Advanced Methods for SAT Solving

Dissertation
der Mathematisch–Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.–Inform. Stephan Kottler
aus Gernsbach

Tübingen
2012

Tag der mündlichen Qualifikation:	08.05.2013
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Michael Kaufmann
2. Berichterstatter:	Prof. Dr. Wolfgang Küchlin
3. Berichterstatter:	Dr. Daniel Le Berre

Acknowledgements

The time I have worked on my PhD thesis was very exciting and full of experiences, nice cooperation and memorable moments. I particularly thank my advisor, Prof. Dr. Michael Kaufmann, for the opportunity to work in a great research group, for his support and suggestions, for the possibility to attend several international conferences and for the freedom I was given to follow many interesting questions in research. The priority program 1307 Algorithm Engineering of the Deutsche Forschungsgemeinschaft (DFG) was a great chance to explore the field of algorithm engineering. I would like to thank the DFG for their financial support and my advisor as well as Prof. Dr. Wolfgang Küchlin for the opportunity to join their challenging project, Structure–based algorithm engineering for SAT solving (StrAlEn-Sats) [KK13]. Practical SAT solving would not be possible without the extensive evaluation of implementations. In this regard, I want to thank the bwGrid project [BWG12] for offering a convenient way to use plenty of computational resources. Furthermore, I want to thank Dr. Werner Dilling and his team at the Zentrum für Datenverarbeitung for their perfect support and, especially, for finding emergency solutions quickly (e.g. when the main cluster had troubles few days before a competition).

I have always appreciated the bright, friendly and uncomplicated atmosphere we have in our group. I am grateful to my current and former colleagues, namely Robert Krug, Christian Zielke, Philip Effinger, Till Bruckdorfer, Andreas Gerasch, Markus Geyer, Tamara Mchedlidze, Martin Siebenhaller and Katharina Zweig. I will certainly miss the Betriebssport, the enjoyable coffee breaks (including the discussions) and the sophisticated and constantly varying point system to reward sports activities of any kind. I am deeply grateful to the proofreaders of this thesis, Megan Foster, Christian Zielke and Kim Kottler, for their helpful feedback and comments. Particularly, I want to thank my wife Kim for all her patience (e.g. when the last bugs in the solver had to be fixed before we could finally leave for the honeymoon). Last but not least I want to thank my parents for supporting my studies.

Contents

Chapter 1

Introduction

The quarter–finals of the European Cup football club tournament in the 1964/65 season were particularly thrilling for the players and supporters of two teams. Both, the first leg and the second leg between the clubs *Liverpool FC* and *1. FC Köln* finished in a no–score draw. The former set of rules, known as the Laws of the Game, did not provide for penalty shootouts. To this end, an extra match was scheduled for the 24th of March 1965 to take place in Rotterdam, Netherlands. Again, the score was tied at the end of regulation time and even extra time did not change the scores. Referee Robert Schaut had to toss a coin to determine the winning team [GW07].

Flipping a coin is a simple and evenhanded way to break a tie rapidly, peacefully and reliably. The outcome is either heads or tails, winners or losers, semi–finals or elimination. When Schaut tossed a coin for the first time, it got stuck vertically in the mud [GW07]: neither heads nor tails, with no immediate[1] winner nor loser, an undefined state in a two–state model.

Boolean logic, the underlying concept of this thesis, knows exactly two values, *true* and $false$. A Boolean formula consists of a distinct set of Boolean variables. Either there is an assignment to the variables such that the formula evaluates to *true*, or any possible assignment to the variables evaluates to $false$. Many real–world problems, or parts of these problems, are modelled as Boolean expressions. Any Boolean expression can be transformed into a standardised formula — the conjunctive normal form (CNF) [Tse68]. Given a formula in CNF, a Satisfiability (SAT) solver may return an assignment to the variables of the formula that satisfies all constraints. Or it may prove that the given formula cannot be satisfied by any possible assignment.

[1]Liverpool FC won thanks to a second toss of the coin.

The problem to decide whether a Boolean formula in CNF is satisfiable is known to be NP–complete [Coo71]. Thus, no efficient algorithm is known that allows for every formula to be solved in reasonable time. Most researchers are convinced that no such poly–time algorithm exists. However, Davis, a renowned researcher in the field of SAT, is unconvinced, as he states in the Preface of the *Handbook of Satisfiability* [BHvMW09]. The basic algorithm he published in 1960 [DP60, DLL62] can still be identified in state–of–the–art SAT solvers.

Nevertheless, SAT solving is applied successfully in several domains. An early application was within the domain of planning [KS92]. The successful application of SAT to planning problems influenced the development of bounded model checking [BCCZ99, BHvMW09]. Along with equivalence checking [KK97, KPKG02], this is a basic concept for hardware verification, one of the most famous applications of SAT technology. Concrete applications and evaluations for practical hardware verification problems are publicly available [Vel02]. The consequent extension to software verification using SAT technology has already been made [CKL04, IYG$^+$08]. Moreover, configuration problems can be analysed and verified with the help of SAT solvers, such as the verification of automotive product configuration [SKK03]. A relatively recent application of SAT technology is within the field of bioinformatics [LMS06]. Several applications of SAT are presented and analysed by Marques-Silva [MS08] and are exhaustively discussed in the *Handbook of Satisfiability* [BHvMW09].

Despite the success of SAT solvers for many real–world SAT problems, many SAT instances cannot be solved even after several hours of computation time. The result of a SAT solver may be something like *Unknown* or *Timeout* for these infeasible problems. Therefore, solving Boolean formulae in practice has three possible outcomes. This is the point where Boolean logic meets reality and where practical SAT solving does not adhere to the two–state model.

Modern SAT solvers are not able to prove that $n + 1$ pigeons cannot be put into n distinct pigeonholes even for small input sizes. However, the use of solvers that apply extended resolution [Hua10, AKS10] may improve on this kind of formula in the future [Coo76]. Another example where SAT technology evidently failed in the recent past is the Eternity II puzzle, depicted in Figure 1.1: 256 square pieces with different patterns at each edge have to be placed on a 16×16 grid. A solution to the puzzle matches all patterns of adjacent edges. The first valid solution would have been rewarded with $\$2,000,000$. Different SAT formulations to solve the puzzle were presented [Heu08b, BFMP09]. However, the puzzle was not solved by any technique within three and a half years and the competition has since been discontinued. The author of this thesis spent several months on the attempt to tackle the problem with the use of SAT technology.

Both examples are so–called *crafted* problems that are developed to be extremely difficult. A choice to put a particular pigeon into some hole or to place a piece onto a position on the grid may not be found to be faulty until all the possibilities for all the remaining items have been explored. To improve SAT solving for different domains, the SAT community distinguishes between three different kinds of instances [Sat11]: crafted benchmarks, randomly created benchmarks and benchmarks that model real–world application problems. The latter type of benchmark is the main objective of this thesis.

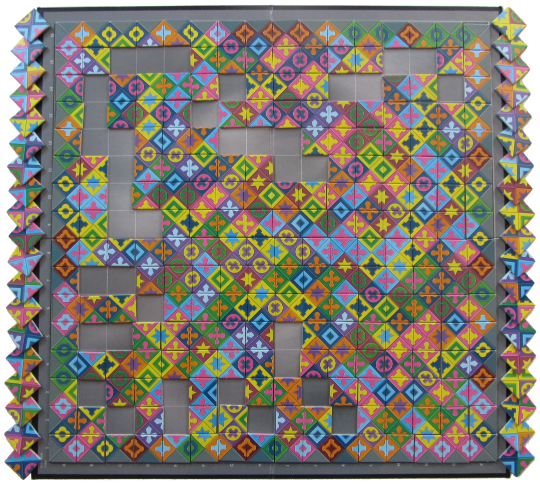


Figure 1.1: The Eternity II puzzle was published by Monckton, and was marketed by Tomy UK Ltd.

The predominant approach for tackling SAT instances from real–world applications is conflict–driven SAT solving with clause learning (CDCL). The algorithm goes back to 1960 [DP60, DLL62] and has been improved considerably since then [MS99, MMZ$^+$01, ES03, Bie08b]. The ongoing improvement of the CDCL procedure using real–world benchmarks demonstrates the concept of Algorithm Engineering[2]. The close interplay of the design, the analysis, implementation and experimental evaluation of practicable algorithms allows for the optimisation of an algorithm for particular applications. Nowadays, state–of–the–art CDCL solvers are highly tuned and several parameters have been determined carefully.

As a consequence of tuning, modern CDCL solvers are highly sensitive to minor changes of parameters. Audemard and Simon address the question whether the practical improvement of a CDCL solver may sometimes originate from a side–effect rather than from the technique [AS08]. Put differently, some promising approaches may produce bad experimental results that arise from some incompatibilities with unknown side–effects rather than by the approach itself. This issue evidently complicates the evaluation of SAT approaches in practice.

[2]`http://www.algorithm-engineering.de`

A corroborative example is presented in Figure 1.2. The well known CDCL solver MiniSat [ES03, ES12] maintains a heap to store heuristic information for variables. We modified the behaviour of the heap in MiniSat2.2 solely for the case when two variables have equal values. A point (x, y) in the plot indicates that x instances can be solved when the time per instance is limited to y seconds. Thus the more to the right the curve is, the more successful the solver is.

The single–line modifications for a seemingly marginal issue already exhibit quite an impact. When MiniSat is run without preprocessing (Figure 1.2 a), the original heap implementation performs best for almost all time limits. However, the stable version of the heap performs much better when preprocessing is also applied.

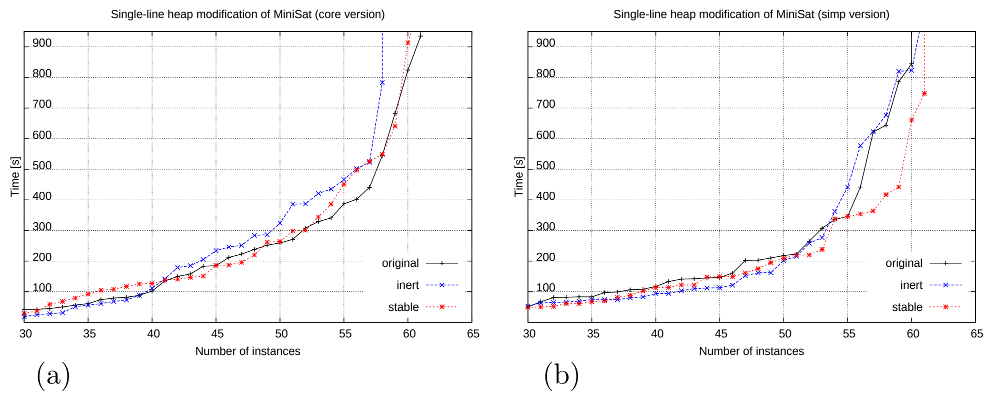


Figure 1.2: The sensitivity of conflict–driven SAT solvers. The two plots show the effect of single–line modifications to the MiniSat2.2 heap (a: pure CDCL, b: with preprocessing). Instances are taken from the SAT–Race 2010. These modifications are revisited in Section 2.2.5.

This work explores SAT solving techniques that go beyond small changes to the predominant CDCL approach. Whilst this work starts with techniques that are close to state–of–the–art CDCL solving, it goes further, and explores and evaluates some rather uncommon ideas. The general purpose is to widen the range of instances for which SAT solvers may compute a result in reasonable time. In the following, we give a brief overview of the chapters of this thesis.

Basic definitions and concepts are introduced in Chapter 2. The approach of CDCL is explained in great detail, which will be referred to in subsequent chapters. Moreover, the most common heuristics of state–of–the–art SAT solvers are described.

Chapter 3 presents some extensions that can be incorporated into CDCL solvers. The first part of the chapter introduces a novel improvement of the data structure to represent clauses within a CDCL solver. The remainder of this chapter studies the enhancement of simplification techniques for SAT formulae. The underlying techniques are asymmetric branching [HS07, PHS08] and hyper–binary resolution as proposed by Bacchus *et al.* [Bac02a, BW03] and later improved by Biere [Bie09a]. Simplification is predominantly applied as preprocessing before the actual CDCL algorithm starts [SP04, EB05]. Moreover, some modern solvers also apply simplification techniques in between CDCL, known as inprocessing [Bie09b, Bie11]. We propose an algorithm to improve the quality of both techniques, and evaluate its application for preprocessing and inprocessing. Some of the material in this chapter has already been published [KK11c, KK11b], though only a rough outline of the simplification algorithm is given in these publications.

A crucial part of conflict–driven SAT solving is the so–called Boolean constraint propagation of assignments. Any clause that has only one literal left implies an assignment of the corresponding variable. While the CDCL algorithm only considers the case where one literal is left, in Chapter 4 we extend Boolean constraint propagation to more general cases where any number of literals may be left. The technique is based on the general concept of hyper–resolution that was introduced by Robinson [Rob83]. Two different implementations are evaluated to study the tradeoff between speed and quality. The second approach extends the concept of dominators that was suggested by Biere [Bie09a] and transfers it to our extension of Boolean constraint propagation. The main points of this chapter are also presented in [KK11a].

We further depart from the standard CDCL algorithm by exploring the alternative DMRP solving approach in Chapter 5. Decision making with reference points (DMRP) was introduced by Goldberg [Gol08a]. Compared to the CDCL algorithm the DMRP approach requires more information for SAT solving. Consequently, more effort has to be spent on maintaining the underlying data structures. We present an efficient implementation for the increased requirements of the DMRP algorithm. Moreover, we suggest a hybrid approach that is competitive to pure CDCL solving. The work presented in this chapter has also been published in [Kot10a].

All the techniques and approaches presented in this thesis have been combined within one SAT solver. Chapter 6 presents the implementation of our parallel solver, SArTagnan, with a high degree of information sharing among different threads. More complex techniques are justified by the benefits of having several solvers running in parallel. The concept and architecture of SArTagnan have been published in [KK11c, KK11b].

While the author was working in the field of SAT solving, various SAT–related projects have arisen over the last years. Some are related to co–supervised student research projects; others were preliminary ideas that gradually turned into exciting projects. Chapter 7 briefly sketches some of these projects without claim to completeness. Finally, the contributions of the thesis are summarised in Chapter 8.

Chapter 2

Preliminaries

The satisfiability (SAT) problem has been studied by different research communities. Some communities are more focused on theoretical analysis of the NP–complete SAT problem [Coo71]. Bounds on the complexity to solve SAT or subproblems of SAT have been steadily improved. Bounds for both, deterministic and probabilistic algorithms have been proven [Sch99, PS04, PS07, KS10, KKS08a]. Furthermore, there are several restricted classes of the SAT problem where satisfiability can be decided in polynomial time, such as Horn–formulae or 2–SAT [FG03, APT79].
On the other hand, SAT has been analysed from a practical point of view. Several real–world problems are modelled as SAT problems and are then computed with the help of efficient SAT solvers. Problems originated from hardware and software verification [BCCZ99, Sht01, Vel02, IYG$^+$08], automotive product configuration [SKK03] and bioinformatics [MS08] are successfully tackled by state–of–the–art SAT solvers. The *Handbook of Satisfiability* [BHvMW09] gives an overview of several aspects and applications of SAT.
In different areas, different notations are frequently used. This chapter defines the basic notation used in this work and introduces the most relevant concepts. Some additional definitions and algorithms will be introduced within later chapters. For basic concepts from graph theory and algorithms we refer the reader to comprehensive textbooks such as [CLRS01].

2.1 Boolean Formulae and Satisfiability

The SAT problem is a decision problem that examines Boolean expressions. A Boolean expression consists of Boolean variables; the operations conjunction (AND, $\wedge$), disjunction (OR, $\vee$), and negation (NOT, $\neg$); constants $true$ and $false$; and parentheses. Further Boolean operations can be derived, e.g. exclusive or (XOR, $\oplus$), implication ($\rightarrow$) and equality ($\leftrightarrow$). Given a Boolean expression the SAT problem asks whether an assignment to the variables exists such that the expression evaluates to *true*. The ground–breaking work by Davis and Putnam [DP60] started to consider Boolean formulae in conjunctive normal form (CNF), to which any Boolean expression can be transformed.

2.1.1 Conjunctive normal form

A formula $\mathcal{F}$ in CNF is a set of clauses that are connected as conjunctions. Let $\mathcal{V}$ be the set of Boolean variables of $\mathcal{F}$. A clause $C \in \mathcal{F}$ is a disjunction of $|C|$ literals, whereas each literal is either a variable or its negation. A clause C is called *unit* if it contains only one literal ($|C| = 1$), *binary* if $|C| = 2$ and *ternary* if $|C| = 3$. We refer to the subset of clauses $\mathcal{F}_2 \subseteq \mathcal{F}$ that contains all unit and binary clauses of $\mathcal{F}$ as $\mathcal{F}_2 = \{C \in \mathcal{F} : |C| \leq 2\}$.

To access a literal within a clause $C \in \mathcal{F}$, we use square brackets. Thus $C[k]$ yields the k-th literal in C whereas $0 \leq k < |C|$. A literal is said to have negative *polarity* if it is a negation of a variable ($\overline{\nu_i}$) and to have positive polarity when it is not negated (ν_i). Double negation of a variable yields the variable itself: $\overline{\overline{\nu_i}} = \nu_i \ \forall \ \nu_i \in \mathcal{V}$. Note that some researchers use the term *phase* as a synonym for the polarity of a variable.

We distinguish between a variable itself, ν_i, and the corresponding literals, λ_i and $\overline{\lambda_i}$. This allows for a more generic access of a particular literal in a clause C. We may write $\lambda_i \leftarrow C[k]$ to access the k-th literal of C even if it is unclear or unimportant whether the literal occurs with positive or negative polarity. λ_i may thus represent ν_i or its negation $\overline{\nu_i}$. The *complementary literal*, $\overline{\lambda_i}$, represents $\overline{\nu_i}$ or ν_i respectively. Consequently, double negation of a literal is the literal itself: $\overline{\overline{\lambda_i}} = \lambda_i$. The set of all literals $\mathcal{L}$ contains all variables in positive and negative polarity: $\mathcal{L} = \{\lambda_i, \overline{\lambda_i} : \nu_i \in \mathcal{V}\}$. Note that the subscript (j) of a literal λ_j is never used to denote the position of the literal within any clause.

An *assignment function* is a function $A \mapsto \{false, true\}$ that assigns Boolean values to all variables of $A \subseteq \mathcal{V}$. We define a *partial assignment* τ as a set of assignment tuples (ν_i, b), where $\nu_i \in A \subseteq \mathcal{V}$ and $b \in \{false, true\}$. Obeying the rule of an assignment function, a variable ν_i can be contained in at most one assignment tuple in τ. An assignment is *complete* if it assigns Boolean values to all variables in $\mathcal{V}$, i.e. $A = \mathcal{V}$ in the assignment function; accordingly, τ contains an assignment tuple for all variables.
Assigning a Boolean value $b \in \{false, true\}$ to a variable ν_j, extends the partial assignment τ to $\tau \cup \{(\nu_j, b)\}$. If b is *true*, we also say that literal λ_j is assigned. Consequently, the partial assignment τ is extended to $\tau \cup \lambda_j$. Analogously, for $b = false$, the literal $\overline{\lambda_j}$ is assigned and τ is extended to $\tau \cup \overline{\lambda_j}$.

In the context of partial assignments, a clause $C \in \mathcal{F}$ can be partitioned into three disjoint subsets. The set $\mathtt{T}_\tau(C) = \{\lambda_j \in C : \lambda_j \in \tau\}$ indicates the subset of literals within C that are *true* by the current partial assignment τ. The set $\mathtt{F}_\tau(C) = \{\lambda_j \in C : \overline{\lambda_j} \in \tau\}$ is the subset of literals of C that are falsified (assigned the complementary value) by the partial assignment τ. Finally, the set $\mathtt{U}_\tau(C) = \{C \setminus \{\mathtt{T}_\tau(C) \cup \mathtt{F}_\tau(C)\}\}$ is the subset of the literals of C whose corresponding variables are not assigned in τ.

