neglected, depending on the implementation. A new period starts with the resulting φ_{active} as φ_{master} and a new order of the variables.

Algorithm 7.1 FLIP_UNITWALK(φ_{master})

```
for $i$ in 1 to MAX_PERIODS do
    if $\varphi_{\text{master}}$ satisfies $F$ then
        break
    end if
    $\pi$ := random order of the variables
    $\varphi_{\text{active}} := \{x_i = *\}$
    for $j$ in 1 to $n$ do
        while unit clause $u \in \varphi_{\text{active}} \circ F$ do
            $\varphi_{\text{active}}[\ VAR(u)\ ] := TRUTH(u)$
        end while
        if $x_{\pi(j)}$ not assigned in $\varphi_{\text{active}}$ then
            $\varphi_{\text{active}}[\ x_{\pi(j)}\ ] := \varphi_{\text{master}}[\ x_{\pi(j)}\ ]$
        end if
    end for
    if $\varphi_{\text{active}} = \varphi_{\text{master}}$ then
        random flip variable in $\varphi_{\text{active}}$
    end if
    $\varphi_{\text{master}} := \varphi_{\text{active}}$
end for
return $\varphi_{\text{master}}$
```

Example 7.5

Consider the example formula and initial settings below. Unassigned values in φ_{active} are denoted by *.

$$\begin{aligned}
\mathcal{F}_{\text{example}} &:= (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \\
&\quad\ (\neg x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_4) \\
\varphi_{\text{master}} &:= \{x_1 = \texttt{0}, x_2 = \texttt{1}, x_3 = \texttt{1}, x_4 = \texttt{0}\} \\
\varphi_{\text{active}} &:= \{x_1 = *, x_2 = *, x_3 = *, x_4 = *\} \\
\pi &:= (x_2, x_1, x_4, x_3)
\end{aligned}$$

Since the formula contains no unit clauses, the algorithm starts by selecting the first variable from the random order - x_2. We assign this variable to true (as in φ_{master}) and perform unit propagation. Due to $\neg x_2 \vee \neg x_3$ this results in one unit clause $\neg x_3$. Propagation of this unit clause - assigning x_3 to false - results in unit clauses x_4, and $\neg x_4$. Because two complementary unit clauses have been generated we found a conflict. However, the UNITWALK algorithm does not resolve this conflict.

Instead, it continues by selecting[23.] one of them, say $\neg x_4$, and assign x_4 to false. After this assignment $\varphi_{\text{active}} \circ \mathcal{F}$ does not contain unit clauses anymore. We conclude this period by assigning x_1 to its value in φ_{master}. This results in the full assignment $\varphi_{\text{active}} = \{x_1 = \mathtt{0}, x_2 = \mathtt{1}, x_3 = \mathtt{0}, x_4 = \mathtt{0}\}$. Notice that the new assignment does not satisfy clause $\neg x_2 \vee x_3 \vee x_4$.

Now, consider the same example, this time using a 4-bit assignment to all the variables. The reader must keep in mind that by parallelizing the former, we aim to satisfy clauses in each bit position! Recall that once a certain clause gets a 0 at some bit position, no satisfying assignment is possible at that bit position. Hence, variables may be flipped in multiple bits, and "conflict" means a conflict in some bit position. For the latter we shall use the term *bit-conflict*. In the multi-bit case, a clause is called unit with respect to a certain bit position if at that bit position one literal is unassigned and all others are falsified. So, a clause can be(come) unit on multiple bit positions and on different literals at the same time. Further, we keep using the term "truth value" for its multi-bit analogue. Notice that in the initial settings below, the first bit in φ_{master} equals the 1-bit example and that the ordering is the same.

$$\begin{aligned}
\varphi_{\text{master}} &:= \{x_1 = \mathtt{0110}, x_2 = \mathtt{1100}, x_3 = \mathtt{1010}, x_4 = \mathtt{0110}\} \\
\varphi_{\text{active}} &:= \{x_1 = {*}{*}{*}{*},\ x_2 = {*}{*}{*}{*},\ x_3 = {*}{*}{*}{*},\ x_4 = {*}{*}{*}{*}\} \\
\pi &:= (x_2, x_1, x_4, x_3)
\end{aligned}$$

Again, we start by assigning x_2 to its value in φ_{master} followed by unit propagation. This will result in two unit clauses :

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

One of them is selected, say x_1 and assigned to its value, resulting in:

$$(\neg x_1 = {*}{*}\mathtt{00} \vee x_2 = \mathtt{1100} \vee x_3 = \mathtt{00}{*}{*}) \quad\Rightarrow\quad x_3 := \mathtt{00}\mathbf{11}$$

Now we assign x_3 which triggers three clauses:

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

When unit propagation stops, only the first two bits of x_1 are still undefined. These bits are set to their value in φ_{master} assigning all variables. The period ends with $\varphi_{\text{active}} = \{x_1 = \mathtt{0111}, x_2 = \mathtt{1100}, x_3 = \mathtt{0011}, x_4 = \mathtt{0000}\}$ - which satisfies the formula in the third and fourth bit.

[23.] In [HK05] the authors suggest to select the truth value used in φ_{master}. However, this is not implemented in the latest version of the solver and we consider it as a choice.

The reader may check that: (1) The order in which unit clauses are propagated, as well as the order in which clauses are evaluated, is not fixed. In case conflicts occur, the order influences φ_{active}. For example, evaluating $\neg x_2 \vee x_3 \vee x_4$ before $\neg x_2 \vee x_3 \vee \neg x_4$ results in a different final φ_{active}. (2) In the 4-bit example the third and fourth bit are the same for all variables. This effect could reduce the parallelism, because the algorithm as such does not intervene here and in fact maintains this collapse. This effect is not restricted to formulas with a small number of variables. To counter this unwanted effect, we added a technique removing duplicates - see Section 7.5.1.

7.4 Implementation UnitMarch

7.4.1 Unit propagation

The UnitPropagation procedure within the UnitWalk algorithm is not confluent: Different implementations yield different results. In short, two design decisions need to be made:

- In case of multiple unit clauses: Which one to select for propagation;
- In case of a conflict: Whether or how to act.

The most recent UnitWalk (version 1.003) implements the following UnitPropagation procedure: Unit clauses are stored in a multi-set (a set that can contain duplicate elements) data-structure. For each iteration a random element u from the multi-set is selected. If the complement of the selected unit clause also occurs in the multi-set - indicating a conflict - all occurrences of u and $\neg u$ are removed from the multi-set. The algorithm continues with the next random element - see Algorithm 7.2. Notice that this is a defensive flip strategy: The truth value for u in φ_{active} tends to be copied from φ_{master}.

Algorithm 7.2 UnitPropagation_MultiSet ( )

```
1: while $UnitMultiSet$ is not empty do
2:     $u$ := random element from $UnitMultiSet$
3:     remove all occurrences of $u$ in $UnitMultiSet$
4:     if unit clause $\neg u$ also occurs in $UnitMultiSet$ then
5:         remove all occurrences of $\neg u$ in $UnitMultiSet$
6:     else
7:         $\varphi_{\text{active}}[\ VAR(u)\ ] := TRUTH(u)$
8:         for all clauses $C_i$ in which $\neg u$ occurs do
9:             if $C_i$ becomes a unit clause then
10:                add $C_i$ to $UnitMultiSet$
11:            end if
12:        end for
13:    end if
14: end while
```

In our implementation we took a slightly different approach, since the above algorithm was hard to implement efficiently in a multi-bit version. Instead of the multi-set we used a queue (first in, first out) data-structure - see Algorithm 7.3: Unit clauses are selected in the order in which they are added to the queue. In general, "early" generated unit clauses will have more bits assigned (at the time of propagation) compared to "recent" unit clauses. Therefore the queue seems a useful data-structure since it always propagates the "earliest" unit clause left.

In addition, conflicts are handled differently: The queue is not allowed to contain complementary or duplicate unit clauses. The truth value of the first generated unit clause will be used during the further propagation. Notice that this flip strategy is more offensive: Given a bit-conflict, the truth value of the variable is flipped in approximately half of the cases. As we will see in the results (Section 7.6), both implementations yield comparable results (the average number of periods).

Algorithm 7.3 UNITPROPAGATION_QUEUE ( )

```
1: while $UnitQueue$ is not empty do
2:     $u$ := removed front element from $UnitQueue$
3:     for all clauses $C_i$ in which $\neg u$ occurs do
4:         if $C_i$ becomes a unit clause then
5:             $v$ := remaining literal in $C_i$
6:             $\varphi_{\text{active}}[\ VAR(v)\ ] := TRUTH(v)$
7:             if $v$ not in $UnitQueue$ then append $v$ to $UnitQueue$
8:         end if
9:     end for
10: end while
```

7.4.2 Detection of Unit Clauses

The UNITWALK algorithm spends most computational time in detecting which clauses became unit clauses given an expansion of φ_{active}. If a variable is assigned a Boolean value, all clauses in which it occurs with complementary polarity are potential unit clauses. In a 1-bit implementation, a potential unit clause can only be unit on a single literal, while in a multi-bit implementation it can become unit on multiple literals (each on a different bit position):

Example 7.6

Given $\varphi_{\text{active}} = \{x_1 = \texttt{010*}, x_2 = \texttt{10*1}, x_3 = \texttt{101*}, x_4 = \texttt{*001}\}$ with x_3 as remaining literal of a unit clause to be propagated and with potential clause $x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4$.

$$(x_1 = \texttt{010*} \vee \neg x_2 = \texttt{01*0} \vee \neg x_3 = \texttt{010*} \vee x_4 = \texttt{*001}) \Rightarrow x_2 := \texttt{1001},\ x_4 := \texttt{1001}$$

In general, a clause can become unit on all literals - apart from the propagation literal.

Encoding.