Clause C is *satisfied* under τ iff $\mathtt{T}_\tau(C) \neq \emptyset$, and it is *falsified* under τ iff $\mathtt{F}_\tau(C) = C$ $(\mathtt{T}_\tau(C) = \mathtt{U}_\tau(C) = \emptyset)$[1]. A clause C is unit under τ iff $|\mathtt{U}_\tau(C)| = 1$ and $|\mathtt{F}_\tau(C)| = |C| - 1$, and binary under τ iff $|\mathtt{U}_\tau(C)| = 2$ and $|\mathtt{F}_\tau(C)| = |C| - 2$ ($\mathtt{T}_\tau(C) = \emptyset$ in both cases). A complete assignment that satisfies all clauses of $\mathcal{F}$ is called a *model* for $\mathcal{F}$. A formula is satisfiable iff at least one model exists; it is unsatisfiable if no such assignment exists.

In some applications of SAT, unsatisfiable formulae may be further analysed to determine a so–called *minimal unsatisfiable core.* An unsatisfiable core for a formula $\mathcal{F}$ is a subset of its clauses $\mathcal{F}_u \subseteq \mathcal{F}$ that is unsatisfiable. If $\mathcal{F}_u$ is a minimal unsatisfiable core, the removal of any clause $C \in \mathcal{F}_u$ will yield a satisfiable instance, $\mathcal{F}_u \backslash C$. The computation of minimal unsatisfiable cores is revisited in Section 7.5.

2.1.2 Resolution

The resolution rule in Boolean logic is an inference rule that allows for the creation of a new valid clause [Rob65, Rob79]. It requires two clauses, $C_1, C_2 \in \mathcal{F}$, that contain a complementary literal. Let $C_1 = (\lambda_i \vee A)$ and $C_2 = (\overline{\lambda_i} \vee B)$, where A and B are a disjunction of some literals in $\mathcal{L}$. The resolution on the variable ν_i is formally written:

$$\frac{(\lambda_i \vee A), (\overline{\lambda_i} \vee B)}{(A \vee B)}.$$

The derived clause $(A \vee B)$ is called the *resolvent* of C_1 and C_2 on the variable ν_i. If A and B contain a pair of complementary literals, then $(A \vee B)$ is a tautology, i.e. it is satisfied for any assignment of variables.

The resolution rule is refutation complete: A formula $\mathcal{F}$ is unsatisfiable iff the empty clause can be derived by resolution. Moreover, the rule of resolution allows for the *existential quantification* of any variable $\nu_k \in \mathcal{V}$. Let $\mathcal{K}_p, \mathcal{K}_n \subseteq \mathcal{F}$ be the sets $\mathcal{K}_p = \{C \in \mathcal{F} : \lambda_k \in C\}$ and $\mathcal{K}_n = \{C \in \mathcal{F} : \overline{\lambda_k} \in C\}$. Let $\mathcal{K}_\times$ contain all clauses that can be derived by the resolution on variable ν_k with any pair of clauses $C_p \in \mathcal{K}_p, C_n \in \mathcal{K}_n$. The formula $\mathcal{F}$ is

[1]We use *iff* as an abbreviation for *if and only if.*

equisatisfiable to the formula $\mathcal{F}' = \mathcal{F} \cup \mathcal{K}_\times \setminus \{\mathcal{K}_p \cup \mathcal{K}_n\}$ in which variable ν_k is not contained. Thus $\mathcal{F}$ is satisfiable if and only if $\mathcal{F}'$ is satisfiable.

2.2 Conflict–Driven Solving with Clause Learning

SAT solvers can generally be categorised into two distinct types, namely *complete* and *incomplete* solvers. Given a formula $\mathcal{F}$ in CNF, both kinds of solvers may compute a satisfying assignment for $\mathcal{F}$. However, complete solvers can also prove *unsatisfiability* for formulae that cannot be satisfied by any assignment to the variables in $\mathcal{V}$.

Incomplete solvers are mostly local search approaches [SKC93], which have been shown to be especially successful for satisfiable random SAT instances [Sat11]. Solving SAT with local search is beyond the scope of this work and we refer the reader to Hoos *et al.* [Hoo98, HS04] and Kautz *et al.* (Chapter 6 in [BHvMW09]). Complete solvers are based on the DPLL algorithm [DP60, DLL62], which may be classified as a branch–and–bound algorithm. State–of–the–art solvers are predominantly variants of CDCL that is an extension of the original DPLL algorithm [MSS99]. Both algorithms are introduced in this section. The effects of different modifications to the original algorithm are studied by Katebi *et al.* [KSMS11].

2.2.1 The DPLL algorithm

The often cited DPLL algorithm is the basis of today's SAT solving algorithms. The elementary rules are introduced in [DP60] and are outlined below (using the established names). In [DLL62], the third rule given below was replaced by the fourth rule to cope with memory restrictions [BHvMW09].

- By the unit clause rule, any clause $(\lambda_i) \in \mathcal{F}$ allows for the following simplification: remove all clauses that contain the literal λ_i and remove the literal $\overline{\lambda_i}$ from any clause it is contained in.
- The pure literal rule can be applied whenever there is a literal λ_i that does not occur in the formula $\mathcal{F}$, i.e. none of the clauses in $\mathcal{F}$ contains λ_i. In that case, all clauses that contain $\overline{\lambda_i}$ can be removed from $\mathcal{F}$.
- A variable ν_i and all the clauses it is contained in can be eliminated when all deducible resolvents on this variable are added to the formula. The concept of existential quantification has been described earlier in Section 2.1.2.
- The splitting or branching rule recursively chooses a variable ν_d and examines both subproblems $\mathcal{F} \cup (\lambda_d)$ and $\mathcal{F} \cup (\overline{\lambda_d})$.

2.2.2 Boolean constraint propagation

The main ingredient of a modern SAT solver is the so–called Boolean constraint propagation (BCP). For CDCL solvers, this is equal to unit propagation and goes back to the first rule of the DPLL algorithm mentioned in Section 2.2.1.

A crucial issue for the efficient implementation of unit propagation in CDCL solvers is the application of the *two watched literals* scheme, whose efficiency was impressively demonstrated by the success of the Chaff solver [MMZ+01]. To detect whether a clause C is unit under the partial assignment τ, it is sufficient to watch two literals of the clause. With this, each literal $\lambda_i \in \mathcal{L}$ holds a *watcher list* that contains all clauses where λ_i is one of the two watched literals. If the complementary literal $\overline{\lambda_i}$ is assigned ($\tau \leftarrow \tau \cup \overline{\lambda_i}$), the watcher list of λ_i is traversed to detect those clauses that become unit or falsified by the assignment of $\overline{\lambda_i}$. To apply the two watched literals scheme, we assume that no clause contains duplicate literals. Moreover, tautological clauses that contain a literal and its complement are assumed to be removed. Algorithm 2.1 sketches the complete process of unit propagation.

Boolean constraint propagation expects an unassigned literal to be assigned in the partial assignment τ. Nevertheless, the two cases when literal λ_a or its complement are already assigned are handled at the beginning of Algorithm 2.1 (line 4 *et seq.*). In the first case, nothing is left to be done; however, the latter case conflicts with the unit clause $(\overline{\lambda_a})$.
The assignment is applied in line 6 and the literal is enqueued into Q for subsequent propagation. A crucial issue of CDCL solvers is to store a reason for each assignment. This is essential for conflict analysis and was proposed by Marques-Silva and Sakallah in the GRASP algorithm (Generic seaRch Algorithm for the Satisfiability Problem) [MSS96, MSS99]. The *reason* for an assignment λ_k is accessed by the function $\texttt{rsn}(\lambda_k)$. It may be a clause C that is unit under the current partial assignment τ and thus forces the assignment of the remaining literal λ_k. In that case, C is also called the *asserting clause* for λ_k. If the assignment is not forced by a unit clause, the reason is empty ($\circ$). Since the assignment of λ_a is supplied externally, $\texttt{rsn}(\lambda_a)$ is set to $\circ$ in line 8. Moreover, for each assignment, the current *decision level* is stored by the function `level`. The decision level represents the number of decisions in the current branch. This value is constant for one call to the function `BCP`.

As long as there are some new assignments that have not been propagated, the next assigned literal, $\lambda_q \in \tau$, is chosen from the queue Q in line 11. In the subsequent propagation, the watcher list of the complementary literal $\overline{\lambda_q}$ is traversed. As described above, this list contains all clauses where $\overline{\lambda_q}$ is one of the two watched literals.

Algorithm 2.1: Boolean constraint propagation

```
   Require Literal $\lambda_a$ to be assigned
   Return Conflicting clause or OK if no conflict arises
   Function BCP ($\lambda_a$)
4      if $\lambda_a \in \tau$ then return OK
       if $\overline{\lambda_a} \in \tau$ then return $(\overline{\lambda_a})$        | conflicting unit clause
6      $\tau \leftarrow \tau \cup \lambda_a$                                  | assign variable
       $Q \leftarrow \{\lambda_a\}$                   | queue of assignments to propagate
8      rsn($\lambda_a$) $\leftarrow$⬡                            | no reason for assignment
       level($\nu_a$) $\leftarrow$ current decision level
       while $Q \neq \emptyset$ do
11         $\lambda_q \leftarrow Q$.dequeue()              | next literal to propagate
           $W_{\overline{q}} \leftarrow$ watchedOf($\overline{\lambda_q}$)          | clauses with watched $\overline{\lambda_q}$
           foreach $C^* \in W_{\overline{q}}$ do
14             $\lambda_p \leftarrow$ otherWatched($C^*, \overline{\lambda_q}$)
15             if $\lambda_p \in \tau$ then continue
               if $\exists\ \lambda_k \in \mathtt{U}_\tau(C^*) \cup \mathtt{T}_\tau(C^*) \setminus \{\lambda_p\}$ then
17                 $W_{\overline{q}} \leftarrow W_{\overline{q}} \setminus C^*$                       | link new watched
                   $W_k \leftarrow W_k \cup C^*$
               else if $\overline{\lambda_p} \notin \tau$ then
20                 $Q$.enqueue($\lambda_p$)             | further unit propagation
                   $\tau \leftarrow \tau \cup \lambda_p$
22                 rsn($\lambda_p$) $\leftarrow C^*$                | reason for assignment
                   level($\nu_p$) $\leftarrow$ current decision level
24             else return $C^*$                          | conflicting clause
       return OK
```

For each clause C^* in the watcher list of $\overline{\lambda_q}$, the other watched literal is accessed as λ_p in line 14. Van Gelder proposed to keep the two watched literals at the front of each clause [Gel02]. On one hand, this requires the internal order of literals to be rearranged whenever one of the watched literals is interchanged. On the other hand, it allows for constant and simple access to both watched literals. In Section 3.1, we propose a further improvement of this issue.

If the other watched literal, λ_p, is assigned in τ (line 15), the clause is already satisfied under τ and the loop continues with the next clause. Note that this applies for the bulk of inspected clauses in practice. Therefore, efficient implementations of Boolean constraint propagation take advantage of this fact and implement an additional caching mechanism described by Chu *et al.* [CHS09].

If the clause C^* is not immediately detected as being satisfied by τ, it has to be inspected in depth. More precisely, it is checked to ascertain whether there is another literal, λ_k in C^*, which is either *true* under the partial assignment τ or not yet affected by τ. If this is the case, λ_k can become a new watched literal of C^* together with λ_p. Thus, in line 17, C^* is moved into the watched list of λ_k. Obviously, any new watched literal λ_k must be distinct from literal λ_p in order to have two different watched literals.

If there is no other literal that can become a watched literal for C^*, clause C^* is either unit under τ or it is falsified by τ. The latter case is handled in line 24 and C^* is returned as a conflicting clause for the partial assignment τ. Conflict analysis takes over at this point, as will be described in Section 2.2.4. If, on the other hand, λ_p is the only literal in clause C^* that is not falsified by τ, another implication is detected. Thus, literal λ_p is enqueued to initiate further unit propagation in line 20 and is assigned in τ subsequently. At this point, C^* is the asserting clause for the assignment of λ_p, which is stored by the function $\texttt{rsn}(\lambda_p)$ in line 22. Moreover, the current number of decisions is also stored by the function $\texttt{level}(\nu_p)$ analogously.

For a set of assignments $K \subseteq \tau$ that implies the assignment of λ_p by the application of unit propagation, we write $K \xrightarrow{\text{UP}} \lambda_p$ for short. If the implication is deduced by the propagation of binary clauses (i.e. clauses of $\mathcal{F}_2$), we write $K \xrightarrow{\text{UP2}} \lambda_p$ for short.

In Chapter 4, we present an enhancement of Boolean constraint propagation. The approach presented in that chapter does not necessarily require a clause to be unit in order to detect implied assignments.

2.2.3 From DPLL to CDCL

Since the introduction of the original DPLL approach, several heuristics and improvements have come and gone. Algorithm 2.2 lists the most important issues of the CDCL algorithm. Details of the subroutines are explained further below and the most common heuristics are presented in Section 2.2.5.

The partial assignment τ is initialised to contain all unit clauses of the given formula $\mathcal{F}$. Moreover, these assignments are propagated in line 5 by the `BCP` procedure presented in Section 2.2.2. The CDCL algorithm searches for an assignment of the variables that satisfies all clauses in $\mathcal{F}$. For the satisfiable formulae, all variables are assigned even if a small subset of variables is sufficient to satisfy all clauses. In line 8, the loop continues until the partial assignment τ is a complete assignment. In line 9, a decision heuristic is applied to decide an assignment for a currently unassigned variable. Decision heuristics are examined in more detail in Section 2.2.5. The assignment

Algorithm 2.2: Outline of the CDCL approach

```
Require Formula $\mathcal{F}$ in CNF
Return Sat or UnSat
Function CDCL ($\mathcal{F}$)
    $\tau \leftarrow \emptyset$                    | $\tau$ is current partial assignment
5   forall $(\lambda_i) \in \mathcal{F}$ do
        $C \leftarrow$ BCP($\lambda_i$)
        if $C \neq$ OK then return UnSat
8   while $|\tau| < |\mathcal{V}|$ do
9       $\lambda_p \leftarrow$ chooseNextDecision($\mathcal{V} \setminus \tau$)
10      $C \leftarrow$ BCP($\lambda_p$)
11      while $C \neq$ OK do
12          $C^* \leftarrow$ analyseConflict($C$)
13          if $C^* = \emptyset$ then return UnSat
14          $\mathcal{F} \leftarrow \mathcal{F} \cup C^*$
15          $\lambda_q \leftarrow \lambda_q \in C^*$ : level($\nu_q$) $\geq$ level($\nu_k$) $\forall\ \lambda_k \in C^*$
16          $\tau \leftarrow$ backjump($C^*$)
17          $C \leftarrow$ BCP($\lambda_q$)
18          rsn($\lambda_q$) $\leftarrow C^*$
    return Sat                         | assignment $\tau$ satisfies $\mathcal{F}$
```

is applied and propagated in line 10 by the previously presented Algorithm 2.1. If a conflict arises within Boolean constraint propagation, the conflicting clause is analysed within the loop starting at line 11.

In line 12, the function `analyseConflict` generates a learnt clause C^*, a so–called lemma. This technique is explained in detail in Section 2.2.4. However, the following three properties are important here:

- The conflicting clause C contains at least two literals, λ_a and λ_b, whose complementary literals ($\overline{\lambda_a}$ and $\overline{\lambda_b}$) are assigned at the current decision level. Let us assume that only one literal $\lambda_a \in C$ is falsified at the current decision level. In that case, C would have been an asserting clause for the assignment of λ_a after a previous decision. If C had no literals from the current decision level, it would have been a conflicting clause at an earlier level in the search.
- The lemma C^* is certain to contain at most one literal λ_q, whose complement $\overline{\lambda_q}$ was assigned at the current decision level. It may also be the empty clause, which means that formula $\mathcal{F}$ is unsatisfiable, as it is handled in line 13. Otherwise, literal λ_q is chosen in line 15.
- The generated lemma is falsified by the current partial assignment τ.

The lemma C^* is learnt by adding it to the formula $\mathcal{F}$ in line 14. In most CDCL solvers, lemmas are marked as redundant clauses. This allows the set of learnt clauses to be reduced from time to time, when the formula contains too many constraints (see Section 2.2.5).

The function `backjump`(C^*), invoked in line 16, undoes some assignments and removes them from τ. More precisely, the first d decisions (together with their implied assignments) are kept in τ, such that C^* becomes unit under τ. Hence, C^* becomes an asserting clause for λ_q at level d. Thus, after jumping back to level d, literal λ_q is assigned and propagated in line 17, and C^* is stored as the asserting clause in line 18. Note that if C^* is a unit clause, all decisions are undone and the algorithm jumps back to level zero. Again, if a conflict arises within BCP, the conflicting clause is analysed by continuing in line 11.

2.2.4 Clause learning

The motivation for clause learning in CDCL solvers, as proposed by Marques-Silva [MS99], is twofold. A generated lemma reflects a sequence of branching decisions that led to a contradiction within search. Adding a lemma to the formula may thus prevent the solver from running into the same conflict in a different branch. Secondly, the generated lemma enables the solver to go back to a previous decision level (line 16 of Algorithm 2.2). The search may jump back over some irrelevant decisions and thereby avoids following some branches where no solution can be found. In Section 2.2.3, three properties regarding conflicting clauses and learnt lemmas are listed. Algorithm 2.3 presents the procedure to meet the listed demands.

Algorithm 2.3 takes a clause C' that is falsified under the current partial assignment τ. As explained above, C' has at least two literals that are falsified at the current decision level d. In line 5, the first version of the generated lemma is initialised to contain all literals of C' that are not falsified at decision level zero. Assignments that are made at level zero will not be changed during the search anymore. Thus the first rule of the DPLL procedure, presented in Section 2.2.1, is applied to omit these literals in the generated lemma. Note, however, that literals from level zero cannot simply be omitted when SAT solving is applied to compute minimal unsatisfiable cores as presented in Section 7.5.

The final lemma C^* has to contain at most one literal from the current decision level. Unless C^* is the empty clause, it contains exactly one such literal. The crucial idea at lemma generation is to replace literals from the current decision level by using the corresponding asserting clauses for resolution. This idea is applied within the loop starting at line 6 of Algorithm 2.3.

Algorithm 2.3: Analyse conflict and generate lemma

```
    Require Conflicting clause $C' \in \mathcal{F}$ with $\mathtt{F}_\tau(C') = C'$
    Return Generated lemma $C^*$
    Function analyseConflict $(C')$
        $d \leftarrow$ current decision level
 5      $C^* \leftarrow \{\lambda_i \in C' : \mathtt{level}(\nu_i) > 0\}$          | generated lemma
 6      while true do
            if FUIP then
 8              $D \leftarrow \{\lambda_i \in C^* : \mathtt{level}(\nu_i) = d\}$
 9              if $|D| < 2$ then return $C^*$
            else
11              $D \leftarrow \{\lambda_i \in C^* : \mathtt{rsn}(\overline{\lambda_i}) \neq \bigcirc\}$
12              if $D = \emptyset$ then return $C^*$
13          $\lambda_p \leftarrow \lambda_p \in D : \nu_p$ most recent assignment
14          $C_p \leftarrow \{\lambda_i \in \mathtt{rsn}(\overline{\lambda_p}) : \mathtt{level}(\nu_i) > 0\}$     | reason for $\overline{\lambda_p}$
15          $C^* \leftarrow C^* \cup C_p \setminus \{\overline{\lambda_p} \cup \lambda_p\}$     | resolvent of $C^*$ and $C_p$
```

The appropriate literals are replaced in reverse order to how they were assigned during the search.

Modern solvers apply the *first unit implication point* (FUIP) scheme for generating learnt clauses [MMZ$^+$01]. The FUIP method generates a lemma that is close to the conflict. When variables are used for resolution in reverse order to how they were assigned, this is the first clause that contains only one literal assigned at the current decision level. A clear and efficient implementation of the FUIP scheme is presented and explained by Ryan [Rya04].

In line 8, the application of the FUIP scheme considers all literals of the current version of C^* that are assigned at the current decision level. If there are less than two such literals, C^* represents the FUIP and the clause is returned in line 9. The next literal λ_p to be replaced in C^* is chosen in line 13. To replace literals in reverse order, this is the one in D that was assigned most recently. Recall that λ_p is falsified by the partial assignment τ. Hence, we have $\overline{\lambda_p} \in \tau$. Let us assume that the assignment of $\overline{\lambda_p}$ was implied by unit propagation. In that case, an asserting clause $C_p = \mathtt{rsn}(\overline{\lambda_p})$ was stored in line 22 of Algorithm 2.1. For the same reason as above, only the literals of C_p that are not falsified at level zero are considered (line 14).

In line 15, resolution on variable ν_p is applied by using the asserting clause C_p and the current version of C^*. The new version of C^* is set to the resolvent of this resolution operation. At this point, the following properties can be observed:

- The assignment $\overline{\lambda_p}$ was implied by unit propagation when all other literals of C_p were already falsified. Thus, all literals that replace λ_p in C^* by the resolution operation were assigned prior to the assignment of $\overline{\lambda_p}$.

- λ_p can never be added to C^* again at a later iteration of the loop. This is because the literals are replaced in reverse order to the order in which the assignments were made. Thus the algorithm moves strictly backwards regarding the order of assignments.

- It can safely be assumed that the assignment of $\overline{\lambda_p}$ was an implication. Let us assume the contrary: that it was a decision at level d. Consequently, it is the first assignment at this level. Hence, it is the last assignment that is reached in reverse order, and thus the only literal in D. In that case, C^* would have been returned in line 9.

- A unit implication point will always be found. The algorithm can replace, at most, the literals that were implied at the current decision level d. The loop will terminate when C^* contains only the decision variable that instituted level d at the latest.

- Each version of C^* contains only the literals falsified by τ. This is ensured at the beginning of the function. All literals that are added by the union with $C_p \setminus \overline{\lambda_p}$ in line 15 are also falsified by τ.

The FUIP scheme is widely applied in modern SAT solvers. However, we use a more general procedure in Section 3.2. A different learning scheme may create a learnt clause that consists purely of decision variables [MSS99]. This is realised when Algorithm 2.3 implements the `else` branch in line 11. The set D of literals that are supposed to be replaced in C^* is initialised accordingly. D contains all literals of the current lemma C^* whose complement is not assigned by a decision, i.e. that have an asserting clause. If D is empty, the function `analyseConflict` returns the generated lemma C^* in line 12. Even though the latter scheme is rarely applied, the idea can be used to minimise learnt clauses [SB09, ES12].

To explore the process of generating learnt clauses and to study several properties related to learning, we have implemented the tool CoPAn (Conflict Pattern Analysis). This tool is briefly described and illustrated in Section 7.1. As well as performing in–depth analysis of properties, CoPAn can also visualise two different kinds of graphs related to learning and allow the user to explore them interactively. One kind of graph is the so–called *implication graph* [MSS99].

In this directed graph, each assigned variable (literal) is represented by one vertex. Vertices that represent decisions have no incoming edges. The vertex that represents the assignment λ_i with asserting reason $C_i = \texttt{rsn}(\lambda_i)$ has $|C_i| - 1$ incoming edges. There is one incoming edge for each assignment that falsifies a literal in C_i. More precisely, there is a directed edge $(\overline{\lambda_k}, \lambda_i)$ in the implication graph iff $\lambda_k \in C_i$ and $\lambda_k \neq \lambda_i$. The procedure `analyseConflict` (see Algorithm 2.3) starts at a conflicting assignment and traverses the edges backwards in the graph. A learnt clause represents a cut through the implication graph.

2.2.5 Heuristics

Heuristics constitute a crucial part for state–of–the–art SAT solvers. Small modifications of a heuristic may lead to major differences in the behaviour of a solver. We distinguish three different areas where heuristics are applied.

Decision heuristics

Different decision heuristics for branching have been studied extensively. Marques-Silva evaluated the most successful heuristics developed up to 1999 [MS99]. The proposed two watched literals scheme of Moskewicz *et al.* [MMZ+01] entails the need for a different decision heuristic. This is because pure CDCL solvers maintain only a small amount of information about the actual solving state. As mentioned above, the solver does not know which clauses are satisfied by a partial assignment τ. Only the reverse case is detected, when a clause is falsified by τ. However, complex decision heuristics require more knowledge about the state of a search. To this end, the engineers of the Chaff solver proposed the *Variable State Independent Decaying Sum* (VSIDS) heuristic for decision making [MMZ+01]. The idea of the VSIDS heuristic is to prefer the variables that contributed to the most recent conflicts for branching. The original idea can be summarised by the following three rules:

- Store an activity value for each literal to record how often it was involved in a conflict. When a conflict arises, the activity values of all contributing literals are increased.
- Consider all unassigned variables for decision making. Among these, choose the literal with the highest activity value.
- Divide all activity values by a constant from time to time. With this, a contribution to more distant conflicts becomes less relevant compared to the contribution to recent conflicts.

The heuristic has been modified by Eén and Sörensson in MiniSat to store activity values for variables instead of literals [ES03]. In the first versions of MiniSat, a decision variable has always been assigned to *false*. Moreover, a heap is implemented to easily access the variable with the highest activity for the next decision. When a conflict arises, the activity values of the following variables are increased: All variables in the generated lemma and all variables that are resolved by the function `analyseConflict` in line 13 of Algorithm 2.3.

Modern SAT solvers mostly apply *phase saving* to assign a value to a decision variable. The technique, suggested by Pipatsrisawat and Darwiche [PD07a, PD07b], caches the assigned value of each variable when it is unassigned during backtracking. When a variable is chosen as decision variable it is assigned to the previously cached value. In many state–of–the–art solvers, almost 100% of the decisions are made by using the VSIDS heuristic. A small percentage of decisions may choose a variable at random. This forces the solver to look into different areas that may have been neglected so far.

Look–ahead solvers utilise still other techniques for the process of decision making. These kind of solvers are very successful on crafted and random unsatisfiable benchmarks. For an overview on look–ahead solvers and solving techniques we refer the reader to the work of Heule [Heu08a].

In Chapter 5, we will explore a decision strategy for real–world SAT problems that is completely different from the VSIDS heuristic. The approach that was introduced by Goldberg [Gol08a] requires more knowledge of the state of a SAT solver. We present an efficient implementation and a hybrid approach that competes with state–of–the–art CDCL solvers.

Restarts

Even though the VSIDS heuristic helps a solver to focus on critical constraints of the formula, a solver may get stuck in a part of the search tree. Therefore, restarts are performed frequently. A restart forces the solver to jump back to decision level zero and start over again. Most solvers keep the set of learnt clauses and do not change the activity values of variables for the next start. However, it is up to the search heuristic to reinitialise activity values or remove some learnt clauses at certain restarts. The benefits of restarts differ for different kinds of formulae. Industrial formulae may often benefit from frequent restarts, whereas, for crafted instances, restarts may often harm the performance of a solver. However, in general, it is hard to predict which restart policy should be applied to particular families of instances [Hua07a].

Static restarts are mostly based on the number of conflicts. For each start of a CDCL search, a maximal number of conflicts *mc* is fixed in advance. This number *mc* may be modified for each restart. Common techniques multiply an initial number of maximal restarts by a constant value after each restart [ES03]. This technique is referred to as geometric restarts. Other solvers add a constant value to *mc* for each restart or do not modify the value at all [Rya04]. A successful static strategy uses the Luby sequence [LSZ93]. It aims to share the available runtime almost equally among different restart strategies. Luby restarts are used and explained in more detail in Section 5.3.

The concept of *dynamic restarts* was introduced by Biere [Bie08a]. The maximal number of conflicts for one start is still fixed in advance. After most starts, the value is increased. However, after some restarts, the value *mc* is reset to an initial value that slightly increases itself. Unlike purely static strategies, a restart may be skipped when the heuristic assesses the solver to be *agile* enough.

The approaches of Sinz and Iser [SI09], and Audemard and Simon [AS09] go even further, and trigger restarts dynamically. A restart is performed when the heuristic considers current progress to be much worse than the average progress. Different criteria can be applied to estimate the progress of the solver, such as the average quality of learnt clauses.

Garbage collection

CDCL solving learns a new clause after each conflict. This is required to assert the implied assignment after backjumping, and it may prevent the solver from searching in areas where no solution can be found. However, an increase in the number of clauses slows down Boolean constraint propagation. Therefore the set of learnt clauses has to be reduced frequently. For this reason, learnt clauses are marked as being redundant and can safely be removed from the set of clauses. In this context, heuristics have to decide the following issues:

- The quality of clauses has to be estimated in order to decide which clauses are worse than others and need to be removed from the clause database. A good solution is to store activities for learnt clauses analogously to the VSIDS heuristic. A clause's activity is increased whenever it is involved in a conflict and all activity values are divided by a constant from time to time. The SAT solver Glucose [AS09, AS12] uses the *literals blocks distance* (LBD) for each learnt clause C^*. The LBD value is defined as the number of different decision levels of the literals in C^*. The smaller a clause's LBD value, the better its estimated quality.

- How many clauses should be kept at garbage collection? Most solvers remove half of the set of learnt clauses and keep the other half. The half to which each clause belongs depends on the estimation of quality, i.e. a clause's activity or its LBD value.
- How often should garbage collection be applied? As for restarts, an initial value for the number of learnt clauses (or conflicts) is fixed. MiniSat sets the initial value to one third of the number of clauses in the formula. The value is increased geometrically after each garbage collection. In contrast, Glucose applies aggressive clause deletion and initialises the total number of learnt clauses to 20,000, or fewer if the formula is very small. The value is incremented by 500 after each garbage collection [AS09].

Other heuristics may revive previously removed learnt clauses [ALMS11]. A similar idea is also applied when an unsatisfiable core of a formula is minimised and several SAT problems are solved consecutively [Nad10].

In the Introduction, an example is used to demonstrate how sensitive a SAT solver may react to small changes in the heuristic. Now that the most common CDCL heuristics have been presented, their modifications can be explained in more detail. Modifications affect the process of percolating an element in the activity heap down from the root. A variable to be percolated down may be exchanged with the child node that has the higher activity value. Three different versions are considered in Figure 1.2. The original MiniSat implementation stops percolating down when the appropriate child node has a smaller or equal activity value. The first modification is *inert* and does not stop percolating down before the child node has a strictly smaller activity value. The second modification keeps the heap *stable*. When the topmost variable is removed from the heap, the deepest variable is moved to the top and is percolated down. At this point, inert percolation is applied to get the variable as deep as possible. All other percolation operations (e.g. reinserting elements at backjumping) remain unchanged.

In general, finding the proper choice of heuristic values is a time–consuming task. The interaction of different heuristics makes things even more difficult. For example, the frequent application of restarts performs much better if the phase saving heuristic is also applied. There may be quite some approaches and solving techniques that have not yet established themselves in state–of–the–art solvers due to a poor choice of heuristics (see also [AS08]).

Chapter 3

Modular Extensions to CDCL

The use of SAT solving for industrial applications and real–world scenarios that are encoded as SAT problems has greatly stimulated the improvement of solving technology. In the early 1990s, the best method to tackle industrial instances and those translated from scheduling problems [KS92] was local search [SKC93]. The idea of combining search and resolution, as introduced by the GRASP algorithm [MSS96], entails a significant improvement for solving real–world SAT problems. The efficient implementation of the GRASP–based procedure [MMZ^{+}01] is commonly referred to as CDCL solving. State–of–the–art solvers for industrial and application SAT instances are based on the CDCL procedure. However, CDCL has constantly been improved over the last years. Most of the effective improvements have been achieved by small changes compared to the first CDCL solver Chaff [MMZ^{+}01], such as:

- Activities for variables instead of activities for literals [ES03],
- A heap data structure for branching decisions [ES03],
- Different kinds of restarts [LSZ93, Bie08b, Hua07a],
- Phase saving of variables [PD07a],
- Estimation of a clause's quality [AS09],
- Cache–conscious data structures [CHS09].

In this chapter, two approaches for extending the classical CDCL solving are presented. In the first (Section 3.1), a simple but effective improvement to the two watched literals scheme is described. The improvement is based on an evident compression of the data structure to store clauses within a SAT solver. Moreover, the presented data structure allows a faster detection of the second watched literal, when a clause is accessed during unit propagation. Both arguments improve the speed of BCP and were crucial for the performance of our solver MoUsSaka in the MUS competition 2011 [Sat11].

In the second part of the chapter, an extension to the so–called simplification technique of asymmetric branching is introduced. Since the implementation of PrecoSAT [Bie09b], several solvers frequently simplify the set of clauses in between solving. This so–called inprocessing makes simplification techniques even more important and motivates further improvement of simplification, regarding both speed and quality. However, the tuning of a SAT solver can become more difficult, since the number of parameters for the proper application of inprocessing increases significantly. Section 3.2 describes an approach to improve the quality of asymmetric branching and in Section 3.3, the algorithm is extended by hyper–binary resolution [Bac02a, Bie09a]. The presented extensions to CDCL are evaluated in Section 3.4 and are briefly summarised in Section 3.5.

3.1 Speeding up Unit Propagation

Any clause of a SAT formula in CNF is basically a static set of literals. However, in practice, the literals of a clause have different roles that change permanently during the solving process. Most state–of–the–art SAT solvers implement the two watched literals scheme [MMZ$^+$01] following the method suggested by Van Gelder [Gel02]. The two watched literals of a clause are always placed in the first two positions of the array of literals. This idea allows for a clear and simple maintenance of the watched literals scheme without any extra information within a clause. As mentioned in Section 2.2.2, the literals have to be reordered whenever one of the watched literals is exchanged. In this section, we present an enhancement of the two watched literals scheme to speed up unit propagation. The improvement is based on a reduction of required memory and, in particular, a reduction of memory accesses during BCP.

The concept has proven its usefulness in our MUS solver, briefly described in Section 7.5, where the speed of unit propagation is seminal due to thousands of consecutive calls of mostly easy SAT formulae. A slight modification of this concept is also beneficial for the implementation of physical clause sharing in multithreaded SAT solving (see Chapter 6 or [KK11c, KK11b]). To handle literals more easily, let the function $ID : \mathcal{L} \mapsto \mathbb{N}$ map any literal to an integer value and let its inverse operation ID^{-1} map an integer back to a literal. Without loss of generality we assume the image set of ID to be a consecutive set of numbers in the range $[0, 2 * |\mathcal{V}|)$. The improvement is based on the following observation:

Property 1. *Whenever a clause $C \in \mathcal{F}$ is addressed by the basic CDCL algorithm, at least one of the two watched literals of C is known.*

In the basic CDCL algorithm (see Algorithm 2.2), there are three main functions where clauses are actually touched:

1. Boolean constraint propagation (BCP),
2. Cleaning the set of clauses, and
3. Conflict analysis.

1. During BCP, the literals that became false by the current partial assignment τ are examined. Their watcher lists are traversed to check for clauses that are unit or falsified by the assignment τ. Hence, when traversing the watcher list of literal λ_i, all clauses that are accessed have λ_i as one of their watched literals.

2. Cleaning the set of clauses can also be done by traversing the watcher lists of all literals. One possible realisation is sketched in Algorithm 3.1. Analogously to BCP, when cleaning the clauses watched by a literal λ_i, each clause that is accessed has λ_i as one of its watched literals.

3. In conflict analysis, the implication graph [MSS99] is traversed backwards. Any clause C that becomes unit by a partial assignment τ during BCP causes the remaining literal λ_i to be assigned to *true*. In doing so, C is stored as an asserting clause (or reason) for the assignment of λ_i. Moreover, λ_i is one of the two watched literals of C when C is stored as the asserting clause for the assignment. If C is traversed during conflict analysis, it will only be accessed as an asserting clause for the assignment λ_i. Thus one watched literal of C is known. With Property 1, the following corollary can be stated.

Corollary 3.1.1. *The information of the two watched literals λ_p and λ_q of a clause C_0 can be saved by one value $\mathcal{X}(C_0) := ID(\lambda_p)$ `XOR` $ID(\lambda_q)$.*

Algorithm 3.1 iterates over all literals in increasing order. In doing so, each clause is accessed twice. The actual test to check whether a clause will be kept or deleted is done when the clause is touched for the first time (line 12). If the application of some criteria (LBD value, activity, clause size) by the procedure `testRemoveCls` decides to remove a particular clause C^*, the appropriate `XOR` value is set to zero. This marks the clause for the traversal of the second watched literal. The use of zero is based on the fact that an `XOR` operation where one operand equals zero will always return the other operand (lines 9 and 13).

Algorithm 3.1: Garbage collection with the XOR–watchers

```
   Require Formula $\mathcal{F}$;
   Function garbageCollection ($\mathcal{F}$)
       for $i \leftarrow 0$ to $2 * |\mathcal{V}| - 1$ do
           $\lambda_i \leftarrow ID^{-1}(i)$
           $W_{\lambda_i} \leftarrow$ watchedOf($\lambda_i$) | clauses with watched literal $\lambda_i$
           foreach $C^* \in W_{\lambda_i}$ do
               $k \leftarrow \mathcal{X}(C^*)$ xor $i$          | XOR value $\mathcal{X}(C^*)$ for $C^*$
               $r \leftarrow false$                       | remove clause or not
9              if $k = i$ then
                   destruct clause           | $C^*$ is marked ($\mathcal{X}(C^*) = 0$)
                   $r \leftarrow true$
12             else if $i < k$ and testRemoveCls ($C^*$) then
13                 $\mathcal{X}(C^*) \leftarrow 0$        | mark clause for second watcher
                   $r \leftarrow true$
           if $r = true$ then remove reference to $C^*$ from $W_{\lambda_i}$
```

In the process of accessing a clause C, one watched literal λ_p is always known. With Corollary 3.1.1, the other watched literal is given by $\lambda_q = ID^{-1}(\mathcal{X}(C)\ \texttt{XOR}\ ID(\lambda_p))$. This idea is similar to the concept of static graphs [NZ02], where edges are only accessed via one of their incident vertices. Thus, the two watched literals of a clause C can be replaced by one value $\mathcal{X}(C)$ without any loss of information. Our solvers SApperloT and MoUsSaka use this technique to represent any clause C with $|C| > 2$ literals by $(|C| - 1)\times$ `sizeof`(literal) bytes. One obvious benefit of this idea is the total reduction of required memory. Having up to one or even more than one million clauses within an industrial SAT instance means that megabytes of main memory may be saved. In conflict–driven SAT solving, the number of clauses usually increases during the solving process, so a reduction in memory used per clause clearly reveals its effectiveness. Permanent access to many different clauses during unit propagation constitutes random access of non–local data. Non–local access may be particularly unfavourable when present–day computer architecture is based on local caching schemes. The reduction of clause sizes can help to improve the cache performance of a SAT solver.

Figure 3.1 depicts the architecture of clauses that apply Corollary 3.1.1. Each clause is referenced by its two watched literals. Therefore, the information about the watched literals of the clause can be compressed to the XOR value of these literals. To avoid storing the absolute size of a clause, each literal field holds one bit to indicate if the current literal is the last literal of a clause.