Since each bit in φ_{active} consists of three possible values (`*,0,1`), we used two bits to encode each value: `00 = *`, `01 = 0`, `10 = 1`, and `11 =` **bit-conflict**[24.]. We used an array φ_{-}^{+} in which both x_i and $\neg x_i$ have a separate assignment: The first bit of each value is stored in x_i while the second bit is stored in $\neg x_i$. Back to the example. φ_{active} is stored as:

$$\begin{cases} \varphi_{-}^{+}[\ \ x_1] = \texttt{0100}, & \varphi_{-}^{+}[\ \ x_2] = \texttt{1001}, \quad \varphi_{-}^{+}[\ \ x_3] = \texttt{1010}, \quad \varphi_{-}^{+}[\ \ x_4] = \texttt{0001} \\ \varphi_{-}^{+}[\neg x_1] = \texttt{1010}, & \varphi_{-}^{+}[\neg x_2] = \texttt{0100}, \quad \varphi_{-}^{+}[\neg x_3] = \texttt{0100}, \quad \varphi_{-}^{+}[\neg x_4] = \texttt{0110} \end{cases}$$

Using φ_{-}^{+} we can compute the unit clauses as below. Conflicts are ignored by only allowing unassigned bits - computed by $\texttt{NOT}(\varphi_{-}^{+}[x_i]\ \texttt{OR}\ \varphi_{-}^{+}[\neg x_i])$ - to be assigned. Back to the example:

$$\begin{aligned} x_1 &:= \varphi_{-}^{+}[x_3]\ \texttt{AND}\ \texttt{NOT}(\varphi_{-}^{+}[x_1]\ \texttt{OR}\ \varphi_{-}^{+}[\neg x_1])\ \texttt{AND}\ \varphi_{-}^{+}[x_2]\ \texttt{AND}\ \varphi_{-}^{+}[\neg x_4] \\ \neg x_2 &:= \varphi_{-}^{+}[x_3]\ \texttt{AND}\ \varphi_{-}^{+}[\neg x_1]\ \texttt{AND}\ \texttt{NOT}(\varphi_{-}^{+}[x_2]\ \texttt{OR}\ \varphi_{-}^{+}[\neg x_2])\ \texttt{AND}\ \varphi_{-}^{+}[\neg x_4] \\ x_4 &:= \varphi_{-}^{+}[x_3]\ \texttt{AND}\ \varphi_{-}^{+}[\neg x_1]\ \texttt{AND}\ \varphi_{-}^{+}[x_2]\ \texttt{AND}\ \texttt{NOT}(\varphi_{-}^{+}[x_4]\ \texttt{OR}\ \varphi_{-}^{+}[\neg x_4]) \end{aligned}$$

The above shows a potential disadvantage of the multi-bit propagation: To check whether a clause of size k becomes a unit clause and to determine the remaining literal(s) is not trivially computed in $\mathcal{O}(k)$ steps - as is the case with 1-bit propagation. However, a $\mathcal{O}(k)$ implementation can be realized by splitting the computation into two stages:

- Compute the *unit mask* of a clause - a multi-bit Boolean which is true on all positions with exactly one not falsified literal (denoted by $M_{\text{NF}=1}$) and false elsewhere;
- Use the unit mask to quickly determine the newly created unit clauses: All literals that are unassigned at a true position in the unit mask became unit.

To compute $M_{\text{NF}=1}$, we use two auxiliary masks, $M_{\text{NF}<1}$ and $M_{\text{NF}<2}$. The masks denote multi-bit Booleans which are `1` on all positions with less than one (and two, respectively) not falsified literals and `0` elsewhere. Notice that $M_{\text{NF}=1} := M_{\text{NF}<1}\ \texttt{XOR}\ M_{\text{NF}<2}$. For each literal l_i in a clause we update $M_{\text{NF}<1}$ and $M_{\text{NF}<2}$ by the following two rules:

$$\begin{aligned} M_{\text{NF}<2} &:= (M_{\text{NF}<2}\ \texttt{AND}\ \varphi_{-}^{+}[\ \neg l_{y,i}\ ])\ \texttt{OR}\ M_{\text{NF}<1} \\ M_{\text{NF}<1} &:= M_{\text{NF}<1}\ \texttt{AND}\ \varphi_{-}^{+}[\ \neg l_{y,i}\ ] \end{aligned}$$

[24.] The bit-conflict value is not possible within or implementation

Algorithm 7.4 ComputeUnitMask (clause C_y)

```
1: $M_{\mathrm{NF}\,<\,1}$ := ALL_BITS_TRUE, $M_{\mathrm{NF}\,<\,2}$ := ALL_BITS_TRUE
2: for $i$ in 1 to $|C_y|$ do
3:     $M_{\mathrm{NF}\,<\,2} := (M_{\mathrm{NF}\,<\,2}$ AND $\varphi^{+}_{-}[\ \neg l_{y,i}\ ])$ OR $M_{\mathrm{NF}\,<\,1}$
4:     $M_{\mathrm{NF}\,<\,1} := M_{\mathrm{NF}\,<\,1}$ AND $\varphi^{+}_{-}[\ \neg l_{y,i}\ ]$
5: end for
6: return $M_{\mathrm{NF}\,<\,1}$ XOR $M_{\mathrm{NF}\,<\,2}$
```

The implementation of the above is shown in Algorithm 7.4. Once $M_{\mathrm{NF}\,=\,1}$ is computed ($M_{\mathrm{NF}\,=\,1} = \texttt{1010}$ in the example) we can determine the newly create unit clauses. For the example we only need the computations:

$$\begin{aligned} x_1 &:= M_{\mathrm{NF}\,=\,1}\ \texttt{AND}\ \texttt{NOT}(\varphi^{+}_{-}[x_1]\ \texttt{OR}\ \varphi^{+}_{-}[\neg x_1]) \\ \neg x_2 &:= M_{\mathrm{NF}\,=\,1}\ \texttt{AND}\ \texttt{NOT}(\varphi^{+}_{-}[x_2]\ \texttt{OR}\ \varphi^{+}_{-}[\neg x_2]) \\ x_4 &:= M_{\mathrm{NF}\,=\,1}\ \texttt{AND}\ \texttt{NOT}(\varphi^{+}_{-}[x_4]\ \texttt{OR}\ \varphi^{+}_{-}[\neg x_4]) \end{aligned}$$

7.5 Communication

The above description of a multi-bit version of the UnitWalk algorithm can be seen as performing the algorithm in parallel without communication. However, communication can be added to the algorithm to possibly further extend performance gain. This section offers two kinds of communication. The first is a parallel detection algorithm for duplicate assignments and the second is a parallel algorithm to compute the largest autarky in a given (full) assignment.

7.5.1 Duplicate assignments

During our experiments we frequently observed convergence of the different bit positions in an assignment. For a given assignment φ, the j-th bit position is called a *duplicate* if there exists a $i < j$ such that all variables are assigned to the same truth value at bit position i and j. On most benchmarks, duplicates were observed. In some cases even (all) $n-1$ bit positions became duplicate. Due to the construction of the UnitWalk algorithm, once a bit position is a duplicate, it will remain a duplicate if no intervention is made. Because duplicates reduce the parallel behavior of the algorithm, we decided to detect duplicates and replace them with a new random assignment.

To detect the duplicates, we used *assignment matrices*: The assignment matrix $M_\varphi(x_i)$ of a variable x_i for a p-bit assignment φ is a symmetric $n \times n$ 0,1-matrix of which each j-th row and column is $\varphi[x_i]$ if x_i is assigned to true on the j-th bit-position and $\varphi[\neg x_i]$ otherwise. The assignment matrix $M_\varphi(\mathcal{F})$ is the entrywise product (so called Hadamard product, denoted by $\bullet$) of the the assignment matrices of all the variables in $\mathcal{F}$.

Example 7.7

Given $\varphi = \{x_1 = \texttt{010010}, x_2 = \texttt{101101}, x_3 = \texttt{110111}, x_4 = \texttt{000000}\}$. Now we compute the assignment matrices:

$$M_\varphi(x_1) = \begin{bmatrix} 1&0&1&1&0&1\\ 0&1&0&0&1&0\\ 1&0&1&1&0&1\\ 1&0&1&1&0&1\\ 0&1&0&0&1&0\\ 1&0&1&1&0&1 \end{bmatrix} \quad M_\varphi(x_2) = \begin{bmatrix} 1&0&1&1&0&1\\ 0&1&0&0&1&0\\ 1&0&1&1&0&1\\ 1&0&1&1&0&1\\ 0&1&0&0&1&0\\ 1&0&1&1&0&1 \end{bmatrix}$$

$$M_\varphi(x_3) = \begin{bmatrix} 1&1&0&1&1&1\\ 1&1&0&1&1&1\\ 0&0&1&0&0&0\\ 1&1&0&1&1&1\\ 1&1&0&1&1&1\\ 1&1&0&1&1&1 \end{bmatrix} \quad M_\varphi(x_4) = \begin{bmatrix} 1&1&1&1&1&1\\ 1&1&1&1&1&1\\ 1&1&1&1&1&1\\ 1&1&1&1&1&1\\ 1&1&1&1&1&1\\ 1&1&1&1&1&1 \end{bmatrix}$$

$$\Rightarrow \; M_\varphi(\mathcal{F}) = \begin{bmatrix} 1&0&0&1&0&1\\ 0&1&0&0&1&0\\ 0&0&1&0&0&0\\ 1&0&0&1&0&1\\ 0&1&0&0&1&0\\ 1&0&0&1&0&1 \end{bmatrix}$$

Notice that all assignment matrices $M_\varphi(x_i)$ have at least as many 1's as 0's. If a row contains 1's in the lower triangle of $M_\varphi(\mathcal{F})$, the corresponding bit position is a duplicate. In the example above, the 4-th, 5-th and 6-th bit positions are duplicates. Using $M_\varphi(\mathcal{F})$ we can obtain $m_{\text{duplicates}}$: Compute the Hadamard product of the strictly lower triangular matrix and $M_\varphi(\mathcal{F})$. Multiply the result with the all one vector. The resulting mask $m_{\text{duplicates}}$ is a p-bit Boolean which has 1's on all bit positions that are duplicates and 0's otherwise. In this example the computation is:

$$m_{\text{duplicates}} = \left(\begin{bmatrix} 0&0&0&0&0&0\\ 1&0&0&0&0&0\\ 1&1&0&0&0&0\\ 1&1&1&0&0&0\\ 1&1&1&1&0&0\\ 1&1&1&1&1&0 \end{bmatrix} \bullet \begin{bmatrix} 1&0&0&1&0&1\\ 0&1&0&0&1&0\\ 0&0&1&0&0&0\\ 1&0&0&1&0&1\\ 0&1&0&0&1&0\\ 1&0&0&1&0&1 \end{bmatrix} \right) \begin{bmatrix} 1\\1\\1\\1\\1\\1 \end{bmatrix} = \begin{bmatrix} 0&0&0&1&1&1 \end{bmatrix}$$

⌊7.4

In UnitMarch $m_{\text{duplicates}}$ is computed as in Algorithm 7.5 - for the p-bit Boolean Algebra. Let n denote the number of variables. Although the algorithm has worst case complexity $\mathcal{O}(pn)$, in practice it is quite fast due to the break command at line 11.