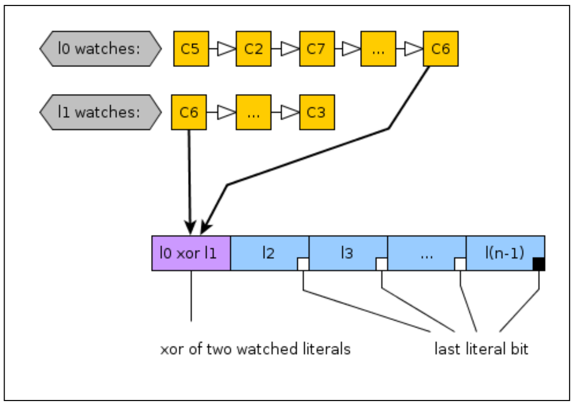


Figure 3.1: `XOR`–implementation of watched literals.

The application of Corollary 3.1.1 for parallel SAT solvers will be discussed in Chapter 6. When sharing the literals of one clause among several solving threads, it has to be borne in mind that the watched literals may differ in each solving thread. The realisation of the two watched literals scheme by this `XOR`–watchers approach can be incorporated to any CDCL solver. However, it requires the modification of some internal functionality.

The simplification algorithm presented in the next section exhibits the characteristic of a module. It can be applied in between different executions of the search procedure of a SAT solver without the need to change the internals of the solver in use.

3.2 The Strength of Asymmetric Branching

SAT instances stemming from real–world applications (e.g. industrial instances) are often transformed into CNF by using the so–called Tseitin transformation [Tse68, BHvMW09]. This approach introduces new variables that are set to be equivalent to a subtree of the original Boolean formula. When chosen properly, the generated CNF formula is significantly smaller than repeatedly reusing the entire subformula. Moreover, application instances often contain several redundant constraints that may be added on purpose or as a consequence of the applied transformation. For this reason, it can often be beneficial for a solver's performance to modify a formula before the actual solving process. With the aim of making a modified formula easier to solve, the modification is referred to as simplification, even though it is not always obvious which modifications are actually beneficial for the solver being used.

Simplification of SAT formulae is now applied by most state–of–the–art SAT solvers, to a certain extent. Since the implementation of inprocessing in PrecoSAT [Bie09b], simplification techniques have experienced a renaissance after the success of MiniSat [ES03] had boosted the engineering of clear, small solvers, as solver names like PicoSat [Bie08b] and TiniSat [Hua07b] indicate. Proper simplification has mainly been applied in preprocessing [SP04, EB05]. Applying simplification techniques frequently in between CDCL searches allows for the additional use of learnt information. For many SAT instances, the performance and power of the solver can clearly be increased, as the success of PrecoSAT and Lingeling [Bie11] shows. This motivates the investigation and improvement of simplification techniques. However, choosing suitable configurations for solving, inprocessing and the interaction between both is a complex task. Furthermore, in parallel portfolio solvers, simplification can be applied concurrently, as described in Chapter 6.

In this section, an algorithm is presented that improves the quality of asymmetric branching, a simple but yet efficient simplification technique. To this end, relevant simplification techniques are briefly explained in Section 3.2.1. Subsequently, different variations of asymmetric branching are shown. A major drawback of asymmetric branching motivates an extension of the original algorithm and places greater demands on its quality. This issue is addressed in Section 3.2.2 and its complexity is analysed in Section 3.2.3. In Section 3.3, the algorithm is further extended.

3.2.1 Related work

Several simplification techniques for CNF formulae have been proposed and are applied successfully in practice. This section only presents the techniques that are relevant to our extension of asymmetric branching. We refer the reader to the original work for different techniques (e.g. [Bra01, BW03, SP04, Bra04, EB05, DDD+05, FGMS07, PHS08, ABH+08, JBH10, HJB11]).

Subsumption and self–subsuming resolution

Some constraints within an industrial SAT instance are obviously redundant. If the literals of a clause C_b constitute a subset of the literals of another clause C_p, then C_b is said to subsume C_p. Clearly, any model of the formula has to satisfy the constraint $C_b \subseteq C_p$ and will thus also satisfy the weaker constraint C_p. Hence any subsumed clause can be removed from the clause database. The number of subsumed clauses often increases after some variables are eliminated by existential quantification. And, *vice versa*, the removal of redundant clauses may allow for existential quantification of a variable without increasing the total number of literals in the formula.

This kind of simplification was introduced by Subbarayan and Pradhan [SP04] and is implemented efficiently in the SatELite preprocessor [EB05] and follow–up implementations [Zie10].

Related to the concept of subsumption is the idea of self–subsuming resolution. Consider the following two clauses that have one clashing literal, i.e. a variable with different polarities: $C_1 = (A \vee \lambda_x)$ and $C_2 = (B \vee \overline{\lambda_x})$. By resolution, the clause $C_x = (A \vee B)$ can be deduced. Now consider the special case that all literals of B are also contained in A, such that $B \subseteq A$. In that case, the literals contained in C_x are exactly those of A, and thus, C_x subsumes C_1. Note that C_x does not need to be contained in the clause database. The detection of self–subsuming resolution allows for the removal of literal λ_x from clause C_1 to implicitly replace C_1 by C_x.

Even though both techniques can be implemented efficiently, it requires the use of complete occurrence lists for literals or variables [EB05]. For a variable, an occurrence list contains all clauses where that variable occurs in. The implementation of the two watched literals scheme makes this kind of information obsolete for pure CDCL solvers. However, there are some approaches for detecting some cases of subsumption during CDCL search and conflict analysis [DDD+05, HS09, HJS10].

Simplification using propagation

Most simplification techniques, such as those presented above, require some extra data structures and the implementation of advanced procedures. In particular, the need for occurrence lists of variables entails the maintenance of an additional data structure. On the contrary, the application of asymmetric branching does not require a CDCL solver to offer additional features. Asymmetric branching considers a given clause, which will be referred to as $C^{\#}$ throughout the chapter. For all literals of $C^{\#}$, the opposite values are assigned and propagated. Depending on the outcome, $C^{\#}$ may be tightened or detected to be redundant.

In Algorithm 3.2 the idea is outlined. All literals that can be removed from $C^{\#}$ are collected in D (line 4) and the clause is tightened when the function returns (line 10). Each literal of $\lambda_p = C^{\#}[i]$ is processed obeying the given order (line 6). If λ_p is already assigned to its opposite value by the current partial assignment τ (line 7), then λ_p can be removed from $C^{\#}$. This can intuitively be explained by the argument below.

Let us assume there is a model τ that satisfies $C^{\#}$ and the model contains $\overline{C^{\#}[i]} = \overline{\lambda_p}$. Hence $C^{\#}$ is satisfied by a literal $\lambda_q \in \tau$ with $\lambda_q \in \{C^{\#} \setminus \lambda_p\}$. If, on the other hand the model contains λ_p, then at least one

literal $\lambda_q = C^\#[j], j < i$ is also in the model, because the assignment of $\overline{C^\#[0]} \wedge \overline{C^\#[1]} \wedge \ldots \wedge \overline{C^\#[i-1]}$ implies the assignment of the literal $\overline{\lambda_p}$, which contradicts the assumption. Thus, in both cases, $(C^\# \setminus \{\lambda_p\})$ is satisfied by λ_q.
A more formal explanation uses the asserting clause C_p for the assignment of literal $\overline{\lambda_p}$. As in lemma generation, the reason C_p can be modified to an asserting clause $C'_p = (\overline{\lambda_p} \vee K)$ that only contains literal $\overline{\lambda_p}$ and some (negated) decisions. This is done by recursively using the asserting clauses for resolution for all implied literals (see Section 2.2.4 and [MSS96]). Clearly, K is a subset of $C^\# \setminus \{\lambda_p\}$, because all decisions in asymmetric branching are chosen from $C^\#$ with opposite polarity (for $\lambda_q \in C^\#$, the decision is $\overline{\lambda_q}$). With this, resolution of clauses $C^\#$ and C'_p results in clause $(C^\# \setminus \{\lambda_p\})$, and thus λ_p can be removed from $C^\#$ by self–subsuming resolution.

Algorithm 3.2: Asymmetric branching

```
Require Clause $C^\#$ to be tightened
Return If conflict, and a set of literals that can be removed from $C^\#$
Function asymBranch ($C^\#$)
4     $D \leftarrow \emptyset$
      for $i \leftarrow 0$ to $|C^\#| - 1$ do
6         $\lambda_p \leftarrow C^\#[i]$
7         if $\overline{\lambda_p} \in \tau$ then $D \leftarrow D \cup \{\lambda_p\}$
          else if BCP($\overline{\lambda_p}$) is conflicting then
9             return $< conflict, D \cup \{C^\#[j] : i < j < |C^\#|\} >$
10    return $< ok, D >$
```

Asymmetric branching may also produce a conflicting assignment, as in line 9 of Algorithm 3.2. If a conflict arises, three different actions may be taken:

(i) As in lemma generation, the conflicting clause C_c can be transformed into a clause C'_c that entirely consists of some negated decisions. Hence C'_c subsumes $C^\#$ and allows for the reduction of $C^\#$ to the literals of C'_c.

(ii) Without analysing the conflict and generating the lemma C'_c it is clear that a conflict (arising in line 9) can only depend on the decisions that have already been made and thus the clause $C''_c = (C^\#[0] \vee \ldots \vee C^\#[i])$ can be learnt directly. $C^\#$ can be reduced to the literals of C''_c.

(iii) $C^\#$ is redundant with the current set of constraints since other clauses prohibit $C^\#$ from being unsatisfied by any model of the formula. $C^\#$ can safely be removed from the formula.

Consider the following example: $C^{\#} = (\lambda_1 \vee \lambda_2 \vee \lambda_3 \vee \lambda_4 \vee \lambda_5), C_1 = (\lambda_1 \vee \lambda_2 \vee \overline{\lambda_6}), C_2 = (\lambda_2 \vee \lambda_4 \vee \overline{\lambda_7}), C_3 = (\lambda_1 \vee \lambda_4 \vee \overline{\lambda_8})$ and $C_4 = (\lambda_6 \vee \lambda_7 \vee \lambda_8)$. When the opposite literals of $C^{\#}$ are propagated as ordered in the clause, the assignments of $\overline{\lambda_1}$ and $\overline{\lambda_2}$ imply the assignment $\overline{\lambda_6}$ due to clause C_1. After the assignment of $\overline{\lambda_3}$ and $\overline{\lambda_4}$, two more assignments, $\overline{\lambda_7}$ and $\overline{\lambda_8}$, are implied due to clauses C_2 and C_3. At this point, clause C_4 is detected as conflicting with the current partial assignment. Now $C^{\#}$ may be removed (option iii) from the clause set, or $C^{\#}$ may be reduced to $(\lambda_1 \vee \lambda_2 \vee \lambda_3 \vee \lambda_4)$, since the decisions that negate the first four literals of $C^{\#}$ are sufficient to cause a conflict (option ii). When analysing the conflict, the clause $C'_c = (\lambda_1 \vee \lambda_2 \vee \lambda_4)$ is created and thus $C^{\#}$ can be reduced to C'_c (option i).

Now consider the example above with the modified clause $C_4 = (\overline{\lambda_5} \vee \lambda_6 \vee \lambda_7 \vee \lambda_8)$. As above, the propagation of decisions $\overline{\lambda_1}$, $\overline{\lambda_2}$, $\overline{\lambda_3}$ and $\overline{\lambda_4}$ implies the assignments $\overline{\lambda_6}, \overline{\lambda_7}$, $\overline{\lambda_8}$ and $\overline{\lambda_5}$ due to clauses C_1, C_2, C_3 and C_4. In the last iteration of Algorithm 3.2, the condition in line 7 is fulfilled since $\overline{\lambda_5}$ has already been assigned. The asserting clause of assignment $\overline{\lambda_5}$, clause C_4, can be transformed to $C'_4 = \overline{\lambda_5} \vee \lambda_1 \vee \lambda_2 \vee \lambda_4$ by recursive resolution with asserting clauses. The resolution of the clauses $C^{\#}$ and C'_4 deduces clause $C' = (\lambda_1 \vee \lambda_2 \vee \lambda_3 \vee \lambda_4)$, which subsumes $C^{\#}$. Hence literal λ_5 can be removed from $C^{\#}$ by self–subsuming resolution with C'_4.

The technique of asymmetric branching has been studied under different names and in different SAT–related communities. Some approaches for using Boolean constraint propagation to simplify an instance have been proposed by Le Berre [Ber01]. For a chosen variable ν_i, both assignments $\nu_i \leftarrow true$ and $\nu_i \leftarrow false$ are propagated. By analysing the consequences of both assignments (e.g. the intersection of $\{\lambda_i \xrightarrow{\text{UP}}\} \cap \{\overline{\lambda_i} \xrightarrow{\text{UP}}\}$), new unit clauses may be deduced. Analogously, the idea is applied to deduce binary clauses when two variables are chosen. This idea is also used as a branching heuristic by Darras *et al.* [DDD$^+$05].

In the preprocessing of MiniSat2.0 [ES03, EB05, ES12] asymmetric branching is applied in a basic manner. However, the application of this technique is switched off by default. Han and Somenzi studied asymmetric branching under the name distillation in a more general context [HS07]. One major issue is the combination of distillation for several clauses to reduce the overhead of unit propagation. For that purpose, clauses are stored in a trie data structure [AHU83]. The solvers PrecoSAT and Lingeling [Bie11] also use the idea of distillation to a certain extent. Piette *et al.* apply asymmetric branching within the preprocessor ReVivAl [PHS08], where the technique is referred to as vivification of clauses. The work is based on a study of Fourdrinoy *et al.* [FGMS07], where all clauses are removed for which asymmetric branching causes a conflict (see option iii above). The more general concept of vivification used in ReVivAl [PHS08] considers the drawback of removing redundant clauses for CDCL solvers. When a conflict arises during vivification of some clause $C^{\#}$, a lemma is created by using the FUIP scheme [ZMMM01]. If the generated lemma subsumes $C^{\#}$, the clause is tightened accordingly. Otherwise $C^{\#}$ is tightened according to the decisions made (see option ii above).

The improved algorithm for asymmetric branching presented in Section 3.2.2 considers alternative reasons for unit propagation. This is related to the concept of inverse arcs presented by Audemard *et al.* [ABH+08] to improve backjumping after conflict analysis in CDCL solving. Our approach uses the idea of alternative reasons in a more general way.

3.2.2 Disregarding the order of propagation

The common applications of asymmetric branching (distillation and vivification) have a major drawback. The detection of self–subsuming resolution strongly depends on the order in which literals of the input clause $C^{\#}$ are propagated. Furthermore, the generation of conflict clauses also depends on the order of decisions. Consider the following example with three clauses: $C^{\#} = (\lambda_1 \vee \lambda_2 \vee \lambda_3 \vee \lambda_4)$ is the input clause that ought to be tightened. There are also the clauses $C_1 = (\overline{\lambda_1} \vee \overline{\lambda_5})$ and $C_2 = (\lambda_2 \vee \lambda_4 \vee \lambda_5)$. Assume that the literals of $C^{\#}$ are processed as ordered within $C^{\#}$. In particular, the assignments $\overline{\lambda_1}$, $\overline{\lambda_2}$, $\overline{\lambda_3}$ and $\overline{\lambda_4}$ are propagated. Due to C_2, unit propagation will also assign literal λ_5. With C_1 being fulfilled, $C^{\#}$ will not be tightened. However, if the literals of $C^{\#}$ are processed in reverse order, the assignments $\overline{\lambda_4}$ and $\overline{\lambda_2}$ imply two more assignments within unit propagation: λ_5 by C_2 and subsequently $\overline{\lambda_1}$ by C_1. As described in the previous section, asymmetric branching detects that clause $C' = (\overline{\lambda_1} \vee \lambda_2 \vee \lambda_4)$ can be generated by resolution. The resolvent of C' and $C^{\#}$ over variable ν_1 subsumes $C^{\#}$, and thus literal λ_1 can be removed from $C^{\#}$. In the remainder of this work, the extension of asymmetric branching that is independent of the order of propagation is referred to as AB^*.

A crucial point for the entire computation is the fact that in asymmetric branching, the number of decisions is bounded in advance. The proposed algorithm to disregard the order of propagation in asymmetric branching consists of three major parts. Algorithm 3.3 outlines the first part, where literals are propagated in some order and additional information is collected for the subsequent parts. In the second part, listed in Algorithms 3.4 and 3.5, information is filtered and prepared for the last part. Algorithm 3.6 finally computes the consequences of asymmetric branching for any order of propagation. Algorithms 3.7 and 3.8 combine the different parts. Below, all algorithms are explained in detail.

Propagation with a bounded number of decisions

At the beginning (from line 2) of Algorithm 3.3, some global variables are initialised. τ holds the partial assignments. `Rsns`(ν) keeps the reason for the assignment of variable ν and, moreover, all clauses that could be the reason for the assignment of ν, if a different propagation order was chosen. The set `Confl` keeps all conflicting clauses during propagation.

Algorithm 3.3: BCP for asymmetric branching, disregarding the order

```
   **Require** Clause $C^{\#} \in \mathcal{F}$ to be tightened
2  $\tau \leftarrow \emptyset$                                   | partial assignment
   Rsns($\nu$) $\leftarrow \emptyset \; \forall \; \nu \in \mathcal{V}$           | alternative reason clauses
   Confl $\leftarrow \emptyset$                                | conflicting clauses
   **Function** asymBCP ($C^{\#}$)
       $Q \leftarrow \emptyset$                              | literal's queue to propagate
7      **for** $i \leftarrow 0$ **to** $|C^{\#}| - 1$ **do**
           $\lambda_q \leftarrow C^{\#}[i]$
           $Q$.enqueue($\overline{\lambda_q}$)                  | propagate new assignment
           $\tau \leftarrow \tau \cup \overline{\lambda_q}$                     | assign opposite literal
11         RBS($\nu_q$) $\leftarrow 2^i$                      | init bitset of variable

       **while** $Q \neq \emptyset$ **do**
14         $\lambda_q \leftarrow Q$.dequeue()              | next literal to propagate
           $W_{\overline{q}} \leftarrow$ watchedOf($\overline{\lambda_q}$)         | clauses with watched $\overline{\lambda_q}$
           **foreach** $C^* \in W_{\overline{q}} \setminus C^{\#}$ **do**
17             $\lambda_p \leftarrow$ otherWatched($C^*, \overline{\lambda_q}$)
               **if** $\lambda_p \in \tau$ **then**
19                 Rsns($\nu_p$) $\leftarrow$ Rsns($\nu_p$) $\cup C^*$
               **else if** $\exists \; \lambda_k \in \mathtt{U}_\tau(C^*) \cup \mathtt{T}_\tau(C^*) \setminus \{\lambda_p\}$ **then**
21                 $W_{\overline{q}} \leftarrow W_{\overline{q}} \setminus C^*$                 | link new watched
                   $W_k \leftarrow W_k \cup C^*$
               **else if** $\overline{\lambda_p} \notin \tau$ **then**
24                 $Q$.enqueue($\lambda_p$)                 | unit propagation
                   $\tau \leftarrow \tau \cup \lambda_p$
                   Rsns($\nu_p$) $\leftarrow C^*$
27                 RBS($\nu_p$) $\leftarrow \bigcup_{\lambda_k \in \{C^* \setminus \lambda_p\}}$RBS($\nu_k$)
               **else if** $C^* \notin$ Confl **then**
29                 $r \leftarrow \bigcup_{\lambda_k \in C^*}$RBS($\nu_k$)
                   Confl $\leftarrow$ Confl $\cup < C^*, r >$          | keep conflict
```

The first loop between lines 7 and 11 assigns the opposite value for all literals of the input clause $C^{\#}$. In addition to reason clauses, each assigned variable ν stores a bitset $\texttt{RBS}(\nu)$ (Reasons Bit Set) that creates an efficient and compressed way to trace back the reasons to the input clause $C^{\#}$. The application of bitsets uses the fact that, unlike common search in CDCL, the number of decisions is bounded by $|C^{\#}|$ right from the beginning. In line 11, the assignment of each variable is related to one particular bit. All subsequent assignments will be a consequence of some of these initial decisions and their bitsets will thus be the union of some initial bitsets.