7.5.2 Autarkies

An *autarky* (or autark assignment) is a partial assignment φ that satisfies all clauses that are "touched" (have at least one literal assigned) by φ. So, all satisfying assignments are autark assignments. Autarkies that do not satisfy all clauses can be used to reduce the size of the formula: Let $\mathcal{F}_{\text{touched}}$ be the clauses in $\mathcal{F}$ that are satisfied by an autarky. The remaining formula $\mathcal{F}^* := \mathcal{F} \setminus \mathcal{F}_{\text{touched}}$

Algorithm 7.5 ComputeDuplicateMask (assignment φ)

```
1: $m_{\text{duplicates}} := [0]^n$
2: for $j$ in 1 to $n-1$ do
3:     $m_{\text{column}} := [0]^j[1]^{n-j}$
4:     for $x_i \in \mathcal{F}$ do
5:         if $x_i$ is assigned to true on the $j$-th bit-position in $\varphi$ then
6:             $m_{\text{column}} := m_{\text{column}}$ AND $\varphi[x_i]$
7:         else
8:             $m_{\text{column}} := m_{\text{column}}$ AND $\varphi[\neg x_i]$
9:         end if
10:        if $m_{\text{column}} = [0]^n$ then
11:            break
12:        end if
13:    end for
14:    $m_{\text{duplicates}} := m_{\text{duplicates}}$ OR $m_{\text{column}}$
15: end for
16: return $m_{\text{duplicates}}$
```

is satisfiability equivalent to $\mathcal{F}$. If we detect an autark assignment we can reduce $\mathcal{F}$ by removing all clauses in $\mathcal{F}_{\text{touched}}$.

Given a partial assignment, one can compute the largest autarky being a reduction of that assignment using the following algorithm [KMT07]:

- Loop through all the clauses;
- If a clause is touched but not satisfied, unassign all variables in that clause;
- Repeat the above until no assignment changes have been made.

The outcome of the algorithm is either an empty assignment, showing that there exists no autarky which is a reduction of the input assignment, or some variables are still assigned which form an autarky. Notice that the algorithm is confluent: All variables that occur in any autarky being a reduction of the input assignment will be in the output. The larger the number of assigned variables of the input assignment, the higher the probability that algorithm will return an autarky. Especially local search Sat solvers - such as UnitWalk - are likely to profit from the algorithm, since at each period they work with a full assignment.

The above algorithm can easily be parallelized using MBA's: Check whether at one or more bit positions the clause is touched but not satisfied. Then unassign all variables on those bit positions. Parallelizing the algorithm has two main advantages: First, since it is easy to perform the detection in parallel, the costs are relatively small. Second, if an autarky is found on a single bit position, clauses can be removed from the formula which will reduce the the propagation costs of the entire solving procedure. Therefore, detecting autarkies and removing clauses in parallel, could (at least in theory) result in a super linear speed-up.

Example 7.8

To explain the multi-bit autarky detection, we start by using a slightly modified example formula from the multi-bit unit propagation example and the same initial φ_{master}. In this example $\circledast$ denotes a bit position that has recently been unassigned.

$$\begin{aligned}
\mathcal{F}'_{\text{example}} &:= (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \\
& (\neg x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_4) \\
\varphi_{\text{master}} &:= \{x_1 = \mathtt{0110}, x_2 = \mathtt{1100}, x_3 = \mathtt{1010}, x_4 = \mathtt{0110}\}
\end{aligned}$$

First, we loop once through all the clauses. If a clause is not satisfied on a certain bit position all variables in that clause are unassigned at that bit position:

$$\begin{array}{rcl}
(x_1 = \mathtt{0110} \vee x_2 = \mathtt{1100}) & \Rightarrow & x_1 := \mathtt{011}\circledast, x_2 := \mathtt{110}\circledast \\
(\neg x_1 = \mathtt{100}{*} \vee \neg x_2 = \mathtt{001}{*} \vee x_3 = \mathtt{1010}) & \Rightarrow & x_1 := \mathtt{0}\circledast\mathtt{1}{*}, x_2 := \mathtt{1}\circledast\mathtt{0}{*}, \\
 & & x_3 := \mathtt{1}\circledast\mathtt{1}\circledast \\
(\neg x_2 = \mathtt{0}{*}\mathtt{1}{*} \vee \neg x_3 = \mathtt{0}{*}\mathtt{0}{*}) & \Rightarrow & x_2 := \circledast{*}\mathtt{0}{*}, x_3 := \circledast{*}\mathtt{1}{*} \\
(\neg x_2 = {*}{*}\mathtt{1}{*} \vee x_3 = {*}{*}\mathtt{1}{*} \vee \neg x_4 = \mathtt{1001}) & \Rightarrow & x_4 := \mathtt{0}\circledast\mathtt{10} \\
(\neg x_2 = {*}{*}\mathtt{1}{*} \vee x_3 = {*}{*}\mathtt{1}{*} \vee x_4 = \mathtt{0}{*}\mathtt{10}) & \Rightarrow & x_4 := \circledast{*}\mathtt{1}\circledast \\
(\neg x_3 = {*}{*}\mathtt{0}{*} \vee \neg x_4 = {*}{*}\mathtt{0}{*}) & \Rightarrow & x_3 := {*}{*}\circledast{*}, x_4 := {*}{*}\circledast{*}
\end{array}$$

Second, we loop again through the clauses. This will unassign one more bit position:

$$(x_1 = \mathtt{0}{*}\mathtt{1}{*} \vee x_2 = {*}{*}\mathtt{0}{*}) \quad \Rightarrow \quad x_1 := \circledast{*}\mathtt{1}{*}$$

Since no assignments are unassigned by the other clauses, the algorithm stops. The presence of assigned variables x_1 and x_2 on bit position 3 indicate that we found an autarky. This autarky satisfies all clauses except $\neg x_3 \vee \neg x_4$. Since the remaining clause is satisfiability equivalent to $\mathcal{F}'_{\text{example}}$, the satisfied clauses can be removed from the formula and we can continue solving only the reduced formula. The example shows that detection of autarkies can reduce the formula considerably and speed-up the solving time.

Detection of autarkies can be implemented more efficiently compared to the description above: Only in the first iteration, one needs to loop through all the clauses. In succeeding iterations, only those clauses that contain a variable that was unassigned (at some bit position) in the prior iteration need to be examined. Another technique to reduce the computational costs of the detection algorithm is to call it once every k periods. In case an autarky exists on some bit position(s), the UnitWalk algorithm will not alter the truth values on those bit positions of the variables contributing to the autarky. Therefore, calling the detection algorithm every once in a while will reveal the same autarkies - although slightly later.

7.6 Results

We implemented the UNITWALK algorithm as a multi-bit local search solver using UNITPROPAGATION_QUEUE. The resulting solver, called UnitMarch, can be used for any number of bits. We added the method which detects and replaces duplicates with new random assignments (see Section 7.5.1). Because the autarky detection feature (see Section 7.5.2) only slightly influences the performance on the selected benchmarks, we decided to present the results from [HvM07]. The performance of UnitMarch is compared with the latest version of UnitWalk[25.].

The latter is a hybrid solver: If after a number of periods the number of unsatisfied clauses is not reduced the solver switches to WALKSAT [SKC94]. In turn, if that algorithm does not find a solution after a multitude of flips it switches back, etc. Since we wanted to compare the influence of multi-bit search on the pure UNITWALK algorithm, this switching was disabled.

Table 7.2 shows a comparison between UnitWalk, UnitMarch 1-bit and UnitMarch 32-bit on various benchmarks. Apart from the *dlx2-bugXX* family[26.], all benchmarks can be found on SATlib[27.] along with a description. For each solver, we set `MAX_PERIODS` $:= \infty$. We used 100 random seeds for all benchmarks.

The solvers UnitWalk and UnitMarch 1-bit show comparable performance. First, the number of periods executed per second is almost the same for all checked benchmarks. This shows that our implementation, with some overhead for parallelization, is fast enough on the benchmarks at hand. Second, the average number of periods between the two versions is comparable. Although they differ slightly between instances, the results are "too close to call": There is no clear winner. Hence, the UNITPROPAGATION_QUEUE procedure shows comparable to the UNITPROPAGATION_MULTISET procedure in terms of performance.

Comparing both 1-bit solvers to UnitMarch 32-bit shows that the latter is the clear winner on almost all experimented instances. We found few exceptions (see *logistics-d*); all having less than 100 periods on the three solvers. Apparently, multi-bit search as implemented is not effective on these simple instances. Figures 7.1 and 7.2 present the effect of using different numbers of bits in more detail. Both figures use logarithmic axes - thus $f(x) = \frac{c}{x}$ is represented as a straight line. Four benchmarks are tested for all bits sizes 1 to 32. Using double logarithmic scaling, these instances show a linear dependency between the average number of periods and the number of used bits. The average time is also diminished on all these instances, although this reduction varies per instance. Notice that on all these instances the trend is strictly decreasing. On instances such as the parity benchmarks, it could be expected that computers with a p-bit architecture with $p > 32$ will boost performance even further.

[25.] version 1.003 available from `http://logic.pdmi.ras.ru/~arist/UnitWalk/`
[26.] available from `http://www.miroslav-velev.com/sat_benchmarks.html`
[27.] `http://www.satlib.org`

Table 7.2 — Comparison between the performance - in average number of periods and average time and standard deviation - of UnitWalk, UnitMarch 1-bit, and UnitMarch 32-bit on various benchmarks. The presented data averages runs using 100 different random seeds.

	UnitWalk 1.003		UnitMarch 1-bit		UnitMarch 32-bit	
	periods	**time**	**periods**	**time**	**periods**	**time**
aim-2-1-1	119336	6.13 $^{(6.36)}$	37520	1.62 $^{(1.65)}$	1339	**0.32** $^{(0.33)}$
aim-2-1-2	1395975	73.56 $^{(71.97)}$	1001609	44.67 $^{(43.37)}$	45934	**11.35** $^{(10.68)}$
aim-2-1-3	26487	1.40 $^{(1.39)}$	12147	0.53 $^{(0.60)}$	646	**0.16** $^{(0.15)}$
aim-2-1-4	57794	3.13 $^{(3.01)}$	30708	1.38 $^{(1.58)}$	945	**0.23** $^{(0.22)}$
aim-3-4-1	89923	7.57 $^{(7.05)}$	62191	3.19 $^{(3.07)}$	2134	**1.40** $^{(1.42)}$
aim-3-4-2	99744	8.43 $^{(7.98)}$	181623	9.33 $^{(8.51)}$	5838	**3.81** $^{(3.33)}$
aim-3-4-3	51898	4.33 $^{(4.07)}$	20870	1.7 $^{(0.90)}$	738	**0.48** $^{(0.45)}$
aim-3-4-4	264125	21.96 $^{(17.79)}$	240856	21.21 $^{(13.43)}$	6234	**4.29** $^{(3.15)}$
bw-large.b	441	0.32 $^{(0.33)}$	311	0.18 $^{(0.13)}$	13	**0.05** $^{(0.03)}$
bw-large.c	13870	47.61 $^{(40.90)}$	9342	19.85 $^{(22.05)}$	498	**7.63** $^{(7.44)}$
dlx2-bug17	1102	6.40 $^{(9.53)}$	432	2.31 $^{(2.80)}$	7	**0.43** $^{(0.41)}$
dlx2-bug39	2830	6.78 $^{(6.13)}$	1899	4.38 $^{(3.72)}$	69	**1.33** $^{(1.76)}$
dlx2-bug40	1632	3.96 $^{(4.02)}$	988	2.34 $^{(2.20)}$	26	**0.55** $^{(0.55)}$
flat200-05	19384	3.46 $^{(3.40)}$	19880	2.19 $^{(2.35)}$	704	**0.81** $^{(0.75)}$
flat200-24	5247	0.98 $^{(1.02)}$	5145	0.56 $^{(0.56)}$	130	**0.16** $^{(0.18)}$
flat200-39	12142	2.16 $^{(2.29)}$	12048	1.31 $^{(1.21)}$	391	**0.44** $^{(0.45)}$
flat200-48	2941	0.52 $^{(0.54)}$	2346	0.26 $^{(0.25)}$	84	**0.10** $^{(0.10)}$
flat200-64	6406	1.14 $^{(1.03)}$	6799	0.75 $^{(0.75)}$	268	**0.34** $^{(0.35)}$
logistics-a	1970338	636.47 $^{(563.21)}$	863165	369.09 $^{(383.97)}$	25100	**55.97** $^{(43.53)}$
logistics-b	6313	1.91 $^{(2.24)}$	11878	5.43 $^{(5.76)}$	354	**0.73** $^{(0.63)}$
logistics-c	133572	72.16 $^{(69.36)}$	310450	228.49 $^{(224.92)}$	9803	**34.19** $^{(31.75)}$
logistics-d	23	0.11 $^{(0.07)}$	24	**0.08** $^{(0.04)}$	5	0.11 $^{(0.03)}$
par16-1	14245	4.97 $^{(4.73)}$	11267	2.65 $^{(2.85)}$	365	**0.21** $^{(0.20)}$
par16-2	21417	7.43 $^{(8.08)}$	20601	5.05 $^{(5.18)}$	702	**0.42** $^{(0.34)}$
par16-3	17913	6.31 $^{(7.04)}$	16872	3.98 $^{(3.93)}$	551	**0.33** $^{(0.42)}$
par16-4	16955	5.94 $^{(5.77)}$	14087	3.33 $^{(3.47)}$	523	**0.34** $^{(0.32)}$
par16-5	18889	6.60 $^{(6.70)}$	23028	5.41 $^{(5.00)}$	640	**0.36** $^{(0.36)}$
qg1-08	101390	424.17 $^{(399.59)}$	121127	362.74 $^{(377.55)}$	4229	**127.57** $^{(120.87)}$
qg2-08	803258	3404.49 $^{(3501.46)}$	1005351	4360.92 $^{(4518.23)}$	26223	**991.23** $^{(967.20)}$
qg3-08	165	0.08 $^{(0.06)}$	166	0.10 $^{(0.10)}$	5	**0.03** $^{(0.03)}$
qg4-09	1344	1.10 $^{(0.96)}$	2098	1.82 $^{(1.66)}$	66	**0.53** $^{(0.53)}$
qg5-11	591	1.92 $^{(1.82)}$	670	2.13 $^{(2.00)}$	23	**0.82** $^{(0.68)}$
qg7-13	92600	492.66 $^{(465.71)}$	98172	408.35 $^{(419.56)}$	2937	**171.63** $^{(146.69)}$
uf250-054	307317	33.69 $^{(35.84)}$	472970	30.03 $^{(27.82)}$	14851	**10.74** $^{(11.57)}$
uf250-062	42137	4.60 $^{(4.85)}$	88670	5.61 $^{(5.44)}$	2427	**1.74** $^{(1.84)}$
uf250-071	135296	14.49 $^{(12.79)}$	218375	13.92 $^{(13.70)}$	6404	**4.59** $^{(4.66)}$
uf250-072	126387	13.91 $^{(13.33)}$	172789	10.95 $^{(9.81)}$	5624	**4.10** $^{(4.28)}$
uf250-093	92110	9.78 $^{(9.71)}$	146132	9.23 $^{(8.37)}$	4521	**3.25** $^{(2.94)}$

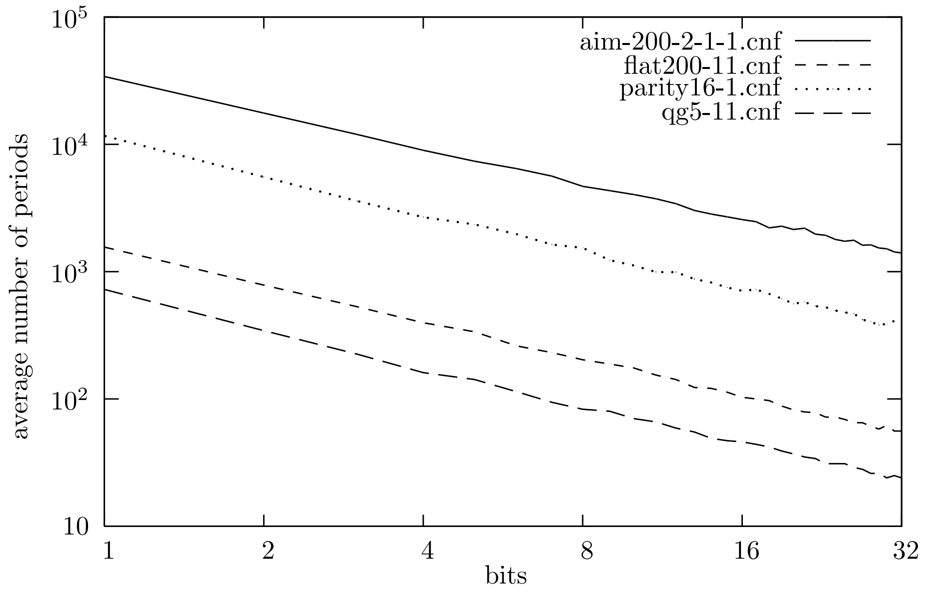

Figure 7.1 — Average number of periods by UnitMarch using different number of bits - computed using 1000 random seeds. Both axes are logarithmic.

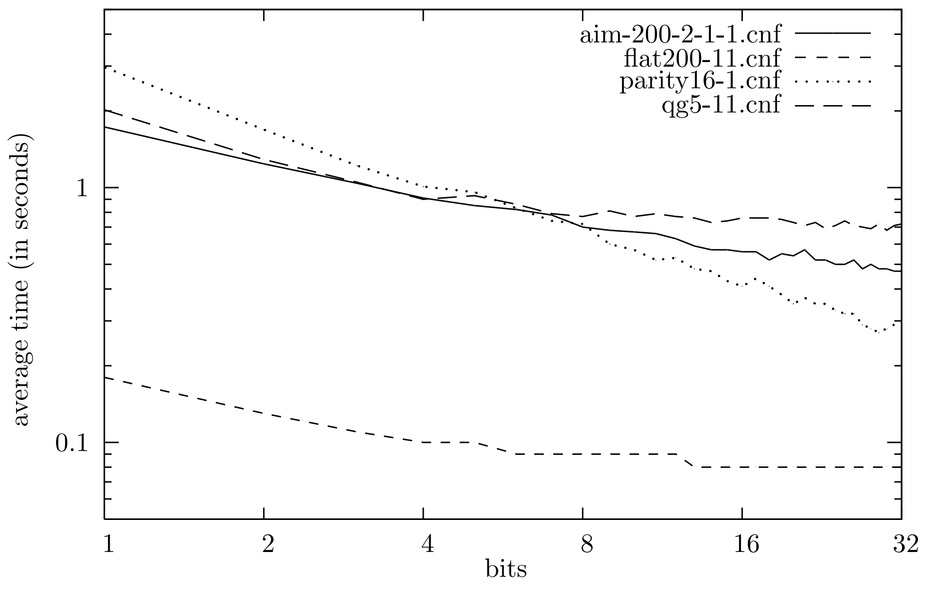

Figure 7.2 — Average time (in seconds) by UnitMarch using different number of bits - computed using 1000 random seeds. Both axes are logarithmic.

Although the detection of autarkies sporadically influenced the results on the selected benchmarks, we present the usefulness of this technique using a separate experiment: We concatenated multiple satisfiable random 3-Sat formulae[28.] such that each formula uses different variables. Each concatenated formula consists of multiple components and for each component there exists an autarky satisfying only the component. Experiments on similar formulae is discussed in [BS06].

The performance of UnitMarch on these formulae - with and without the autarky feature - is shown in Figure 7.3. The version with autarky detection is orders of magnitude faster. Also, the larger the number of components, the larger the speed-up factor realized by the technique. So, if formulae consist of independent components, they can be solved much faster using detection of autarkies. Practical applications for this technique are under current research.