Starting from line 14, all implied assignments are propagated in the usual way. The main loop inspects each clause C^* within the list $W_{\overline{q}}$, that keeps all clauses where literal $\overline{\lambda_q}$ is one of the two watched literals. The second watched literal is referred to as λ_p in each inner loop pass (line 17).

If the other watched literal λ_p is *true*, C^* does not require further attention for this propagation. However, in line 19, C^* is kept as a possible alternative reason for the assignment $\lambda_p \in \tau$. C^* does not necessarily have to be unit under τ at this stage of propagation, though, it may become unit under the possibly growing τ and must thus not be missed. To this end, Algorithm 3.4 will inspect C^* again when propagation is complete. This constitutes a major difference to the application of inverse arcs in [ABH$^+$08].

If there is another free literal λ_k (i.e. unassigned or *true*) that is different from λ_p, then λ_k can become the new watched literal of C^* together with λ_p (line 21), and C^* is processed. If this is not possible, C^* is either unit or conflicting. If C^* is unit under τ, then λ_p is not yet assigned and unit propagation forces λ_p to be assigned (line 24) with the reason clause C^*. In addition to common BCP, the reason for the assignment of λ_p is also expressed by the bitset $\texttt{RBS}(\nu_p)$ in line 27. It is initialised as the union of the `RBS` values of all other literals in C^* (bitwise `OR`). With this, the i-th bit in $\texttt{RBS}(\nu_p)$ is set if the initial assignment $\overline{C\#[i]}$ contributes to the assignment of λ_p in any way.

If none of these cases apply, C^* is a conflicting clause where all literals are *false*. Analogously to line 27 when C^* is unit, in line 29, the bitset r indicates the reason why C^* is conflicting. Unlike an assignment reason, a conflict reason has to take the union of the `RBS` values of all literals in C^*. The set `Confl` stores a tuple of the conflicting clause together with the conflict reason. Note that each conflicting clause is touched twice, once for each watched literal, since propagation does not terminate when a conflict arises.

Filter relevant information

During the propagation of assignments, as presented in Algorithm 3.3, the process collects clauses that may constitute an alternative reason for an assignment within τ. However, during propagation, it cannot be decided whether or not a clause will be unit under the final partial assignment τ. To this end, these clauses are inspected after propagation. Furthermore, some assignments may have been made, that are neither relevant to any conflict nor are they relevant to any alternative reason for a variable of $C^{\#}$.

Algorithm 3.4 is basically a depth–first search procedure that traverses the graph of reason clauses backwards. Unlike the implication graph, as described in Section 2.2.4, alternative reason clauses are also considered. A set of already handled (and hence important) variables and a set of important clauses are given as parameters. A tuple of the extended sets is returned. Traversal is started from a given variable ν_p if it has not already been handled (line 6). In line 8, each possible reason C for the assignment of variable ν_p is inspected. At first, the process tests whether C is unit under the partial assignment τ (line 9) – if all literals of C but one are $false$. If C is not a valid alternative reason for ν_p, it is removed from $\texttt{Rsns}(\nu_p)$ and the next clause is processed. If C is a valid reason (asserting the assignment of ν_p), it is added to the set R_C in line 11.

Algorithm 3.4: Compute relevant variables and valid reasons

```
    Require Preceding execution of asymBCP ($C^{\#}$), start variable $\nu_p$
    Return Set of relevant variables and relevant clauses
    $R_V \leftarrow \emptyset$                                  | relevant variables
    $R_C \leftarrow \emptyset$                                  | relevant clause reasons
    Function getRelevant ($\nu_p, R_V, R_C$)
6       if $\nu_p \in R_V$ then return $< R_V, R_C >$
        $R_V \leftarrow R_V \cup \nu_p$
8       forall $C \in \texttt{Rsns}(\nu_p)$ do
9           if $|\texttt{F}_\tau(C)| \neq |C| - 1$ then
                Rsns($\nu_p$) $\leftarrow$ Rsns($\nu_p$) $\setminus C$; continue
11          $R_C \leftarrow R_C \cup C$
12          $b \leftarrow$ initBitset ()                    | reason by this clause
            forall $\nu_q \neq \nu_p : \lambda_q \in C$ do
14              $b \leftarrow$ bitsetAddVar ($\nu_q, b$)
15              $< R_V, R_C > \leftarrow < R_V, R_C > \cup$ getRelevant ($\nu_q, R_V, R_C$)
16          bitsetFinish ($\nu_p, b$)
        return $< R_V, R_C >$
```

Algorithm 3.5: Compute new bitset for a reason clause

```
Function initBitset ()
    return 0
Function bitsetAddVar ($\nu_q, r$)
    return $r \cup$ RBS($\nu_q$)
Function bitsetFinish ($\nu_p, r$)
    RBS($\nu_p$) $\leftarrow$ RBS($\nu_p$) $\cap\ r$
```

Some functionality of the remaining algorithm is outsourced to Algorithm 3.5. This may appear over–the–top at first sight, but in doing so, the relevant functionality can be reused and modified in Section 3.3. For the validated alternative reason clause C, an additional bitset b is computed, which expresses this reason in a compressed way (lines 12 and 14). This is analogous to Algorithm 3.3 in line 27: b is the union of all `RBS` values of the falsified literals of C. In addition, in line 15, a recursive function call for each such variable of C is made.

A crucial part of the whole computation lies in calling the function `bitsetFinish` in line 16. The `RBS` value of variable ν_p is intersected (bitwise `AND`) with the computed value b. Thus some bits that were set before may be cleared. The intuition is that the reasons bitset of a variable ν_p is used to indicate those assignments of the variables of $C^{\#}$ that are indispensable for the assignment of ν_p. If a variable has alternative reasons, the intersection of its `RBS` value only keeps those bits that mark a dependency on the assignments of the variables of $C^{\#}$ that exist in all alternative reason clauses.

Consider a simple example with three clauses $C^{\#} = (\lambda_1 \vee \lambda_2 \vee \lambda_3 \vee \lambda_4)$, $C_1 = (\lambda_1 \vee \lambda_2 \vee \lambda_5)$ and $C_2 = (\lambda_2 \vee \lambda_4 \vee \lambda_5)$. When the opposite literals of $C^{\#}$ are propagated in order, the `RBS` value of variable ν_5 is initialised to $2^0 + 2^1 = \langle 0011 \rangle$ in line 27 of Algorithm 3.3 due to C_1. The alternative reason by C_2 depends on the assignments $\overline{\lambda_2}$ and $\overline{\lambda_4}$, and generates the value $2^1 + 2^3 = \langle 1010 \rangle$ in lines 12 and 14 of Algorithm 3.4. The intersection operation invoked in line 16 sets $\texttt{RBS}(\nu_5) = \langle 0010 \rangle$, since the assignments $\overline{\lambda_1}$ and $\overline{\lambda_4}$ are not required simultaneously for the assignment of λ_5. However, a single computation for an `RBS` value of a variable may not be sufficient to get rid of all assignments of $C^{\#}$ that are not required simultaneously. This issue will be tackled below, as shown in Algorithm 3.6.

Mandatory assignments of $C^{\#}$

The algorithm presented above creates a set of relevant variables R_V and a set of relevant clauses R_C. More precisely, for any variable $\nu_p \in R_V$, all clauses that can be a reason for the assignment of ν_p are contained in R_C. In particular, `Rsns`(ν_p) keeps all these clauses that have all but one literal assigned $false$ by the partial assignment τ. The sets `Rsns`(ν_p) and `Rsns`(ν_q) are disjoint for any pair of variables $\nu_p, \nu_q \in R_V, \nu_p \neq \nu_q$. As in lemma generation of CDCL solvers, any asserting clause for the assignment λ_p allows for a resolution operation with clauses containing literal $\overline{\lambda_p}$ [MSS96]. In CDCL learning schemes, this fact is used to resolve the literals of a conflicting clause until only one assignment of the current decision level is left (FUIP). However, this procedure can be continued until a conflicting clause only contains decision variables [MSS96] (see Algorithm 2.3).

The example above shows that an assignment of a variable $\nu_p \in R_V$ does not generally require all initial assignments of the opposite literals of $C^{\#}$. In terms of lemma generation, this is because using alternative reason clauses for an assignment may generate different lemmas.

For any implied assignment λ_q consider all reason clauses $\mathcal{F}_q \subseteq R_C$. Let $\mathcal{F}'_q$ contain all clauses that can be generated, analogously to lemma generation, by using reason clauses from R_C, such that each clause in $\mathcal{F}'_q$ contains literal λ_q and decision variables only. Let λ_p be a literal of $C^{\#}$. We call the initial decision $\overline{\lambda_p}$ mandatory for the assignment λ_q if every clause in $\mathcal{F}'_q$ contains literal λ_p.

Algorithm 3.6 pursues the idea of differentiating the initial decisions between those that are mandatory and those that are not simultaneously required for a particular variable assignment. This is realised by the use of `RBS` bitsets, as already mentioned above. However, a crucial point is that reducing the set of mandatory assignments for one variable may reduce the set for further variables. Algorithm 3.6 reduces the set of mandatory assignments until a fixpoint.

Besides the sets R_V and R_C, Algorithm 3.6 takes the bitset rem as a parameter to indicate the literals of $C^{\#}$ that are of interest for this call to the function `compMandatory`. Let $M^{rem} \subseteq C^{\#}$ be the literals indicated by rem. For any variable $\nu_p \in R_V$, let $M_p^{rem} \subseteq M^{rem}$ be the set of assignments of literals in M^{rem} that are mandatory for the assignment of variable ν_p. The function `compMandatory` minimises these mandatory assignments for each variable in R_V with respect to M^{rem}. When the function returns, the bitset `RBS`$(\nu_p) \cap rem$ indicates the set M_p^{rem} for each variable ν_p in R_V.

In line 6, the set P is initialised to hold those variables whose relevant `RBS` value does not contain the entire set rem. The queue Q of clauses to be handled is initialised with all clauses that contain at least one variable of P (line 7). The while loop (line 8) continues as long as Q contains any clause C. Clause C contains exactly one literal λ_a that is *true* by the partial assignment τ (line 10). Let *rel* be the relevant bits of its current `RBS` value (line 11). If all relevant bits are cleared in line 12, variable ν_a does not contain any mandatory literals of M; `RBS`(ν_a) is thus already minimal for the current run. Between lines 13 and 16, the `RBS` value for variable ν_a is updated as in Algorithm 3.4. Due to the intersection of bitsets within the function `bitsetFinish` (Algorithm 3.5), invoked in line 16, the `RBS` value can only be a subset of its previous value. If there is any update of the `RBS` value regarding the relevant bits (line 17), then ν_a may cause an update for other clauses. This may apply for any reason clause that contains literal $\overline{\lambda_a}$. Thus in line 18, all these clauses are enqueued into Q.

Algorithm 3.6: Compute mandatory literals

Require Bitset $0 < rem < 2^{|C^\#|}$, and preceding executions of: `asymBCP` $(C^\#)$ and $< R_V, R_C > \leftarrow$ `getRelevant` (ν, R_V, R_C)
Result Bitset `RBS`$(\nu_p) \cap rem$ indicates the mandatory assignments M_p^{rem} for each variable $\nu_p \in R_V$.

```
Function compMandatory ($rem, R_V, R_C$)
6       $P \leftarrow \{\nu_p \in R_V : (\texttt{RBS}(\nu_p) \cap rem) \neq rem\}$          | start vars
7       $Q \leftarrow \{C \in R_C : \exists\ \lambda_p \in C, \nu_p \in P\}$          | clauses queue
8       while $Q \neq \emptyset$ do
            $C \leftarrow Q$.dequeue()
10          $\lambda_a \in \texttt{T}_\tau(C)$          | $C \in \texttt{Rsns}(\nu_a)$ asserting for $\lambda_a$
11          $rel \leftarrow \texttt{RBS}(\nu_a) \cap rem$          | relevant bits
12          if $rel = 0$ then continue
13          $b \leftarrow$ initBitset ()
            forall $\lambda_q \in C, \lambda_q \neq \lambda_a$ do
                $b \leftarrow$ bitsetAddVar $(\nu_q, b)$
16          bitsetFinish $(\nu_a, b)$
17          if $rel \neq (\texttt{RBS}(\nu_a) \cap rem)$ then
18              $Q \leftarrow Q \cup \{C_o \in R_C : \overline{\lambda_a} \in C_o\}$
```

Greedy tightening of $C^\#$

Algorithm 3.7 lists the procedure for tightening the given clause $C^\#$ directly. The notion of direct tightening indicates that conflicts during propagation are not considered at all. Those literals λ_p of $C^\#$ whose initial decision assignment $\overline{\lambda_p}$ can also be generated as an implication of other initial assignments are found.

Algorithm 3.7: Direct tightening

```
   Require Clause $C^{\#}$ to be tightened
   Return Minimised clause with a subset of literals of $C^{\#}$
   Function directTightening ($C^{\#}$)
       asymBCP ($C^{\#}$)
5      forall $\nu \in \tau$ do RBS_bkp($\nu$) ← RBS($\nu$)
       $all \leftarrow 2^{|C^{\#}|} - 1$                      | complete bitset
       $D \leftarrow \emptyset$                        | deleted literals from $C^{\#}$
8      $< R_V, R_C > \leftarrow$ allRelevant ($C^{\#}$)

10     for $i \leftarrow 0$ to $|C^{\#}| - 1$ do
11         compMandatory ($2^i, R_V, R_C$)
           $\lambda_p \leftarrow C^{\#}[i]$
13         if RBS($\nu_p$) $= 0$ then
               $D \leftarrow D \cup \lambda_p$                 | remove literal
15             $all \leftarrow$ resetRemove ($2^i, all$)
16     return $C^{\#} \setminus D$

17 Function allRelevant ($C$)
       $R_V \leftarrow \emptyset$                          | relevant variables
       $R_C \leftarrow \emptyset$                           | relevant clauses
       forall $\lambda_p \in C$ do
           $< R_V, R_C > \leftarrow < R_V, R_C > \cup$ getRelevant ($\nu_p, R_V, R_C$)
       return $< R_V, R_C >$

23 Function resetRemove ($rem, all$)
       $all \leftarrow all \cap \neg rem$
       forall $\nu_q \in R_V$ do
26         if (RBS_bkp($\nu_q$) $\cap\ rem$) $\neq 0$ then RBS_bkp($\nu_q$) ← $all$
27         RBS($\nu_q$) ← RBS_bkp($\nu_q$)              | reset bitsets
       return $all$
```

More formally, $\exists\ A \xrightarrow{\text{UP}} \overline{\lambda_p}$, whereas $A = \{\overline{\lambda_k} \neq \overline{\lambda_p} : \lambda_k \in C^{\#}\}$. Hence, there is an alternative reason clause $C = (\overline{\lambda_p} \vee K) \in$ Rsns(ν_p) whose literals $\lambda_k \in K$ can all be resolved to be replaced by some literals of $\{C^{\#} \setminus \lambda_p\}$. With this, self–subsuming resolution allows for the removal of literal λ_p from $C^{\#}$.

After propagating the opposite values of literals in $C^{\#}$, the original RBS values are backed up (line 5). The bitset *all* is the union of all bitsets used. The sets of relevant clauses and variables are initialised (line 8) by calling the procedure getRelevant (Algorithm 3.4) for all variables of $C^{\#}$ (done by

the function `allRelevant` in line 17). All literals of $C^{\#}$ are then processed separately (line 10) and the function `compMandatory` is invoked with the bitset indicating only one literal $\lambda_p = C^{\#}[i]$ in the i-th step (line 11). Note that in the `RBS` value of ν_p the i-th bit is set at most (see Algorithm 3.3). If the `RBS`(ν_p) value is cleared (line 13), then the assignment $\overline{\lambda_p}$ can be expressed by an alternative reason using only literals of $C^{\#}$. In this case, λ_p can be removed from $C^{\#}$ at the end (line 16).
The removal of a literal invalidates the `RBS` values, for which reason they are reset in line 15. Any subsequent removal of another literal must not treat literal λ_p as being contained in $C^{\#}$ anymore. However, some `RBS` values may already have been reduced due to an intersection operation with a bitset containing 2^i.

The function `resetRemove` (line 23) resets the `RBS` values to the original values that have been saved before (line 27). For those `RBS` values where an invalid bit is set, the value is replaced by the union of all valid bits (line 26). Note that the `RBS` value for each literal that is still contained in $C^{\#}$ is reset to the unique value 2^i. This allows for a valid update of the relevant `RBS` values by another call to the function `compMandatory`.

Unlike Algorithm 3.7, the procedure listed in Algorithm 3.8 also considers conflicting clauses for tightening $C^{\#}$. After the initialisation, as in Algorithm 3.7, the conflict clause C^* with the smallest conflict reason r is selected (line 6). If the conflict reason does not require all literals of $C^{\#}$ then $C^{\#}$ can already be reduced to the literals indicated by r (line 8). Unlike `directTightening`, all variables of all conflict clauses (line 13) are relevant variables in the following procedure.

As for `directTightening`, the literals of $C^{\#}$ are handled separately (line 16), and `indirectTightening` is only tried if the literal under consideration has not been removed (line 18). Basically, in line 20, for all conflict clauses, the conflict reason is recomputed. If the new conflict reason does not require all initial assignments of $C^{\#}$ (line 22), $C^{\#}$ can be tightened. In line 23, exactly one literal is chosen (its indicating bit respectively) to be removed from $C^{\#}$. Note that even if the difference between the bitsets *all* and *rem* contains more than two bits, only one literal must be removed at a time, since the removal of one literal may change the dependencies of the conflict reason. Moreover, it could happen that the recomputed conflict reason is reduced but still contains the bit 2^i under consideration. For this reason, the algorithm safely chooses any bit of the difference in line 23 and removes the corresponding literal in line 25.

Algorithm 3.8: Indirect tightening

```
Require Clause $C^{\#}$ to be tightened and Confl $\neq \emptyset$
Return Minimised clause with a subset of literals of $C^{\#}$
Function indirectTightening ($C^{\#}$)
      ...                     | as first four lines in Algorithm 3.7

6     $< C^*, r > \; \leftarrow \; < C, r > \in$ Confl : $|r|$ minimal
                              | take conflict with smallest reason
      $rem \leftarrow all \cap \neg r$                  | remove these literals
8     if $rem \neq 0$ then
          $all \leftarrow$ resetRemove ($rem, all$)
          $D \leftarrow$ literals $\in C^{\#}$ indicated by $rem$

      $< R_V, R_C > \leftarrow$ allRelevant ($C^{\#}$)
13    forall $C \in$ Confl do
          $< R_V, R_C > \; \leftarrow \; < R_V, R_C > \cup$ allRelevant ($C$)

16    for $i \leftarrow 0$ to $|C^{\#}| - 1$ do
          ...                                   | as in Algorithm 3.7
18        if $\lambda_p \in D$ then continue         | already removed
          forall $< C, r > \; \in$ Confl do
20            $cone \leftarrow \bigcup_{\lambda_q \in C}$ RBS($\nu_q$)
              $rem \leftarrow all \cap \neg cone$                 | difference
22            if $rem \neq 0$ then
23                $rem \leftarrow 2^k : (rem \cap 2^k) = 2^k$          | any one bit
                  $all \leftarrow$ resetRemove ($rem, all$)
25                $D \leftarrow$ literal $\in C^{\#}$ indicated by $rem$

      return $C^{\#} \setminus D$
```

3.2.3 Correctness and complexity

In this subsection, the correctness of the presented approach where the order of propagation is disregarded in asymmetric branching is proven. Moreover, the greedy removal of literals is justified.

Property 2. *The partial assignment τ is independent of the propagation order of the variables of $C^{\#}$.*

The DPLL branching algorithm [DLL62] uses Property 2. With CDCL solving, however, the order of propagation becomes important since different reason clauses for implied assignments may be used.

Lemma 3.2.1. *For every assigned literal, $\lambda_p \in \tau$ the set* $\texttt{Rsns}(\nu_p)$ *contains all clauses of the formula $\mathcal{F}$ that are unit under the partial assignment τ and contain literal λ_p.*
More formally: $\texttt{Rsns}(\nu_p) = \{C \in \mathcal{F} : \lambda_p \in \texttt{T}_\tau(C) \wedge \overline{\lambda_k} \in \tau \;\forall\; \lambda_k \in \{C \setminus \lambda_p\}\}$.

Proof. Let τ be the partial assignment generated by asymmetric branching with Algorithm 3.3, and let $\lambda_p \in \tau$ be any assigned literal. Let $C = (\lambda_p \vee K)$ be any clause of the formula where K is a disjunction of literals that are all $false$ under τ. C has two watched literals, one of which was set to λ_p, when the second–last assignment $\overline{\lambda_k} : \lambda_k \in K$ was propagated, at the latest. As soon as one watched literal is set to λ_p, the propagation of the other watched literal keeps C as alternative reason clause for ν_p in line 19 of Algorithm 3.3. □

For the following lemmas, it is important to notice that not all possible reductions are compatible with each other and can thus not be applied simultaneously. Due to the greedy behaviour of Algorithm 3.7 and Algorithm 3.8 some reductions may no longer be possible when particular literals have been removed previously. Consider the example with clauses $C^{\#} = (\lambda_1 \vee \lambda_2 \vee \lambda_3)$, $C_1 = (\overline{\lambda_1} \vee \lambda_2 \vee \lambda_3)$ and $C_2 = (\overline{\lambda_2} \vee \lambda_1 \vee \lambda_3)$. If literal λ_1 is removed from $C^{\#}$ by self–subsuming resolution with clause C_1, no further reductions are possible. Alternatively, literal λ_2 could be removed from $C^{\#}$ by self–subsuming resolution with clause C_2. But the reductions are not compatible and cannot be applied simultaneously.

Lemma 3.2.2. *Assume an order of literals in $C^{\#}$ exists, such that the application of asymmetric branching removes literal $\lambda_p \in C^{\#}$ by an implied assignment $\overline{\lambda_p}$ in line 7 of Algorithm 3.2. Literal λ_p is removed from $C^{\#}$ by* `directTightening` *(Algorithm 3.7) unless another literal of $C^{\#}$ is removed, which is required for self–subsuming resolution to remove literal λ_p.*

Proof. Let τ be the partial assignment generated by Algorithm 3.2 and let R be the set of reason clauses that allows the generation of clause $C' = (\overline{\lambda_p} \vee K)$, where K is a disjunction of literals of $C^{\#} \setminus \{\lambda_p\}$. Due to Property 2, the partial assignment generated by Algorithm 3.3 is equal to τ and due to Lemma 3.2.1, all reasons R are found by Algorithm 3.3. Since Algorithm 3.4 is called recursively for the variable ν_p, in line 8 of Algorithm 3.7, R_C contains all clauses of R. Let λ_p be the i-th literal of clause $C^{\#}$ as supplied to Algorithm 3.7. For all `RBS` values of variables of $K \subset C^{\#}$ the i-th bit is not set, due to the initialisation of `RBS` values in line 11 of Algorithm 3.3. Note that this also applies if a literal of $C^{\#} \setminus \{K \cup \lambda_p\}$ was removed and the bitsets of variables were reset by the function `resetRemove` in line 15 of Algorithm 3.7.

When Algorithm 3.6 is invoked with bitset 2^i, in line 11 of Algorithm 3.7, the i-th bit is cleared for all variables that have a clause of R as alternative reason. Eventually, the i-th bit is cleared for the variable ν_p and literal λ_p is removed from $C^{\#}$ in line 13 of Algorithm 3.7. □

Lemma 3.2.3. *If an order of literals in $C^{\#}$ exists, such that the application of asymmetric branching removes literals $D \subset C^{\#}$ due to a conflicting clause C_c in line 9 of Algorithm 3.2 then literals D are removed from $C^{\#}$ by Algorithm 3.8 unless another literal of $\{C^{\#} \setminus D\}$ is removed.*

Proof. The proof is analogous to the proof of Lemma 3.2.2. Let τ be the partial assignment generated by Algorithm 3.2, and let R be the set of reason clauses that are used to generate the lemma $C'_c \subseteq C^{\#}$ starting from C_c. We assume $D = \{C^{\#} \setminus C'_c\}$ as presented in Section 3.2.1. Due to Property 2, the partial assignment generated by Algorithm 3.3 is equal to τ, which implies that C_c is also found as a conflicting clause by Algorithm 3.3. Due to Lemma 3.2.1, all reasons R are found by Algorithm 3.3. Since Algorithm 3.4 is applied recursively for all variables of C_c in line 13 of Algorithm 3.8, R_C contains all clauses of R. Let d be the disjunction of the initial `RBS` values of literals in $D \subset C^{\#}$. For all `RBS` values of variables of C'_c none of the bits of d is set, due to the initialisation of `RBS` values in line 11 of Algorithm 3.3. As above, this also applies if a literal of D was removed from $C^{\#}$ and the bitsets of variables were reset by the function `resetRemove`.

When Algorithm 3.6 is invoked with one bit $b \subseteq d$ by Algorithm 3.8, bit b is successively cleared for all variables that have a clause of R as alternative reason. In particular, bit b is cleared for all variables of $C_c \in$ `Confl`. In line 25 of Algorithm 3.8, one literal of D can be removed from $C^{\#}$ unless an incompatible reduction has been done before. □

Corollary 3.2.4. *With* `indirectTightening`*, the consequence of asymmetric branching AB^* that disregards the order of propagation can be computed unless incompatible reductions are made.*

With Lemma 3.2.2 and Lemma 3.2.3, Algorithm 3.8 can tighten a given clause $C^{\#}$, as it can be done by any propagation order with asymmetric branching of $C^{\#}$ unless incompatible reductions are made.

Corollary 3.2.5. *For a given clause $C^{\#}$, let $MinimalAB$ be the problem to compute a minimal set of the literals of $C^{\#}$ to which $C^{\#}$ can be tightened by the application of asymmetric branching AB^*. The $MinimalAB$ problem can be computed in polynomial time.*

With Corollary 3.2.4, all compatible reductions of clause $C^{\#}$ are applied. When the main algorithm terminates, all compatible reductions of $C^{\#}$ are made. Algorithm 3.4 is basically a depth–first search procedure and can be bounded by $|R_V| + |R_C|$. In Algorithm 3.6, the `RBS` value for one variable is never increased. In particular, Algorithm 3.6 is always called with a bitset where only one bit 2^i is set, so each variable can only modify its `RBS` value once. Thus each clause can only change the `RBS` value of its asserted literal once. The implementation can be improved by choosing one responsible variable within each clause $\in R_C$, whose `RBS` value contains the bit 2^i. When the `RBS` value is changed for a variable ν_p, only the clauses for which ν_p is responsible have to be put into the queue Q in line 18 of Algorithm 3.6. With this, the inspection of each clause C in lines 13 to 16 requires each literal of C to be touched once at most. Hence, one invocation of Algorithm 3.6 is linear in the number of literals of R_C. Since Algorithm 3.6 is called $|C^{\#}|$ times at most (once for each initial bit), the entire computation runs in polynomial time. Clearly, the unit propagation for $|C^{\#}|$ decisions in Algorithm 3.3 also runs in polynomial time.

Tightening to minimum size

The presented algorithms do not claim to tighten a given clause $C^{\#}$ to the minimum size that could be achieved by considering all the possibilities of asymmetric branching. This is justified by the fact that finding the minimum subset of literals is NP–hard. Moreover, it is already NP–hard to reduce a given clause to its minimum if only self–subsuming resolution is applied.

Corollary 3.2.6. *AB^* of clause $C^{\#}$ finds all clauses that subsume $C^{\#}$ and all clauses that tighten $C^{\#}$ by self–subsuming resolution, provided that no incompatible reductions are made previously.*

Proof. Any clause $C_s \subseteq C^{\#}$ generates a conflict by the application of common asymmetric branching (Section 3.2.1) and thus the detection of subsuming clauses is obvious. As described in Section 3.2.1, any clause C_r that allows the application of self–subsuming resolution for $C^{\#}$ has the following properties: $C_r = (\lambda_p \vee K)$, where K is a disjunction of the literals of $C^{\#}$ and $\overline{\lambda_p} \in C^{\#} \setminus \{K\}$. If common asymmetric branching assigns the opposite values of all literals in K, the assignment $\overline{\lambda_p}$ is implied due to clause C_r. Literal l can be removed from $C^{\#}$ by Algorithm 3.2 in line 7. With Corollary 3.2.4, literal λ_p is removed from $C^{\#}$ if no other incompatible reduction is made. □

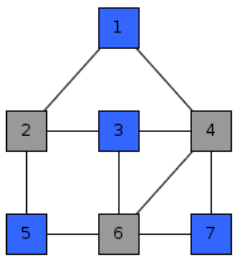
Lemma 3.2.7. *Let $SelfSub$ be the problem of tightening a given clause by the application of self–subsuming resolution. Let $MinSelfSub$ be the variant of $SelfSub$ that tightens a given clause to minimum size. $MinSelfSub$ is NP–hard.*

Lemma 3.2.7 can be proven by the reduction $MaxIS \leq_{poly} MinSelfSub$, where MaxIS is the Maximum Independent Set problem. MaxIS is a graph theoretical problem, and an independent set IS in an undirected graph

$G = (V_G, E_G)$ is a subset of vertices $I \subseteq V_G$, such that each edge in E_G has, at most, one endpoint in I. Thus no pair of vertices in I is adjacent. A maximum independent set I^* is an independent set that contains the greatest number of vertices possible, such that there is: $|I^*| \geq |I|\ \forall I$ of G. MaxIS is NP–hard for $\emptyset \neq I^* \neq V_G$ [GJ79].

For some given undirected graph $G = (V_G, E_G)$ the problem $MaxIS$ of finding a maximum independent set of vertices of G can be transformed into a problem of minimum self–subsuming resolution $MinSelfSub_G$ with $|V_G|$ variables and $|V_G|+1$ clauses. Without loss of generality, G does not contain self–loops, i.e. edges with the same source and target vertex.

For each vertex i of V_G, one variable λ_i is introduced for $MinSelfSub_G$. Let $C^{\#} = (\lambda_1 \vee \lambda_2 \vee \ldots \vee \lambda_{|V_G|})$ be the clause to be tightened, which contains all variables with positive polarity. In addition, each vertex $i \in V_G$ introduces one clause $C_i = (\overline{\lambda_i} \vee \alpha_i)$, where α_i is the disjunction of the (positive) literals of vertices in A_i and A_i denotes the set of corresponding vertices adjacent to i in G. Hence literal λ_j is contained in clause C_i if and only if an edge $i, j \in E_G$ exists in G. Note that ν_i is the only variable with negative polarity in C_i. Moreover, C_i cannot contain λ_i since G has no self–loops, i.e. edges (i, i). A small example of a graph and the corresponding clause set is shown in Figure 3.2.



$$\begin{aligned}
C^{\#} &= \{\lambda_1 \vee \lambda_2 \vee \lambda_3 \vee \lambda_4 \vee \lambda_5 \vee \lambda_6 \vee \lambda_7\}\\
C_1 &= \{\overline{\lambda_1} \vee \lambda_2 \vee \lambda_4\}\\
C_2 &= \{\overline{\lambda_2} \vee \lambda_1 \vee \lambda_3 \vee \lambda_5\}\\
C_3 &= \{\overline{\lambda_3} \vee \lambda_2 \vee \lambda_4 \vee \lambda_6\}\\
C_4 &= \{\overline{\lambda_4} \vee \lambda_1 \vee \lambda_3 \vee \lambda_6 \vee \lambda_7\}\\
C_5 &= \{\overline{\lambda_5} \vee \lambda_2 \vee \lambda_6\}\\
C_6 &= \{\overline{\lambda_6} \vee \lambda_3 \vee \lambda_4 \vee \lambda_5 \vee \lambda_7\}\\
C_7 &= \{\overline{\lambda_7} \vee \lambda_4 \vee \lambda_6\}
\end{aligned}$$

Figure 3.2: Transformation of Independent Set to self–subsuming resolution

The transformation is clearly polynomial. It remains to be proven that any solution of IS can be transformed into a solution for $SelfSub_G$ and *vice versa*.

$IS \Rightarrow SelfSub_G$: Let I be any given independent set for G. We define $H = \{V_G \setminus I\}$ as the complementary vertex set[1] of I. In the transformed

[1]Note that H is a Vertex Cover for G.

$SelfSub$ problem, $C^{\#}$ can be tightened stepwise to C', where a literal λ_i is in C' if and only if vertex $i \in H$. Where necessary, we refer to the different versions of $C^{\#}$ as $C_0^{\#} \supset C_1^{\#} \supset \ldots \supset C_{|I|}^{\#} = C'$.

For any vertex $i \in I$, consider clause $C_i = (\overline{\lambda_i} \vee \alpha_i)$ as defined above. By the definition of an independent set, all adjacent vertices to i, i.e. vertices in A_i, are elements of H. More formally, we have $A_i \subseteq H$ and $A_i \cap I = \emptyset$. Consequently, all literals in α_i are contained in C', and thus we have $\alpha_i \subseteq C' \subseteq C_k^{\#}\ \forall\ 0 \leq k \leq |I|$. The resolution of the two clauses $C_k^{\#}$ and C_i deduces the resolvent C^i that contains all literals of $C_k^{\#}$ except for literal λ_i. Hence C^i subsumes $C_k^{\#}$ and literal λ_i can be removed from $C^{\#}$ by self–subsuming resolution.
Vertices of I can be handled in arbitrary order to tighten clause $C^{\#}$ step by step to C'. One step for any vertex $j \in I$ is independent of all other steps since the resolution operation requires only the literals of $\alpha_j \subseteq C'$ that are contained in all versions $C_k^{\#} : 0 \leq k \leq |I|$.

$SelfSub_G \Rightarrow IS$: Let $C^{\#}$ be tightened to clause C'. Let $R = \{C^{\#} \setminus C'\}$ be the literals that are removed by the application of self–subsuming resolution. Hence several resolution operations are performed to resolve clause C'. In the corresponding IS problem, I is an independent set, where I contains exactly the vertices corresponding to the variables in R. We assume the contrary, that there are two vertices $i, j \in I$ that are adjacent in G. Thus the literals λ_i, λ_j are removed from $C^{\#}$. To remove λ_i (or λ_j) from $C^{\#}$ by self–subsuming resolution, the clause C_i (or C_j) has to be used for resolution, since literal $\overline{\lambda_i}$ (or $\overline{\lambda_j}$) has only one occurrence with negative polarity. W.l.o.g. we assume λ_i is removed from clause $C_l^{\#}$ first, in step k, and λ_j is removed later in step l $(k < l < |I|)$.

For the removal of λ_j, the clauses $C_l^{\#}$ and $C_j = (\overline{\lambda_j} \vee \alpha_j)$ are resolved to C^j. Since i and j are adjacent in G, we have $\lambda_i \in \alpha_j$. Accordingly, λ_i is contained in C^j but is not contained in $C_l^{\#}$, since it was removed in $C_{k+1}^{\#} \supseteq C_l^{\#}$. With $C^j \not\subset C_l^{\#}$, self–subsuming resolution cannot be applied to remove literal λ_j from $C_l^{\#}$.

Corollary 3.2.8. *It is NP–hard to tighten a clause to its minimum size by the application of asymmetric branching that disregards the order of propagation.*

Corollary 3.2.8 follows directly from Lemma 3.2.7 and the fact that self–subsuming resolution is covered by the modification of asymmetric branching presented here.

3.3 Supplementary Hyper–Binary Resolution

Removing a literal from any clause of a Boolean formula in CNF cannot decrease its deductive power [HS07]. As mentioned in Section 3.2.1, simplification techniques may allow redundant clauses to be removed. Moreover, simplification may also add redundant clauses, as this is also done by CDCL solving when conflict analysis generates a new clause. These simplifications have to be applied carefully, since it is not obvious whether the modified formula is easier or even harder to solve for a particular SAT solving algorithm.

The idea of hyper–binary resolution allows for the generation of redundant binary clauses. Since the method for detecting hyper–binary resolution uses a slight modification of unit propagation, it can be incorporated into CDCL solving, and, in particular, it can be applied within asymmetric branching. For example, this is applied in distillation of clauses within some solvers implemented by Biere [Bie09b, Bie11].
In general, increasing the number of binary clauses can be beneficial for several other simplification techniques, such as equivalence reasoning, where strongly connected components in the binary implication graph are replaced by one representative literal. Moreover, the simplification techniques presented by Brafman and Heule *et al.* [Bra04, HM04, HJB11] and the approach suggested in Chapter 4 benefit from having a large amount of binary clauses. Furthermore, added binary clauses may increase the power of unit propagation and thus they may improve the power of asymmetric branching.

In Section 3.3.1, the concept of hyper–binary resolution is presented. Thereafter, hyper–binary resolution is incorporated into the algorithms presented in the previous sections. Despite the benefit of additional binary clauses in other simplification techniques and propagation, they may also slow down Boolean constraint propagation. For this reason, all binary clauses that are added by hyper–binary resolution are marked as special redundant clauses so that the SAT solver can decide whether to use or ignore these clauses for CDCL search.

3.3.1 Related work

The concept of hyper–binary resolution was introduced by Bacchus and Winter [BW03] and goes back to Robinson [Rob83]. Consider the following example with $n + 1$ clauses: $C_0 = (\lambda_x \vee \lambda_1 \vee \lambda_2 \vee \ldots \vee \lambda_n)$ and the binary clauses $C_1 = (\lambda_y \vee \overline{\lambda_1})$, $C_2 = (\lambda_y \vee \overline{\lambda_2})$, $\ldots$, $C_n = (\lambda_y \vee \overline{\lambda_n})$. One resolution of C_0 with C_1 deduces the clause $C' = (\lambda_x \vee \lambda_y \vee \lambda_2 \vee \ldots \vee \lambda_n)$. Thereafter, C' can be tightened to $C'' = (\lambda_y \vee \lambda_x)$ by using the clauses $C_2 \ldots C_n$ for self–subsuming resolution. C'' can be added to the formula as a redundant binary clause. In special cases, λ_y may be equal to λ_x, which allows for the deduction of a unit clause $C'' = (\lambda_y)$. The example becomes less obvious if we allow for binary relations that are not explicitly given by one binary clause but by a set of binary clauses. If we replace clause C_2 from above with two clauses $C_{2a} = (\lambda_z \vee \overline{\lambda_2})$ and $C_{2b} = (\lambda_y \vee \overline{\lambda_z})$, hyper–binary resolution can still generate clause C'' if implicit binary relations are considered.

Based on the work of Gershman and Strichman [GS05], the idea of lazy hyper–binary resolution was introduced by Biere [Bie09a] to apply hyper–binary resolution on–the–fly and efficiently during unit propagation within search. Unit propagation has to be extended in three ways:

- For any assigned literal λ_p, all binary clauses that contain literal $\overline{\lambda_p}$ are propagated first. Clauses containing more than two literals are propagated, but not before propagation of the binary clauses of all the assigned literals is finished.
- Each assignment λ_p keeps a so–called dominator $dom(\nu_p)$. For any assignment that is implied during the propagation of binary clauses, the dominator is equal to the assignment that first initiated the propagation. By default, the dominator of an assignment is equal to the assigned literal itself.
- If a clause C_p, which has more than two literals, becomes the asserting clause for an assignment λ_p, it is checked whether all literals of $\{C_p \backslash \lambda_p\}$ have the same dominator λ_q. If this applies, a binary clause $(\lambda_p \vee \overline{\lambda_q})$ is added.