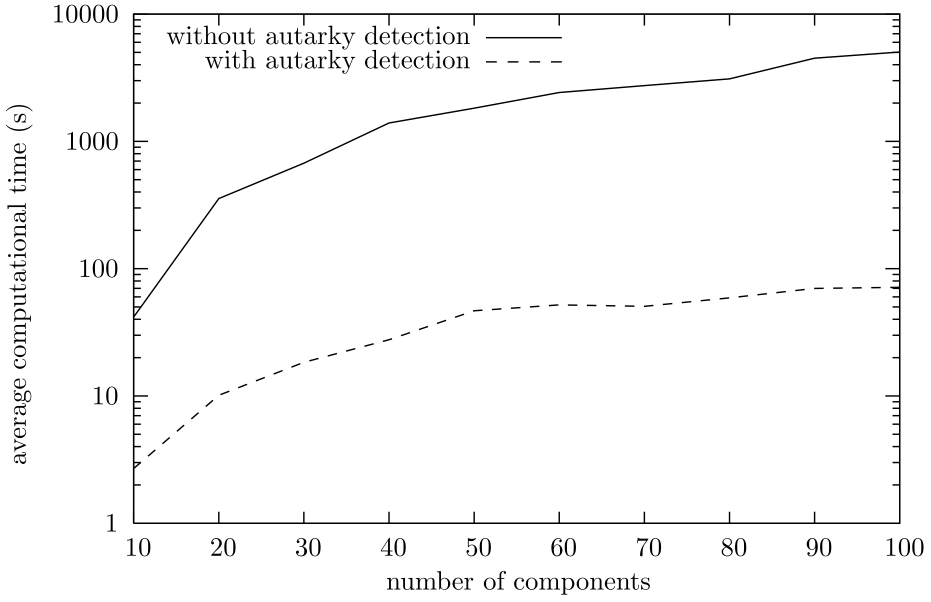

Figure 7.3 — Performance of UnitMarch 32-bit with and without autarky detection on concatenated formulae of random 3-Sat instances.

[28.] with 200 variables and 860 clauses, also from `http://www.satlib.org`

7.7 Conclusions and future work

Our first observation is that the probability of hitting a solution of propositional Boolean formulae is increased by using bigger Boolean Algebras. Also, Boolean formulae with n variables can be checked on feasibility with a single (generic) multi-bit Boolean assignment using the 2^n bit Boolean Algebra. Compared to conventional checking algorithms, the above just exchanges time for space. However, the architecture of today's computers is 32- or 64-bit - which enables execution of 32 (or 64) 1-bit operations simultaneously. Although many algorithms do not seem suitable for this kind of parallelism, the UnitWalk algorithm appears to be a suitable first candidate, as well as a state-of-the-art Sat solver [LS03].

Our multi-bit implementation of this algorithm, called UnitMarch, shows that this algorithm can be parallelized in such a way that the 1-bit version shows comparable performance to the UnitWalk solver. Using double logarithmic scaling, these instances show a linear dependency between the average number of periods and the number of used bits. Most importantly, the average time to solve instances is reduced by using the 32-bit version.

The implementations of UnitWalk and UnitMarch are currently comparable (regardless the multi-bit feature) but are far from optimal: For instance, in both solvers unit clauses in the original CNF are propagated in each period. Another performance boost is expected by adding (redundant) clauses - for instance as implemented in the local search solver R^+AdaptNovelty$^+$ [APSS05] - because they will increase the number of unit propagations. Finally, further experiments (not presented in this chapter) showed that ordering the variables less randomly and more based on multi-bit heuristics results in improved performance on many benchmarks. Developing enhancements (like replacement of duplicate assignments and detection of autarkies) and effective multi-bit heuristics is under current research.

Take away paradox from the thinker
and you have a professor.
Soren Kierkegaard

8 Conclusions

We concluded our analysis regarding state-of-the-art SAT solving techniques (see Section 2.5) with five topics of interesting future research areas: Enhancing the LOOKAHEAD architecture, enhancing the UNITWALK architecture, adaptive heuristics, direction heuristics, and representation. In this chapter, we will summarize our contributions in these areas and offer possible extensions or continuations of the work presented here.

8.1 A relic with a future

About 90% of the complete SAT solvers that participated during the recent SAT competitions were based on the conflict-driven architecture. Look-ahead SAT solvers are far less popular. They are considered only strong on unsatisfiable random k-SAT formulae. However, we believe that look-ahead SAT solvers can be competitive on a wide variety of benchmarks.

Many contributions to the look-ahead SAT solver architecture have been presented in this thesis. First, we reduced the computational costs of several existing look-ahead SAT solving techniques and added equivalence reasoning in such a way that it is only applied when useful (Chapter 3). Second, to improve the performance of march on structured instances, we enhanced the preprocessor, added adaptive heuristics and developed a new branching strategy (Chapter 4). Also, we improved our own adaptive heuristics (Chapter 5) and studied the influence of direction heuristics. Then we capitalized on the observed results (Chapter 6).

These contributions boosted the performance of our look-ahead SAT solver march. The different versions of this solver won various awards during the SAT competitions. Amongst these are the first prizes in the divisions: Crafted ALL (SAT 2004), crafted UNSAT (SAT 2004), crafted SAT (SAT 2007), and random UNSAT (SAT 2007). These awards prove that march is very competitive on both crafted and random formulae.

Yet conflict-driven SAT solvers are still superior on industrial benchmarks. Quite some additional progress is required to close this gap. Still, we expect it can be done by adding more reasoning techniques - such as adding conflict clauses. Fast performance on industrial benchmarks would really prove that the LookAhead architecture could function as a foundation for general purpose Sat solvers.

Another challenge for look-ahead Sat solvers is the ability to solve large hard random k-Sat formulae. Although the presented techniques have boosted the performance on structured benchmarks, only small gains were established on random formulae. For instance, unsatisfiable random 3-Sat formulae near the phase transition density with 1000 variables cannot be solved within years of computational time. Look-ahead Sat solvers perform best on the smaller sized instances and therefore are likely the most fruitful candidates to solve larger formulae in the future.

8.2 Marching on?

Techniques such as unit propagation and autarky detection appeared to be suitable for parallelization using multi-bit Booleans. Although performance improved by implementing the parallelization in our incomplete Sat solver UnitMarch, it was not enough to make the solver very competitive. The question arises: Can we capitalize on these results for future research?

A first direction is to further enhance UnitMarch by adding new techniques and possibly try to parallelize these as well. Currently, we experiment with the use of a less random order of the variables as input for each period. Early results show that the speed improves significantly on various instances, although performance losses have been observed too.

A second possible continuation of this research is to parallelize other techniques in a similar manner outside the scope of UnitMarch. A possible alternative is studying the possibilities to parallelize look-aheads on a single processor. At first sight, look-aheads seem an interesting candidate: 1) Most of the computational time is spent performing unit propagation, 2) look-aheads are independent of each other, and 3) they all start from the same formula. However, look-aheads also perform some counting, which is difficult to parallelize.

8.3 Adaptation, adaptation, adaptation

Not only does there exist a wide variety of Sat formulae, the reduced formulae that arise while solving a certain problem appear to be quite different too. Therefore, solving strategies should adapt towards these differences as well. Chapter 5 presented such an adaptive algorithm to guide the DoubleLook procedure. Due to this adaptive technique, this procedure reduces the solving time on a vast majority of benchmarks.

Besides the reduction of the overall computational costs, adaptive heuristics have several other advantages. For instance, users of Sat solvers do not require knowledge of optimal parameter settings for their specific problem. Also, one does not need to update parameter settings after adding a new feature, which is often the case using magic (static) constants.

Apart from the adaptive algorithm for the DoubleLook procedure, we developed an adaptive algorithm to determine the optimal number of look-ahead variables (see Section 4.4.1). In general, this algorithm does improve the solving speed, but some performance losses were also observed. It appeared hard to construct an adaptive algorithm which is elegant (uses few magic constants) and sets the parameters to near optimal values.

Ultimately, one desires an adaptive algorithm that not only effectively modifies the parameters of the various heuristics. Ideally, it even selects the optimal (Sat solving) architecture for each (reduced) formula. For instance, the original formula might be best reduced using a conflict driven solver, while the remaining formula is best solved using the WalkSat architecture.

Concluding, there is a high potential for adaptive algorithms and therefore these are an interesting topic of Sat research. Yet, quite some progress is still required to make Sat solvers really adaptive.

8.4 Left or right, that is the question

Direction heuristics, although very powerful in theory, are not a well-studied topic within the field of Sat solving. Chapter 6 provides a first study on how direction heuristics in look-ahead Sat solvers influence the distribution of solutions on random k-Sat formulae. All studied Sat solvers showed a similar bias on these formulae, but some biased the distribution of solutions more than others. We currently study the bias of direction heuristics in look-ahead Sat solvers on structured instances.

Capitalizing on the observed bias is a logical next step. We developed a new jump strategy that visits subtrees in decreasing probability (based on the observations) of containing a solution. This jump strategy called distribution jumping appeared successful on the studied random k-Sat formulae as well as on many structured benchmarks. Yet the usefulness of distribution jumping can likely be improved: First, by developing direction heuristics that bias the distribution of solutions even more: The larger the bias, the larger the expected gain. Second, the generalized jump order π_{left} is probably not the optimal one. Constructing a generalized jump order with more similarities to the greedy jump orders (see Section 6.4.1) could further improve performance.

Last but not least, the effect of direction heuristics in conflict-driven Sat solvers needs to be studied. Due to the addition of conflict clauses, direction heuristics influence performance on unsatisfiable formulae as well. This complicates the choice for the left of the right branch even more. Current strong conflict-driven Sat solvers either *branch negatively* (always assign decision variables to false) [ES03] or perform *progress saving* (assign decision variables to their last forced value) [PD07]. Both techniques are very simple (in terms of computational costs). More sophisticated direction heuristics that focus on early detection of conflicts may improve conflict-driven Sat solvers.

8.5 Re-representation

SAT solving has become a quite powerful method to effectively solve a wide variety of problems. Transforming these problems into SAT can be realized in various ways. Not much is known about which transformation results in an ideal representation for a certain SAT solver. This thesis provides only some minor contributions to this research area. Amongst these are the pre-processing techniques in Section 4.3. Additional progress of SAT solvers is expected if we would obtain more knowledge about an effective representation: A pre-processor can use this knowledge to translate a transformed problem into such a representation.