Consider the clauses $C_0 \ldots C_n$ from the example above and assume that literal $\overline{\lambda_y}$ is assigned by a decision. Its dominator is the literal $\overline{\lambda_y}$ itself. Propagation of the binary clauses implies the assignments $\xrightarrow{\text{UP2}} \overline{\lambda_1} \wedge \ldots \wedge \overline{\lambda_n}$, due to the clauses $C_1 \ldots C_n$. Each assignment gets the same dominator $\overline{\lambda_y}$. After the propagation of binary clauses, literal λ_x is assigned due to clause C_0. Since all literals in $C_0 \setminus \{\lambda_x\}$ have the same dominator $\overline{\lambda_y}$, the binary clause $(\lambda_x \vee \overline{\overline{\lambda_y}}) = (\lambda_x \vee \lambda_y) = C''$ is added.

3.3.2 Asymmetric branching and hyper–binary resolution

As described in Section 3.3.1, dominator variables can be used to detect the possibility of hyper–binary resolution. However, an unfavourable order of propagation in asymmetric branching may miss some possibilities of adding new binary clauses. Consider the following example with the clauses $C^{\#} = (\lambda_1 \vee \lambda_2 \vee \lambda_3), C_1 = (\lambda_1 \vee \overline{\lambda_5}), C_2 = (\lambda_2 \vee \overline{\lambda_6}), C_3 = (\lambda_5 \vee \lambda_6 \vee \overline{\lambda_7})$ and $C_4 = (\lambda_2 \vee \overline{\lambda_5})$. When propagating the opposite literals of $C^{\#}$ in the given order, literal $\overline{\lambda_5}$ is set by C_1 with the dominator $\overline{\lambda_1}$, literal $\overline{\lambda_6}$ is set by C_2 with the dominator $\overline{\lambda_2}$ and literal $\overline{\lambda_7}$ is set by C_3 where the dominator is $\overline{\lambda_7}$ itself. If, however, $\overline{\lambda_2}$ was propagated before $\overline{\lambda_1}$, the use of dominators would detect a hyper–binary resolution: $\overline{\lambda_5}$ is now assigned by C_4 with the dominator $\overline{\lambda_2}$, literal $\overline{\lambda_6}$ is assigned by C_2 also with the dominator $\overline{\lambda_2}$. Literal $\overline{\lambda_7}$ is assigned by C_3 with the dominator $\overline{\lambda_2}$ since the assignments $\overline{\lambda_5}$ and $\overline{\lambda_6}$ both have the dominator $\overline{\lambda_2}$. Thus the binary clause $(\overline{\overline{\lambda_2}} \vee \overline{\lambda_7}) = (\lambda_2 \vee \overline{\lambda_7})$ can be added.

Because in asymmetric branching, the number of decision variables is bounded in advance, the dominators can be expressed as bitsets that indicate the decision variables, analogous to reasons in Section 3.2.2. Unlike normal dominators, the dominator bitsets only represent the decision variables. By dismissing the order of propagation for the `RBS` values, the dominator bitsets can also be adapted. This allows for a better detection of hyper–binary resolution. In this section, the idea of dominators is incorporated into the AB^* computation. The algorithms presented in the previous section are adapted to meet the requirements of enabling hyper–binary resolution. At first, binary clauses have to be propagated before other clauses.

Algorithm 3.9 modifies Algorithm 3.3. Since binary clauses have to be propagated before other clauses, two propagation queues are initialised in line 4. When a literal is enqueued for propagation, it is always put into both queues (lines 7 and 20). As for the `RBS` values, the dominator bitsets `DBS` are initialised with one unique bit for each literal of $C^{\#}$ in line 10. In line 13, binary clauses are propagated separately as listed in Algorithm 3.10. The first two `if`–cases of the inner loop until line 19 are equal to those in Algorithm 3.3. If unit propagation implies an assignment for an unassigned variable, the `DBS` is computed analogously to the use of a single dominator, i.e. as the intersection of the `DBS` values of all falsified literals of C^* (line 24). The function `addHyperBinary` invoked in line 25 adds new binary clauses if the `DBS` value is not zero.

As for the case of unit clauses, a dominator bitset can be computed for conflicting clauses. The intersection of all `DBS` values of the variables in C^* is computed in line 27. If this value is not zero, there is at least one initial assignment that implies the assignment of all literals of C^*. In that case, a unit clause can be added which is mimicked in the pseudo–code by setting the second literal of a generated binary clause to $false$ (line 28). If a unit clause can be added in line 28, $C^{\#}$ is subsumed by that clause and asymmetric branching terminates early.

The invoked function `addHyperBinary` in line 32 checks if the given dominator bitset indicates clauses to be added (line 33). For each dominator that is set within the given bitset in line 35, a binary clause is added to $\mathcal{F}$.

Algorithm 3.9: Asymmetric branching with hyper–binary resolution.

```
Require Clause $C^{\#} \in \mathcal{F}$ to be tightened
$\tau \leftarrow \emptyset$; $\texttt{Confl} \leftarrow \emptyset$; $\texttt{Rsns}(\nu) \leftarrow \emptyset \; \forall \; \nu \in \mathcal{V}$        | see Algorithm 3.3
Function asymBCP ($C^{\#}$)
    $Q \leftarrow \emptyset$; $B \leftarrow \emptyset$                               | two queues to propagate
    for $i \leftarrow 0$ to $|C^{\#}| - 1$ do
        $\lambda_q \leftarrow C^{\#}[i]$
        $B.\texttt{enqueue}(\overline{\lambda_q})$; $Q.\texttt{enqueue}(\overline{\lambda_q})$        | propagate separately
        $\tau \leftarrow \tau \cup \overline{\lambda_q}$                             | assign opposite literal
        $\texttt{RBS}(\nu_q) \leftarrow 2^i$                           | init bitset for reasons
        $\texttt{DBS}(\nu_q) \leftarrow 2^i$                       | init bitset for dominators
    while $Q \neq \emptyset$ do
        $Q \leftarrow Q \cup \texttt{bcpBins}(B)$                          | binaries first
        $\lambda_q \leftarrow Q.\texttt{dequeue}()$                 | next literal to propagate
        $W_{\overline{q}} \leftarrow \texttt{watchedOf}(\overline{\lambda_q})$              | clauses with watched $\overline{\lambda_q}$
        foreach $C^* \in \{W_{\overline{q}} \setminus C^{\#}\} : |C^*| > 2$ do
            $\lambda_p \leftarrow \texttt{otherWatched}(C^*, \overline{\lambda_q})$
            ...          | first two if cases as in Algorithm 3.3
            else if $\overline{\lambda_p} \notin \tau$ then
                $B.\texttt{enqueue}(\lambda_p)$; $Q.\texttt{enqueue}(\lambda_p)$         | separate BCP
                $\tau \leftarrow \tau \cup \lambda_p$
                $\texttt{Rsns}(\nu_p) \leftarrow C^*$
                $\texttt{RBS}(\nu_p) \leftarrow \bigcup_{\lambda_k \in \{C^* \setminus \lambda_p\}} \texttt{RBS}(\nu_k)$
                $\texttt{DBS}(\nu_p) \leftarrow \bigcap_{\lambda_k \in \{C^* \setminus \lambda_p\}} \texttt{DBS}(\nu_k)$
                addHyperBinary $(C^{\#}, \lambda_p, \texttt{DBS}(\nu_p))$
            else if $C^* \notin \texttt{Confl}$ then
                $d \leftarrow \bigcap_{\lambda_k \in C^*} \texttt{DBS}(\nu_k)$                 | has dominators?
                if addHyperBinary $(C^{\#}, \mathit{false}, d)$ then
                    return                  | unit clauses subsume $C^{\#}$
                $r \leftarrow \bigcup_{\lambda_k \in C^*} \texttt{RBS}(\nu_k)$
                $\texttt{Confl} \leftarrow \texttt{Confl} \cup (C^*, r)$             | keep conflict
Function addHyperBinary ($C^{\#}, \lambda, \mathit{mask}$)
    if $\mathit{mask} = 0$ then return $\mathit{false}$
    for $i \leftarrow 0$ to $|C^{\#}| - 1$ do
        if $(2^i \cap \mathit{mask}) \neq 0$ then
            $\mathcal{F} \leftarrow \mathcal{F} \cup (C^{\#}[i], \lambda)$               | add binary clauses
    return $\mathit{true}$
```

Algorithm 3.10: Propagate binaries separately

```
   Require Queue of literals to be propagated
   Return A queue of literals to be propagated using non–binary clauses
   Function bcpBins ($B$)
4      $Q \leftarrow \emptyset$;
       while $B \neq \emptyset$ do
6          $\lambda_q \leftarrow B$.dequeue()        | next literal to be propagated
           $W_{\overline{q}} \leftarrow$ getBinariesWith($\overline{\lambda_q}$)           | binaries $(\overline{\lambda_q}, *)$
           foreach $C^* \in \{W_{\overline{q}} \setminus C^\#\}$ do
               $\lambda_p \leftarrow$ otherWatched($C^*, \overline{\lambda_q}$)
               if $\lambda_p \in \tau$ then
                   Rsns($\nu_p$) $\leftarrow$ Rsns($\nu_p$) $\cup\, C^*$
12                 DBS($\nu_p$) $\leftarrow$ DBS($\nu_p$) $\cup$ DBS($\nu_q$)
13                 RBS($\nu_p$) $\leftarrow$ RBS($\nu_p$) $\cap$ RBS($\nu_q$)          | optional
                   if RBS($\nu_p$) $= 0$ and $\overline{\lambda_p} \in C^\#$ then remove $\overline{\lambda_p}$ from $C^\#$
15             else if $\overline{\lambda_p} \in \tau$ then
                   ...          | as conflict case in Algorithm 3.9
               else
18                 $B$.enqueue($\lambda_p$); $Q$.enqueue($\lambda_p$)       | separate BCP
                   $\tau \leftarrow \tau \cup \lambda_p$
                   Rsns($\nu_p$) $\leftarrow C^*$
21                 RBS($\nu_p$) $\leftarrow$ RBS($\nu_q$)
                   DBS($\nu_p$) $\leftarrow$ DBS($\nu_q$)
       return $Q$
```

The preferential propagation of binary clauses is outlined in Algorithm 3.10. The set Q (line 4) keeps the implied assignments that have to be returned to the main propagation routine. In line 6, the next literal to propagate is taken from the binary propagation queue. If the other literal λ_p of a propagated binary clause C^* is already assigned *true*, the DBS value for ν_p is unified with the dominator bitset of this propagation to allow different dominators for the assignment of λ_p (line 12). In line 13, the RBS value may be adapted immediately. A literal $\overline{\lambda_p}$ may then be removed from $C^\#$ in the subsequent line. However, these two lines are optional and a removal would be detected and applied later (in line 13 within directTightening).

For conflicting clauses (line 15), the treatment is equal to the one in Algorithm 3.9. If an assignment is implied, the literal has to be added into both propagation queues (line 18). The two bitsets for a newly assigned variable, RBS and DBS, are both set as equal to the implying assignment of λ_q as shown in line 21.

Algorithm 3.11: Adaption of Algorithm 3.5 for hyper–binary resolution

```
    Function initBitset ()
2       return $< 0, 2^{|C^{\#}|} - 1 >$
    Function bitsetAddVar ($\nu_q, < r, d >$)
        $r \leftarrow r \cup$ RBS($\nu_q$)
5       $d \leftarrow d \cap$ DBS($\nu_q$)
        return $< r, d >$
    Function bitsetFinish ($\nu_p, < r, d >$)
8       $new \leftarrow d \cap \neg$DBS($\nu_p$)                  | new dominator bits
        DBS($\nu_p$) $\leftarrow$ DBS($\nu_p$) $\cup\ d$
10      if DBS($\nu_p$) $\neq 0$ then
11          if |DBS($\nu_p$)| $> 1$ then $r \leftarrow 0$
12          else $r \leftarrow r \cap$ DBS($\nu_p$)
13      RBS($\nu_p$) $\leftarrow$ RBS($\nu_p$) $\cap\ r$
14      addHyperBinary ($C^{\#}, \lambda_p \in \tau, new$)       | add binary clauses
```

Whenever an alternative reason clause is processed in Algorithm 3.4 and Algorithm 3.6, the bitset RBS is updated. The update is done by three functions listed in Algorithm 3.5. To meet the requirements for hyper–binary resolution their functionality has to be modified, as shown in Algorithm 3.11.

The RBS values for the falsified literals of one reason clause are always unified, since an assignment asserted by that clause depends on each of the indicated assignments. However, the DBS values are intersected since a dominator is only valid if all falsified literals have a common dominator. Thus in line 5 of Algorithm 3.11, the DBS values are intersected and are therefore initialised to the complete bitset in line 2.

After all literals of a reason clause are processed, the dominator bits that are not set in the current DBS value are determined in line 8, before the DBS value is extended. If the DBS value contains at least one dominator, it can be used to reduce the RBS value for that variable (line 10). If the assignment of ν_p is solely implied by two different initial assignments (line 11), then ν_p has no mandatory assignments. If the assignment of ν_p has one dominator (line 12) then this initial assignment can be the only mandatory assignment for ν_p. Compared to the computation in Section 3.2, the use of dominator bitsets to reduce the reason bitsets does not induce other RBS values. However, it speeds up the reduction of RBS values. In line 13, the RBS value is finally reduced to the mandatory assignments for this stage. New binary clauses $(\lambda_p \vee *)$ are created for all new dominator bits in line 14 by calling the function listed in Algorithm 3.9, whereas λ_p is the assigned literal of variable ν_p.

3.4 Evaluation

In the following section, we evaluate the different approaches that are presented in this chapter. For the evaluation of any algorithm, it is desirable to measure only those effects that are directly related to the evaluated technique. For this reason, test runs with different solver configurations are generally performed by changing only one parameter under consideration. However, there are some parameters that may have several side–effects. Consider the application of preprocessing. Besides the simplification of a SAT instance many preprocessing approaches may implicitly change the order in which variables, literals or clauses are added to the solver. As already illustrated in the Introduction, such minor changes in the proceedings of a SAT solver may have significant effects.
All evaluations are based on a solid, pure CDCL implementation using C++. This is basically the CDCL part of our solver SApperloT without special tweaks. In doing so, we attempt to keep the side–effects caused by other solving techniques as small as possible. The solver applies the VSIDS heuristic with phase saving for decision making with no degree of randomness. Moreover, it applies the Glucose restart policy and uses LBD values for the estimation of clause's quality, as described in Section 2.2.5. Other features of Glucose, such as the additional conflict minimisation using binary clauses, are not applied.

3.4.1 The `XOR`–watchers implementation

In Section 3.1, we introduced the `XOR`–watchers scheme to speed up unit propagation in SAT solving. We apply this technique in all CDCL–based solving modules of the actual versions of our solvers SApperloT (after version 2010), SArTagnan and MoUsSaka. Note that the first two solvers utilise additional solving techniques, which are described in more detail in Chapter 5 and Chapter 6. Moreover, for the parallel solver SArTagnan, the `XOR`–watchers scheme had to be modified, which is also described in Chapter 6.

To evaluate the effect of the suggested technique, tests have been run on all 614 industrial instances of the SAT competitions of 2007 and 2009 [Sat11] and the SAT–Races of 2008 and 2010 [Sat10]. Figure 3.3 presents the result by using a so–called cactus plot, as used in the Introduction. A point (x, y) in the plot indicates that x instances can be solved when the time per instance is limited to y seconds.

The `XOR`–watchers implementation is represented by the green curve. The common watched literals implementation, where the two watched literals correspond to the first two literals in the clause [Gel02], is represented by the red curve. For more than 650 seconds, the green curve is clearly more to

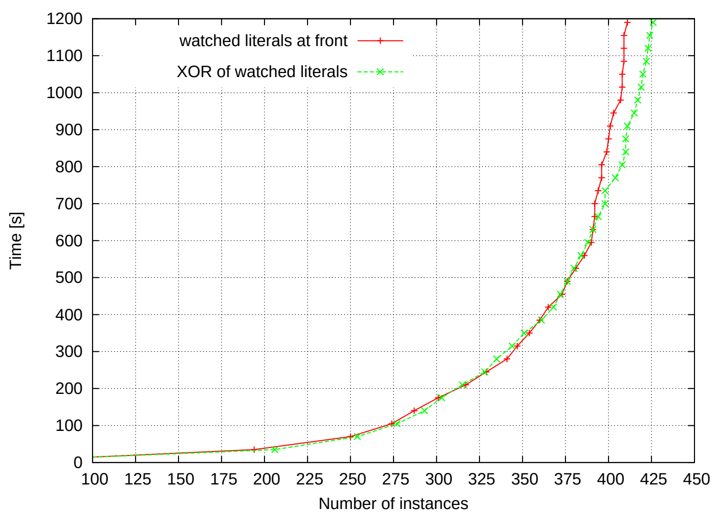


Figure 3.3: Comparison of the watched literals implementation on 614 benchmarks

the right than the red curve. This indicates the better performance of the XOR–watchers implementation, since it solves considerably more instances than the common watched literals implementation within the same amount of time. With a time bound below 650 seconds, no effect is observable. This may be due to the increasing number of learnt clauses when the solver runs for a longer time. On average, the solver configuration that used the XOR–watchers implementation allocated one MB less memory. In Section 6.3, we revisit Figure 3.3 when analysing the modified data structure for parallel solvers.

3.4.2 Asymmetric branching with hyper–binary resolution

In Section 3.2, we present an approach that extends the application of asymmetric branching to be independent of the order of propagation (AB^*). In addition, in Section 3.3, the application of hyper–binary resolution is incorporated into the approach.
In the following, we analyse the effect of both techniques with several different configurations. We study the presented approaches in combination with other common simplification techniques, such as subsumption, self–subsuming resolution and variable elimination as described in Section 3.2.1 (cf. [SP04, EB05, Zie10]). Moreover, we apply equivalence reasoning, as presented by Brafman [Bra01], where each occurrence of literal λ_i (or $\overline{\lambda_i}$) is replaced by literal λ_j (or $\overline{\lambda_j}$) if both binary implications $\lambda_i \xrightarrow{\text{UP2}} \lambda_j$ and

$\overline{\lambda_i} \xrightarrow{\text{UP2}} \overline{\lambda_j}$ hold. This technique depends on the binary clauses of a formula and is thus interesting to use in combination with hyper–binary resolution. The computation of equivalent literals is revisited in Chapter 4. Note that AB^* is only applied when no other simplification is possible. All tests are performed for the 300 instances of the application (or industrial) category of the SAT competition 2011 [Sat11]. However, most presented plots depict the results for only a subset of instances to emphasise relevant issues. We analyse the presented approaches in terms of quality by considering different aspects. Moreover, their impact on the total runtime of our CDCL solver is evaluated. We distinguish between the application in preprocessing and the more frequent application in inprocessing (in between CDCL searches).

Reduction of literals

Figure 3.4 compares the tightening of clauses by the application of asymmetric branching AB^* for different configurations of preprocessing. Each configuration is represented by one plotted curve. The y–axis indicates the average number of literals that are removed by the execution of `indirectTightening` for one clause, $C^\#$ (cf. Algorithm 3.8). For each configuration, the instances are ordered by their y values such that all curves decrease monotonically. Thus a point (x, y) in the plot indicates the number of instances x for which at least y literals are removed on average.