Furthermore, we should study whether pure CNF is the ideal representation. Hybrid representations consisting of both clauses and seperate higher level constraints (such as cardinality constraints) may be very useful to SAT solvers in the future. An advantage of a hybrid representation is that new forms of (specialized) reasoning can be added. On the other hand, existing techniques such as conflict analysis and look-ahead may become more complex and therefore more expensive. In Section 3.6, we showed how equivalence clauses can be extracted from the CNF to create a hybrid representation. The equivalence reasoning as implemented in march improves performance on several benchmarks, without hurting it on others. Similar enhancements could be added with other higher level constraints or to other SAT solvers.

Bibliography

[AM04] Carlos Ansótegui and Felip Manyà, *Mapping problems with finite-domain variables to problems with Boolean variables.*, In Hoos and Mitchell [HM05], pp. 1–15.

[APSS05] Anbulagan, Duc Nghia Pham, John K. Slaney, and Abdul Sattar, *Old resolution meets modern SLS.*, AAAI (Manuela M. Veloso and Subbarao Kambhampati, eds.), AAAI Press / The MIT Press, 2005, pp. 354–359.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu, *Symbolic model checking without BDDs*, TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (London, UK), Springer-Verlag, 1999, pp. 193–207.

[BIK$^+$92] Paul Beame, Russell Impagliazzo, Jan Krajíček, Toniann Pitassi, Pavel Pudlák, and Alan Woods, *Exponential lower bounds for the pigeonhole principle*, STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing (New York, NY, USA), ACM, 1992, pp. 200–220.

[BMZ05] Alfredo Braunstein, Marc Mézard, and Riccardo Zecchina, *Survey propagation: An algorithm for satisfiability*, Random Struct. Algorithms **27** (2005), no. 2, 201–226.

[Bro90] Frank Markham Brown, *Boolean reasoning: The logic of Boolean equations*, Kluwer Academic Publishers, Dordrecht, 1990.

[BS92] Alain Billionnet and Alain Sutter, *An efficient algorithm for the 3 satisfiability problem*, Operation Research Letters **12** (1992), 29–36.

[BS96] Max Böhm and Ewald Speckenmeyer, *A fast parallel SAT-solver - efficient workload balancing*, Annals of Mathematics and Artificial Intelligence **17** (1996), no. 159, 381–400.

[BS06] Armin Biere and Carsten Sinz, *Decomposing SAT problems into connected components*, Journal on Satisfiability, Boolean Modeling and Computation **2** (2006), 191–198.

[BSK03] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin, *Parallel propositional satisfiability checking with distributed dynamic learning*, Parallel Comput. **29** (2003), no. 7, 969–994.

[BW05] Fahiem Bacchus and Toby Walsh (eds.), *Theory and applications of satisfiability testing, 8th international conference, SAT 2005, St. Andrews, UK, june 19-23, 2005, proceedings*, Lecture Notes in Computer Science, vol. 3569, Springer, 2005.

[Con04] Harold S. Connamacher, *A random constraint satisfaction problem that seems hard for DPLL*, In Hoos and Mitchell [HM05].

[Coo71] Stephen A. Cook, *The complexity of theorem-proving procedures*, STOC '71: Proceedings of the third annual ACM symposium on Theory of computing (New York, NY, USA), ACM, 1971, pp. 151–158.

[DD] Gilles Dequen and Olivier Dubois, *Source code of the kcnfs solver*, Available at `http://www.laria.u-picardie.fr/~dequen/sat/`.

[DD01] Olivier Dubois and Gilles Dequen, *A backbone-search heuristic for efficient solving of hard 3-SAT formulae.*, IJCAI (Bernhard Nebel, ed.), Morgan Kaufmann, 2001, pp. 248–253.

[DD03] Gilles Dequen and Olivier Dubois, *kcnfs: An efficient solver for random k-SAT formulae*, In Giunchiglia and Tacchella [GT04], pp. 486–501.

[DLL62] Martin Davis, George Logemann, and Donald Loveland, *A machine program for theorem-proving*, Commun. ACM **5** (1962), no. 7, 394–397.

[DP60] Martin Davis and Hilary Putnam, *A computing procedure for quantification theory*, Journal of the ACM **7** (1960), no. 3, 201–215.

[EB05] Niklas Eén and Armin Biere, *Effective preprocessing in SAT through variable and clause elimination.*, In Bacchus and Walsh [BW05], pp. 61–75.

[ES03] Niklas Eén and Niklas Sörensson, *An extensible SAT-solver*, In Giunchiglia and Tacchella [GT04], pp. 502–518.

[ES05] Niklas Eén and Niklas Sörensson, *Minisat – a SAT solver with conflict-clause minimization.*, 2005, Solver description for SAT 2005.

[ES06] Niklas Eén and Niklas Sörensson, *Translating pseudo-Boolean constraints into SAT*, Journal on Satisfiability, Boolean Modeling and Computation **2** (2006), 1–25.

[Fre95] Jon William Freeman, *Improvements to propositional satisfiability search algorithms.*, Ph.D. thesis, University of Pennsylvania, Philadelphia, PA, USA, 1995.

[GSC97] Carla P. Gomes, Bart Selman, and Nuno Crato, *Heavy-tailed distributions in combinatorial search.*, In Smolka [Smo97], pp. 121–135.

[GT04] Enrico Giunchiglia and Armando Tacchella (eds.), *Theory and applications of satisfiability testing, 6th international conference, SAT 2003. Santa Margherita Ligure, Italy, may 5-8, 2003 selected revised papers*, Lecture Notes in Computer Science, vol. 2919, Springer, 2004.

[HDvZvM04] Marijn J.H. Heule, Mark Dufour, Joris E. van Zwieten, and Hans van Maaren, *March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver*, In Hoos and Mitchell [HM05], pp. 345–359.

[Her06] Paul Herwig, *Decomposing satisfiability problems.*, Master's thesis, TU Delft, 2006.

[HK05] Edward A. Hirsch and Arist Kojevnikov, *UnitWalk: A new SAT solver that uses local search guided by unit clause elimination*, Annals of Mathematics and Artificial Intelligence **43** (2005), no. 1-4, 91–111.

[HM05] Holger H. Hoos and David G. Mitchell (eds.), *Theory and applications of satisfiability testing, 7th international conference, SAT 2004, Vancouver, BC, Canada, may 10-13, 2004, revised selected papers*, Lecture Notes in Computer Science, vol. 3542, Springer, 2005.

[Hoo02] Holger H. Hoos, *An adaptive noise mechanism for WalkSAT*, Eighteenth national conference on Artificial intelligence (Menlo Park, CA, USA), American Association for Artificial Intelligence, 2002, pp. 655–660.

[HvM04] Marijn J.H. Heule and Hans van Maaren, *Aligning CNF- and equivalence-reasoning*, In Hoos and Mitchell [HM05], pp. 145–156.

[HvM06] Marijn J.H. Heule and Hans van Maaren, *March_dl: Adding adaptive heuristics and a new branching strategy*, Journal on Satisfiability, Boolean Modeling and Computation **2** (2006), 47–59.

[HvM07] Marijn J.H. Heule and Hans van Maaren, *From idempotent generalized boolean assignments to multi-bit search*, In Marques-Silva and Sakallah [MSS07], pp. 134–147.

[IKM$^+$02] Kazuo Iwama, Daisuke Kawai, Shuichi Miyazaki, Yasuo Okabe, and Jun Umemoto, *Parallelizing local search for CNF satisfiability using vectorization and PVM.*, ACM Journal of Experimental Algorithms **7** (2002), 2.

[KHR$^+$02] Henry Kautz, Eric Horvitz, Yongshao Ruan, Carla Gomes, and Bart Selman, *Dynamic restart policies*, Eighteenth national conference on Artificial intelligence (Menlo Park, CA, USA), American Association for Artificial Intelligence, 2002, pp. 674–681.

[KKM96] Florian Krohm, Andreas Kuehlmann, and Arjen Mets, *The use of random simulation in formal verification*, ICCD, IEEE Computer Society, 1996, pp. 371–376.

[KMT07] Oliver Kullmann, Victor W. Marek, and Miroslaw Truszczyński, *Computing autarkies and properties of the autarky monoid*, 2007, In preparation.

[Kul99] Oliver Kullmann, *On a generalization of extended resolution*, Discrete Applied Mathematics **96-97** (1999), no. 1, 149–176.

[Kul00] Oliver Kullmann, *Investigations on autark assignments*, Discrete Applied Mathematics **107** (2000), no. 1-3, 99–137.

[Kul02] Oliver Kullmann, *Investigating the behaviour of a SAT solver on random formulas*, Tech. Report CSR 23-2002, University of Wales Swansea, Computer Science Report Series (`http://www-compsci.swan.ac.uk/reports/2002.html`), October 2002, 119 pages.

[Kul08] Oliver Kullmann, *A survey on practical SAT algorithms*, Complexity of Constraints (Nadia Creignou, Phokion Kolaitis, and Heribert Vollmer, eds.), Springer, 2008.

[LA97a] Chu Min Li and Anbulagan, *Heuristics based on unit propagation for satisfiability problems.*, IJCAI (1), 1997, pp. 366–371.

[LA97b] Chu Min Li and Anbulagan, *Look-ahead versus look-back for satisfiability problems.*, In Smolka [Smo97], pp. 341–355.

[LeB01] Daniel LeBerre, *Exploiting the real power of unit propagation lookahead*, Proceedings of SAT2001: Workshop on Theory and Application of Satisfiability Testing, vol. 9, Elsevier, 2001.

[LH05] Chu Min Li and Wen Qi Huang, *Diversification and determinism in local search for satisfiability*, In Bacchus and Walsh [BW05], pp. 158–172.

[Li99] Chu Min Li, *A constraint-based approach to narrow search trees for satisfiability*, Information processing letters **71** (1999), no. 2, 75–80.

[Li03] Chu-Min Li, *Equivalent literal propagation in the DLL procedure*, Discrete Applied Mathematics **130** (2003), no. 2, 251–276.

[LS03] Daniel LeBerre and Laurent Simon, *The essentials of the SAT 2003 competition*, In Giunchiglia and Tacchella [GT04], pp. 452–467.

[LS04] Daniel LeBerre and Laurent Simon, *Fifty-five solvers in Vancouver: The SAT 2004 competition*, In Hoos and Mitchell [HM05], pp. 321–344.

[LS06] Daniel LeBerre and Laurent Simon, *Preface special volume on the SAT 2005 competitions and evaluations*, 2006, Journal on Satisfiability, Boolean Modeling and Computation **2**.

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik, *Chaff: Engineering an efficient SAT solver.*, DAC, ACM, 2001, pp. 530–535.

[MSK97] David McAllester, Bart Selman, and Henry Kautz, *Evidence for invariants in local search*, Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97) (Providence, Rhode Island), 1997, pp. 321–326.

[MSL92] David G. Mitchell, Bart Selman, and Hector J. Levesque, *Hard and easy distributions for SAT problems*, Proceedings of the Tenth National Conference on Artificial Intelligence (Menlo Park, California) (Paul Rosenbloom and Peter Szolovits, eds.), AAAI Press, 1992, pp. 459–465.

[MSS96] Joao P. Marques-Silva and Karem A. Sakallah, *GRASP – a new search algorithm for satisfiability*, ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design (Washington, DC, USA), IEEE Computer Society, 1996, pp. 220–227.

[MSS07] João Marques-Silva and Karem A. Sakallah (eds.), *Theory and applications of satisfiability testing - SAT 2007, 10th international conference, Lisbon, Portugal, may 28-31, 2007, proceedings*, Lecture Notes in Computer Science, vol. 4501, Springer, 2007.

[MvVW07] Dimos Mpekas, Michiel van Vlaardingen, and Siert Wieringa, *The first steps to a hybrid SAT solver*, 2007, MSc report, SAT@Delft.

[PD07] Knot Pipatsrisawat and Adnan Darwiche, *A lightweight component caching scheme for satisfiability solvers*, In Marques-Silva and Sakallah [MSS07], pp. 294–299.

[PK01] Donald J. Patterson and Henry Kautz, *Auto-WalkSAT: A self-tuning implementation of walksat*, Proceedings of SAT2001: Workshop on Theory and Application of Satisfiability Testing, vol. 9, Elsevier, 2001.

[Pre07] Steven David Prestwich, *Variable dependency in local search: Prevention is better than cure*, In Marques-Silva and Sakallah [MSS07], pp. 107–120.

[SKC94] Bart Selman, Henry A. Kautz, and Bram Cohen, *Noise strategies for improving local search*, AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1) (Menlo Park, CA, USA), American Association for Artificial Intelligence, 1994, pp. 337–343.

[SLH05] Laurent Simon, Daniel LeBerre, and Edward A. Hirsch, *The SAT2002 competition*, Annals of Mathematics and Artificial Intelligence **43** (2005), no. 1, 307–342.

[SLM92] Bart Selman, Hector J. Levesque, and D. Mitchell, *A new method for solving hard satisfiability problems*, Proceedings of the Tenth National Conference on Artificial Intelligence (Menlo Park, California) (Paul Rosenbloom and Peter Szolovits, eds.), AAAI Press, 1992, pp. 440–446.

[Smo97] Gert Smolka (ed.), *Principles and practice of constraint programming - CP97, third international conference, Linz, Austria, october 29 - november 1, 1997, proceedings*, Lecture Notes in Computer Science, vol. 1330, Springer, 1997.

[VB03] Miroslav N. Velev and Randal E. Bryant, *Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors*, J. Symb. Comput. **35** (2003), no. 2, 73–106.

[vL06] Martijn van Lambalgen, *3MCard: A lookahead cardinality solver*, Master's thesis, TU Delft, 2006.

[WvM98] Joost P. Warners and Hans van Maaren, *A two-phase algorithm for solving a class of hard satisfiability problems*, Operation Research Letters **23** (1998), no. 3-5, 81–88.

[ZBH96] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang, *PSATO: a distributed propositional prover and its application to quasigroup problems*, Journal of Symbolic Computation **21** (1996), no. 4, 543–560.

[Zha06] Hantao Zhang, *A complete random jump strategy with guiding paths*, SAT (Armin Biere and Carla P. Gomes, eds.), Lecture Notes in Computer Science, vol. 4121, Springer, 2006, pp. 96–101.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik, *Efficient conflict driven learning in boolean satisfiability solver.*, ICCAD, 2001, pp. 279–285.

[ZS00] Hantao Zhang and Mark Stickel, *Implementing the Davis-Putnam method*, Journal of Automated Reasoning **24** (2000), no. 1-2, 277–296.

Summary

SmArT solving
Tools and techniques for satisfiability solvers

The satisfiability problem (Sat) lies at the core of the complexity theory. This is a decision problem: Not the solution itself, but whether or not a solution exists – given a specified set of requirements – is the central question. Over the years, the satisfiability problem has taken center stage as a means of effective representation to tackle problems with different characteristics: Many problems can first be translated into Sat and then solved by means of software dedicated to the Sat problem. Due to the increasing power of these Sat solvers, the number of applications climbs every year. Examples of this kind of 'translatable' problems are scheduling problems, verification of software and hardware, bounded model checking and a wide variety of mathematical puzzles.

Sat solvers come in two flavors: Complete and incomplete. Complete solvers systematically go over the whole search space and are able to determine with certainty whether a solution exists. Incomplete Sat solvers look for a solution at a venture. They don't follow a system, yet they might hit a solution.

Most complete Sat solvers are based on the ConflictDriven architecture. At a dead end, they analyze what went wrong and where in the search space it happened. Then they resume the search from there. Conflict-driven solvers make relatively cheap decisions (in terms of computational costs), which enables them to search the space swiftly.

Only a few complete Sat solvers are based on an architecture that chooses its battles, so to speak. The LookAhead architecture peers further into the search space before making the next move. Look-ahead solvers make expensive decisions in order to keep the search space as small as possible.

Incomplete Sat solvers have a dominant and a rare type of architecture as well. The commonly used WalkSat makes cheap decisions, while UnitWalk – a bit off the wall – makes more costly moves.

This thesis deals with a number of contributions to the development of Sat solvers – both complete and incomplete. With the adagium 'poundwise, pennyfoolish' in mind, its focus is primarily set on both rare, more expensive approaches. On one hand, expensive procedures are implemented efficiently to reduce their relative costs, while they maintain their impact on the search space. On the other hand, building up reasoning power further limits the search space.

The growing attention for SAT solvers generates a growing group of users. More and more of them will not be familiar with the specific ins and outs of their application, which makes it difficult to tune the SAT solvers. Here, adaptive heuristics may be of use. Given a specific problem, these heuristics set the parameters in such a way that the solver performs (almost) optimally. This thesis presents some elegant adaptive heuristics for a technique that looks even further ahead into the search area to learn even more: The DOUBLELOOK procedure. Thanks to these adaptive heuristics – that keep on fine-tuning the parameters on the fly – this 'binocular-technique' can be helpful to solve more problems.

The raison d'être of incomplete SAT solvers is the assumption that on many problems, they find a solution faster than complete solvers. Yet they are only superior within a certain niche; they work well on big (random) problems. To extend the span of incomplete SAT solvers, this thesis presents a SAT solver with the more rarely used UNITWALK architecture. Several expensive calculations are run simultaneously – on a single processor – which keeps costs relatively low. In addition, the solver features some communicative skills: Whenever a solution is found for part of the problem, all further calculations will run more efficiently. Despite these enhancements and its good results, this solver is not (yet) competitive on structured problems; an field dominated by complete SAT solvers.

However, our complete SAT solver march, based on the LOOKAHEAD architecture, has won many awards in the prestigious SAT competitions, among which the gold medal for random problems without solutions (2007); a traditional stronghold of LOOKAHEAD solvers. More intriguing is the gold medal for crafted problems with solutions (2007). This is a field which is dominated – besides march – by conflict-driven solvers. In conclusion, the new techniques presented in this thesis have enhanced the LOOKAHEAD architecture to such an extend, that this type of solvers can compete on more problems while sustaining dominance on random instances.

Samenvatting

Goedkoop is duurkoop: effectieve technieken voor het vervulbaarheidsprobleem

Het vervulbaarheidsprobleem (satisfiability, afgekort Sat) ligt aan de basis van de complexiteitstheorie. Dit is een beslissingsprobleem: niet de oplossing zelf, maar de vraag óf er een oplossing bestaat gegeven een bepaald eisenpakket staat centraal. Daarnaast staat het vervulbaarheidsprobleem steeds meer in de belangstelling als een effectieve representatie om problemen van andere aard te lijf te gaan: veel problemen kunnen vertaald worden naar Sat en middels een programma voor dit probleem (Sat solver) worden opgelost. Door de toenemende kracht van Sat solvers groeit het aantal toepassingen ieder jaar. Voorbeelden van dit soort 'vertaalbare' problemen zijn roosteringsvraagstukken, software- en hardware-verificatie, bounded model checking en een verscheidenheid aan wiskundige puzzels.

Sat solvers zijn er in twee smaken: compleet en incompleet. Complete solvers kammen systematisch de hele zoekruimte uit en kunnen daardoor uitsluitsel geven of een probleem oplosbaar is of niet. Incomplete solvers zoeken louter op de bonnefooi naar een oplossing. Ze hanteren geen systematiek, maar kunnen wel op een oplossing stuiten.

De meeste complete Sat solvers zijn gebaseerd op de ConflictDriven architectuur, die bij een doodlopende weg in de zoekruimte analyseert waar het is misgegaan, naar dat punt terugkeert en de zoekprocedure vervolgt. Deze architectuur maakt goedkope beslissingen en kan daardoor snel de zoekruimte afspeuren. Slechts een enkele complete Sat solver maakt gebruik van een architectuur die vooruitblikt (LookAhead). Deze neemt juist dure beslissingen met als doel de zoekruimte zo klein mogelijk te houden.

Ook incomplete Sat solvers kennen een dominante architectuur (WalkSat) die goedkope beslissingen neemt, en een zeldzamere architectuur (UnitWalk) die minder goedkope beslissingen neemt.

Deze thesis behandelt een groot aantal bijdragen aan de ontwikkeling van Sat solvers. Daarbij ligt de focus met name op de voortgang van de beide zeldzamere en duurdere benaderingswijzen met als belangrijkste motivatie: goedkoop is duurkoop. Enerzijds worden kostbare beslissingen efficiënt geïmplementeerd, zodat ze minder duur worden maar hun impact op de zoekruimte behouden. Anderzijds wordt de zoekruimte verder verkleind door de redeneerkracht van de solvers uit te breiden.

De groeiende populariteit van Sat solvers creëert een groeiende groep gebruikers van deze software. Steeds meer gebruikers zullen de ins en outs van de solvers niet kennen. Daardoor is het voor hen niet eenvoudig om hun specifieke probleem met een Sat solver op te lossen. Hier kunnen adaptieve heuristieken uitkomst bieden. Gegeven een probleem stellen deze heuristieken de parameters zodanig in, dat de solver (bijna) optimaal presteert. Deze thesis presenteert een elegante adaptieve heuristiek voor een techniek die ver in de zoekruimte vooruit kijkt om zo extra veel te leren, de DoubleLook procedure. Door de inzet van deze adaptieve heuristiek die ook gaandeweg het zoekproces de parameters blijft bijstellen is deze 'verrekijktechniek' nu op veel meer problemen effectief inzetbaar.

Het bestaansrecht van incomplete Sat solvers ligt in de veronderstelling dat zij op tal van problemen met oplossingen sneller zouden werken. Toch zijn ze alleen superieur in een bepaalde niche, namelijk de grote willekeurige (random) problemen. Om de reikwijdte van incomplete Sat solvers te vergroten, presenteert dit proefschrift een Sat solver met de minder gangbare UnitWalk architectuur. De kosten blijven beperkt door meerdere dure berekeningen tegelijkertijd op een enkele processor uit te voeren. Daarnaast zijn communicatieve vaardigheden toegevoegd: wanneer een oplossing voor een deel van het probleem gevonden is, zullen alle berekeningen efficiënter verlopen. Ondanks de verbeterde prestaties van deze UnitWalk-variant, is de solver (nog) niet competatief op gestructureerde problemen, waar complete Sat solvers de dienst uitmaken.

Daarentegen, de door ons ontwikkelde complete Sat solver march, gebaseerd op de LookAhead architectuur, heeft een groot aantal prijzen gewonnen op de gezaghebbende Sat competities[29.], waaronder de gouden medaille voor random problemen zonder oplossingen (2007), een gebied waar LookAhead solvers traditioneel sterk zijn. Interessanter is de gouden medaille voor crafted problemen met oplossingen (2007). Dit is een categorie die naast march wordt gedomineerd door conflictgedreven solvers. Concluderend, de nieuwe techieken gepresenteerd in dit proefschrift hebben de LookAhead architectuur dusdanig versterkt, dat deze soort solvers op veel meer gebieden kunnen concurreren, zonder op random problemen aan prestatie in te boeten.

[29.] `www.satcompetition.org`

Acknowledgements

After grinding the numbers, punching the code and writing this thesis, I guess it's fair to say that I am kind of an expert on searching. However, those who know me well, will be amused if not amazed by this qualification, to say the least. They know that reality is laughing in the face of theory, for they have not forgotten 'the vacuum cleaner incident'. Some years ago, the day before my mother would return from a trip to Bali, we – my dad, my brother, my sister and me – urgently needed to clean the house, which had suffered severely in her absence. Time was pressing on but we could not find the thing we needed most; the vacuum cleaner. My father promised to look around the couch on which he was sitting, while I frantically searched the rest of the house. My mind was spinning with search strategies. I opened doors, closets, removed desks and checked everywhere systematically, but even the most scrutinizing query could not shine light on my blind spot: House cleaning and the supplies that go with it. After an hour, I gave up. Finally, my sister found it in the doorway to my own room on the top floor, where it had been laying that whole week. I must have stepped over it a hundred times – even while I was looking for it. I guess it's one thing to have an idea, but it is quite another – and it takes more than just me – to put it into gear and deliver the goods.

With this in mind, it is no small wonder that I owe much gratitude to many people who helped me on this project. First and foremost, I thank Hans van Maaren, who triggered my scientific senses from the moment I entered the Satisfiability course. He made and kept me enthusiastic about research ever since. I thank him for the freedom I enjoyed, playing around and thinking out loud while Hans kept a sharp eye on scientific relevance. Plus we had a lot of fun in the process. Second, I thank Cees Witteveen for welcoming me so warmly into his group. Also, I very much appreciate his suggestions for presenting this thesis in a more general context.

Furthermore, I thank Joris van Zwieten and Mark Dufour for teaching me to write software efficiently and for showing what the higher art of programming is all about: Working until four in the morning, skipping showers and debugging with black coffee and yesterday's pizza. Without their help, the first version of march would have never been so successful. I thank Sean Weaver for his involvement with everything we discuss, whose enthusiasm crosses oceans, who can talk without moving his lips. Thanks to Mathijs de Weerdt, my colleague and roommate, for help and comments. Over the last year our conversations stretched far beyond the professional horizon. I thank Denise van der Helm for her thoughts and suggestions and for going to school together all our lives. Also, I would like to thank Denis de Leeuw Duarte for his contributions to the development of UnitMarch, and Stephan van Keulen for his efforts to compute most of the data used in Chapter 6.

I thank my friends and travel companions. Joost Röselaers with whom I share both the spiritual and the profane things in life. Harmen Dikkers for his sharpness, persistent good spirits and good fortune and Dirk van Egmond for his friendship and his help with anything, anytime.

I want to thank my mom, dad, brother and sister for their love and support, and for keeping me with my feet on the ground, whenever my mind is elsewhere. Finally, I want to thank my friend Arjen van Lith for his textual scrutiny, for designing the cover of this thesis and for just being by my side.

Curriculum Vitae

Marienus (Marijn) Johannes Hendrikus Heule was born on March 12, 1979 in Rijnsburg. He solved his first 100-piece puzzle before he could walk. Marijn attended the Rijnlands Lyceum in Oegstgeest from 1991 until 1997, when he obtained his *VWO-diploma*.

Following his life-long fascination with puzzles, he enrolled in the Delft University of Technology in 1997, where he studied Computer Science. For his *ingenieurs* degree (MSc, obtained in 2004), he first studied solvability of two-player board games with perfect information with Dr.drs. L.J.M. Rothkrantz and finished his final assignment on satisfiability solving with Dr. H. van Maaren. Marijn received the award for the best graduate student of the year in Computer Science.

Subsequently, Marijn started working as a PhD student (2004-2008) at the Algorithms group of Delft University of Technology with Dr. H. van Maaren as his supervisor. He studied a wide variety of subjects within the field of satisfiability (SAT), which resulted in this thesis. Meanwhile, he kept his focus on solving itself and wrote software dedicated to the SAT problem. His solver won several awards in various editions of the prestigious SAT competitions.

In addition to his research, Marijn helped set up the Journal on Satisfiability, Boolean Modeling and Computation (JSAT) and now contributes to this journal as a production editor. At Delft University, he gives courses in Satisfiability, supervises several graduate students and advises them on their masters' theses.

Currently Marijn works as a production editor of the Handbook on Satisfiability (IOS Press, eds. Armin Biere, Hans van Maaren, and Toby Walsh). He will continue his research in Delft as a post-doc, this time focusing on cardinality solving.

CV.1 **Awards**

- Best SAT solver on ALL crafted problems by march_eq at SAT'04
- Best SAT solver on SAT crafted problems by march_eq at SAT'04
- Silver medal on ALL random problems by march_dl at SAT'05
- Silver medal on UNSAT random problems by march_dl at SAT'05
- Bronze medal on ALL crafted problems by march_dl at SAT'05
- Silver medal on SAT crafted problems by march_dl at SAT'05
- Bronze medal on UNSAT crafted problems by march_dl at SAT'05
- Silver medal on ALL problems by march_ks at SAT'07
- Gold medal on UNSAT random problems by march_ks at SAT'07
- Gold medal on SAT crafted problems by march_ks at SAT'07

CV.2 Publications

Henriette Bier, Adriaan de Jong, Gijs van der Hoorn, Niels Brouwers, Marijn J.H. Heule and Hans van Maaren. *Prototypes for Automated Architectural 3D-Layout.* VSMM07 Springer LNCS, to appear, 12 pages.

Marijn J.H. Heule and Hans van Maaren. *Parallel SAT Solving using Bit-level Operations.* Accepted for Journal on Satisfiability, Boolean Modeling and Computation.

Marijn J.H. Heule and Hans van Maaren. *Whose side are you on? Finding solutions in a biased search-tree.* Submitted to Journal on Satisfiability, Boolean Modeling and Computation.

Hans van Maaren, Linda van Norden, and Marijn J.H. Heule. *Sums of squares based approximation algorithms for MAX-SAT.* Accepted for Discrete Applied Mathematics.

Marijn J.H. Heule and Hans van Maaren. *From Idempotent Generalized Boolean Assignments to Multi-bit Search.* SAT 2007 Springer LNCS **4501** (2007), pp. 134–147.

Marijn J.H. Heule and Hans van Maaren. *Effective Incorporation of Double Look-Ahead Procedures.* SAT 2007 Springer LNCS **4501** (2007), pp. 258–271.

Marijn J.H. Heule and Leon J.M. Rothkrantz. *Solving Games: Dependence of applicable solving procedures.* Elsevier Science of Computer Programming **67**(1) (2007), pp. 105–124.

Paul Herwig, Marijn J.H. Heule, Martijn van Lambalgen, and Hans van Maaren. *A new method to construct lower bounds for Van der Waerden numbers.* The Electronic Journal of Combinatorics **14** (2007), #R6.

Marijn J.H. Heule and Hans van Maaren. *March_dl: Adding Adaptive Heuristics and a New Branching Strategy.* Journal on Satisfiability, Boolean Modeling and Computation **2** (2006), pp. 47–59.

Marijn J.H. Heule and Hans van Maaren. *Observed Lower Bounds for Random 3-Sat Phase Transition Density using Linear Programming.* SAT 2005 Springer LNCS **3569** (2005), pp. 122–134.

Marijn J.H. Heule and Hans van Maaren. *Aligning CNF- and Equivalence-Reasoning.* SAT 2004 Springer LNCS **3542** (2005), pp. 145–156.

Marijn Heule, Joris van Zwieten, Mark Dufour and Hans van Maaren. *March_eq: Implementing Additional Reasoning into an Efficient Look-Ahead Sat Solver.* SAT 2004 Springer LNCS **3542** (2005), pp. 345–359.