We distinguish four parameters for each configuration. The first parameter, $cs < n$, states the maximal clause size (cs = number of literals) for which AB^* is applied. The second parameter indicates whether all simplification techniques are applied (*all*) or only AB^* and equivalence reasoning ($ab + er$). This distinction is interesting because neither `indirectTightening` nor equivalence reasoning require occurrence lists of variables or literals. If *dflt* is stated as the third parameter, input clauses are considered by default. Otherwise, clauses are only considered if they are modified by another simplification technique. Finally, the fourth parameter indicates if hyper–binary resolution is applied additionally (*hb*), as presented in Section 3.3.

We start by observing the first configuration (red plot), where only clauses with less than 13 literals are considered for AB^*. As expected, it removes the least number of literals on average. Clearly, the removal of literals from longer clauses seems more likely, but may probably have less effect on the solving process of the formula. The tightening achieved for this configuration seems quite small. The curve indicates that for 50 of 300 instances only every fifth call of `indirectTightening`, at least, is successful. However, recall that AB^* is only invoked after all other simplification techniques applied to the full. Moreover, the removal of one literal by AB^* may trigger the application of other simplification techniques.

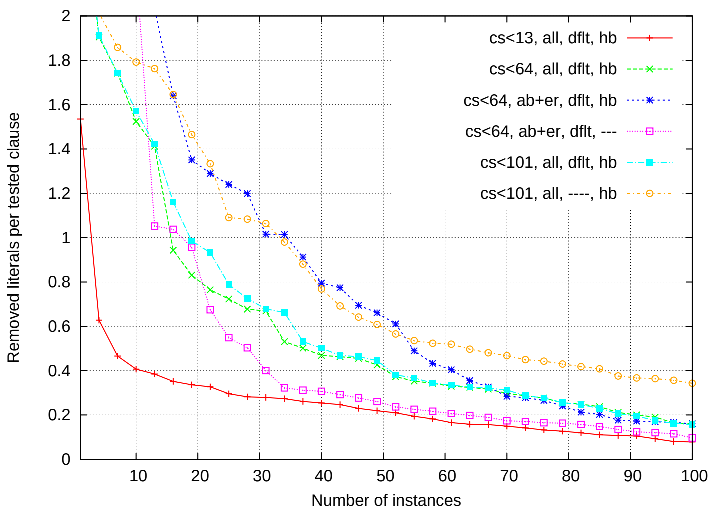


Figure 3.4: Reduction of literals by asymmetric branching AB^*

Now consider the next two configurations (green and blue). They differ only in the application of other simplification techniques. The second of these configurations, where only AB^* and equivalence reasoning are applied, exhibits the clearly better effect of asymmetric branching. The difference of these two configurations demonstrates how many literals are removed by self–subsuming resolution that could also be detected by AB^*. However, since the application of self–subsuming resolution is less costly than asymmetric branching, its application is preferred.

The third and fourth configurations (blue and magenta) differ in their additional application of hyper–binary resolution. The benefit for AB^* when hyper–binary clauses are added on–the–fly is significant. Note that both configurations only apply AB^* and equivalence reasoning. If other simplification techniques are also applied, the benefit of adding hyper–binary clauses is less observable for AB^*. This is because other simplification techniques, such as subsumption and self–subsuming resolution, obtain greater benefits from the added binary clauses.

The last two configurations (cyan and orange) apply AB^* for long clauses. The interesting issue is the default treatment of clauses. The last configuration applies AB^* only for clauses that have already been simplified by other techniques. The effect is significant and shows that different heuristics on the choice of clauses that are considered by AB^* exhibit significantly different behaviour.

The fifth and first configurations (cyan and red) only differ in the size of the clauses considered by AB^*. It is worth mentioning that the more extensive removal of literals (cyan) goes along with an increase in the average runtime by a factor of five, compared to the first configuration.

Different parts of AB^*

In Section 3.2, three main parts of AB^* are distinguished, which are represented by the functions `asymBCP`, `getRelevant` and `compMandatory` (Algorithms 3.3, 3.4 and 3.6). Moreover, we distinguish between direct and indirect tightening in Algorithm 3.7 and Algorithm 3.8. The incorporation of hyper–binary resolution requires the separate propagation of binary clauses, as explained in Section 3.3.

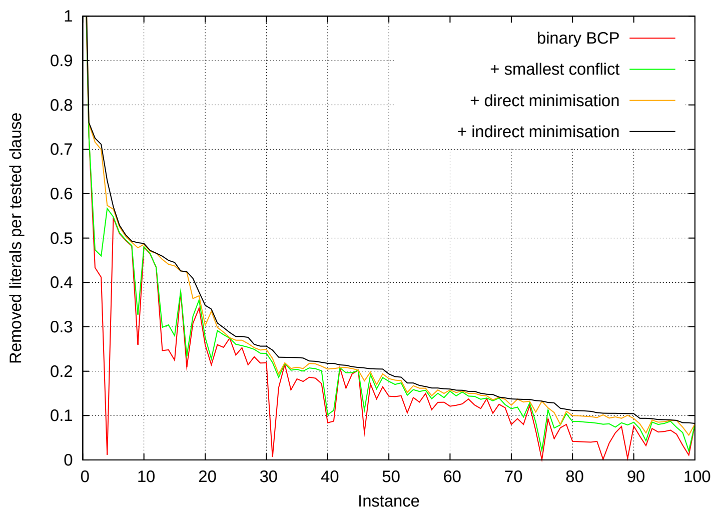


Figure 3.5: Relation of different parts of AB^*

Figure 3.5 and Figure 3.6 compare these different parts of the AB^* approach in terms of their contribution to tightening and runtime for one preprocessor configuration that applies hyper–binary resolution. Figure 3.5 distinguishes four parts of AB^* at which the tightening of a clause may be detected: The propagation of binary clauses (line 13 of Algorithm 3.10), the inspection of conflict reasons after propagation is completed (line 6 of Algorithm 3.8), direct tightening (Algorithm 3.7) and indirect tightening (from line 18 of Algorithm 3.8). The average number of literals removed per inspected clause is shown on the y–axis. The contributions of the four parts of AB^* are stacked. Each x value represents one SAT instance. Thus, for a SAT instance represented by x_i, the vertical line l_i at position x_i depicts how many literals are removed by binary propagation (the intersection of l_i with the red curve), how many literals are additionally removed by choosing the smallest conflict (the intersection of l_i with the green curve) and so on. All instances are ordered by the total number of literals removed (black curve).

For the first 100 instances (according to the order) depicted in Figure 3.5, it is clearly observable that most literals can be removed by the propagation of binary clauses. A much smaller number of literals are removed by considering the smallest conflict reason (distance between the green and red curves) and direct tightening (distance between the orange and green curves). The application of indirect tightening (distance between the black and orange curves) has only a very small effect. However, the propagation of binary clauses benefits from the application of hyper–binary resolution in the minimisation parts.

Figure 3.6 compares the runtime of four different parts within AB^* for the same configuration as Figure 3.5: The propagation of binary (red curve) and non–binary (green curve) clauses, the execution of `getRelevant` (Algorithm 3.4, blue curve) and the remaining minimisation (black curve). Analogous to Figure 3.5, each x value represents one SAT instance and the curves are stacked. The y–axis shows the runtime in relation to the complete simplification. Instances are plotted in decreasing order of their relative runtime for AB^*.

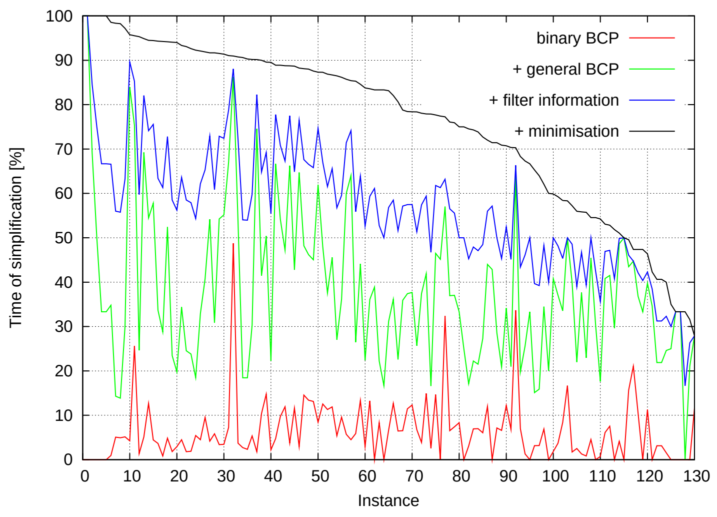


Figure 3.6: Relative runtime for different parts of AB^*

The depicted plot zooms in on the first 130 instances with the highest relative runtime. For most of these instances, the propagation of binary clauses requires the least runtime. This is opposed to the removal of literals. The propagation of non–binary clauses increases the runtime significantly. This effect is also due to having separate watcher lists for binary clauses. As already mentioned above, it has to be considered that the propagation of binary clauses benefits from hyper–binary resolution, which is applied in the remaining parts. The black curve also indicates that for 10% of the instances (30 of 300), the application of AB^* requires more than 90% of the total runtime. Figures 3.5 and 3.6 motivate a better differentiation for different SAT instances. This includes the selection of clauses for which AB^* is applied at all and, moreover, to what extent AB^* is applied for each clause.

Hyper–binary resolution in AB^*

The influence of hyper–binary resolution can be observed in Figure 3.4. In Figure 3.7, we consider the effect of hyper–binary resolution in more detail for different configurations. The y–axis indicates the average number of binary clauses that could be added for each clause $C^\#$ that was inspected by AB^*. A configuration is represented by one plotted curve. A point (x, y) in the plot indicates that for x SAT instances, at least y binary clauses could be added per run of AB^*. The plot zooms in on the first 100 instances and is split into two pieces to apply different scales on the y–axes. The four configurations can be distinguished by three parameters.

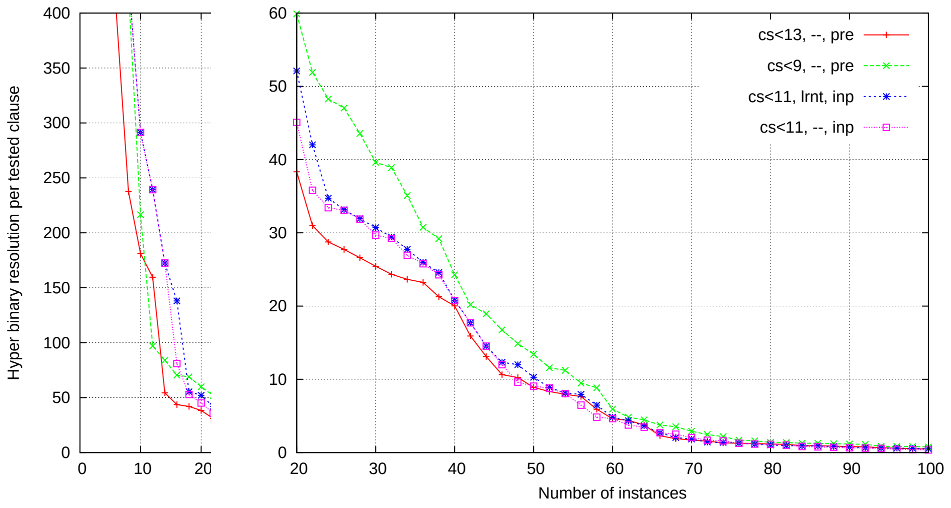


Figure 3.7: Hyper–binary resolution per run of AB^*

The parameter $cs < n$ states the maximal clause size for which AB^* may be applied. The last parameter indicates if simplification is applied as preprocessing only (*pre*) or also in between CDCL searches (*inp*). If inprocessing is applied, the second parameter states whether learnt clauses are also considered for AB^* (*lrnt*).

For all configurations, the number of added binary clauses is quite high for at least 50 of 300 tested instances. Consider the first two configurations (red and green), which only differ in the maximal clause length. The configuration that applies AB^* for clauses with less than nine literals exhibits a clearly higher application of hyper–binary resolution. This is opposed to the removal of literals per inspected clause, as depicted in Figure 3.4. The third and fourth configurations (blue and magenta) apply all simplification techniques within preprocessing and inprocessing. They only differ in the treatment of learnt clauses. However, the additional application of AB^* for learnt clauses does not cause an observable effect for hyper–binary resolution. A reason for the similar behaviour of both configurations is due to the restriction of the clause size $cs < 11$. Many learnt clauses have more than 11 literals. However, on average, AB^* still inspects around $20,000$ clauses ($\approx 14\%$) more when comparing the third to the fourth configuration.

For the instances in the left–hand plot of Figure 3.7, the number of added binary clauses is surprisingly high for all configurations. Most of these instances stem from the domain of planning and are translated into SAT by using the approach of Rintanen *et al.* [RHN06]. Consider the planning instance *grid-strips-grid-y-4.025* with $2,464,339$ clauses. The set of clauses can basically be subdivided into $2,420,825$ binary clauses ($\approx 98.2\%$) and $39,450$ clauses that contain at least ten literals ($\approx 1.6\%$). The application of AB^* within the third configuration (blue) adds $11,182,130$ new binary clauses. Since AB^* is invoked for 7143 clauses the achieved ratio is 1565. Even though the entire simplification reduces the set of clauses by more than 98%, solving is faster for this instance if no simplification is applied at all.

The sheer amount of binary clauses that may be added by AB^* suggests a particular marking of these hyper–binary clauses. This allows the CDCL solver to optionally ignore such clauses. However, the mark has to be removed if a hyper–binary clause is used for another simplification. This applies, for example, whenever a non–redundant clause is found to be subsumed by a hyper–binary clause. It is remarkable that in most configurations, more than 20% of the binary clauses that are added during AB^* are eventually upgraded for 40 of 300 instances. This indicates that the application of hyper–binary resolution can be useful for other simplification techniques. Note that the percentage of upgraded binary clauses does not reflect the benefits for unit propagation.

Figure 3.8 illustrates the impact of hyper–binary resolution within AB^* on equivalence reasoning. A point (x, y) in the plot indicates that for x of 300 instances, at least y percent of variables could be replaced by equivalence reasoning. Two configurations for preprocessing are compared, which apply equivalence reasoning and AB^* only. They only differ in the additional application of hyper–binary resolution. For these configurations the effect of hyper–binary resolution for equivalence reasoning is clearly observable. Note, however, that variable elimination can often achieve a much more extensive removal of variables.

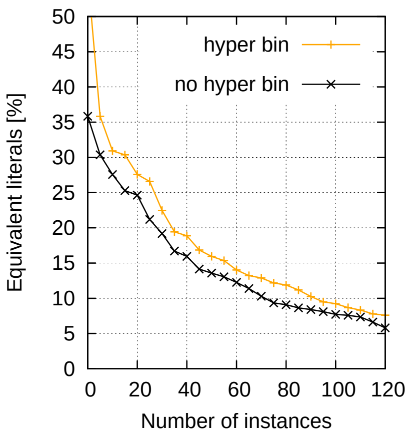


Figure 3.8: AB^* and equivalence reasoning

Runtime comparison

To evaluate the effect of AB^* on the total runtime of a SAT solver, we have run several different configurations on the 300 benchmarks. We have already pointed out that a good choice of the clauses that are inspected by AB^* can show significant differences. This can also be observed in the performance of different configurations. The best configurations apply AB^* in between CDCL searches (at inprocessing) but not within preprocessing. This can be explained by the fact that within preprocessing, it is not clear which clauses may be important constraints for the solver. At inprocessing, this is different. We implemented the following heuristic: As described in Section 2.2.5, modern CDCL solvers apply the phase saving heuristic that caches the previously assigned value for each variable. A decision variable will always be assigned to its cached value. The cached values can be interpreted as complete assignment (cf. [Kot10a, ALMS11]). Consequently, there are clauses that are falsified by this (implicit) assignment or satisfied by only a few literals. These clauses exhibit important constraints for the solver since they are likely to be conflicting in subsequent searches. At inprocessing, only these clauses are inspected by AB^* in order to tighten important constraints. The restriction to clauses that are falsified by the implicit assignment turned out to be the best. By using this selective heuristic for AB^*, it is even better to consider both original and learnt clauses.

Figure 3.9 compares three configurations with pure CDCL solving. A point (x, y) in the cactus plot indicates that x instances can be solved within y seconds per instance. On the one hand, the presented configurations are

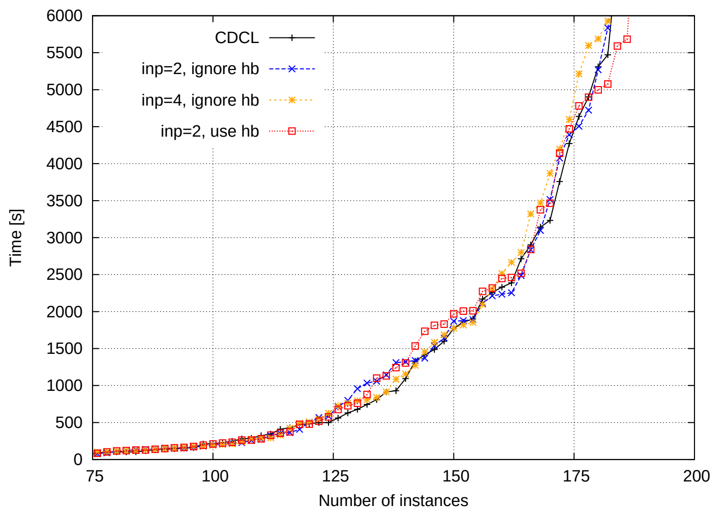


Figure 3.9: Time for solving

comparable to pure CDCL. On the other hand, no significant improvement can be observed for the complete set of benchmarks. All three configurations apply AB^* with the described clause selection heuristic for original and learnt clauses. The parameter $inp = n$ indicates that simplification is frequently applied after $n \cdot |\mathcal{F}|$ conflicts in a CDCL search. The second parameter states whether hyper–binary clauses are considered within CDCL. A configuration may use all hyper–binary clauses for BCP (*use hb*) or ignore them within CDCL (*ignore hb*). In both cases, hyper–binary clauses are always stored for subsequent inprocessing. Surprisingly, using all hyper–binary clauses within CDCL does not obviously harm the solver for the tested configurations.

As mentioned above, we implemented the AB^* technique on top of a pure CDCL solver. This allows for a specific evaluation of AB^*. However, according to the results of the SAT competition 2011 [Sat11], highly optimised state–of–the–art solvers clearly perform better on the complete set of benchmarks. The optimisation of AB^* is difficult and time–consuming, since its application within preprocessing and inprocessing entails hundreds of possible options that have to be adjusted and optimised.

The improvement of AB^* on specific domains of SAT instances looks promising. Kullmann provides a set of benchmarks that encode key discovery problems (Advanced Encryption Standard I (AES) benchmarks). Pure CDCL could only solve two of these instances, whereas the application of AB^* solves four AES benchmarks in less than 2000 seconds.

3.5 Summary

In this chapter, we presented extensions to the conflict–driven SAT solving algorithm with clause learning. CDCL is the predominant solving technique for SAT instances that stem from industrial and other real–world applications. The technique is implemented in most state–of–the–art SAT solvers and has been highly optimised within the last decade.

The first presented extension addresses the data structure for storing clauses within a solver. The underlying idea uses the fact that for any clause that is accessed by a CDCL search, one of the two watched literals is always known. This allows for a `XOR` compression of the two watched literals. Considering that industrial SAT instances consist of thousands or sometimes even millions of clauses, the overall compression is considerable with regard to memory consumption and runtime. The effect is evaluated in Section 3.4.

The second extension studies the application of asymmetric branching [PHS08, HS07]. We explain why the success of this simplification technique depends on the order in which literals of a clause are examined. We introduce an approach AB^* to overcome this issue and present the relevant algorithms in detail. Moreover, we prove that it is NP–hard to reduce a clause to minimum size by the application of AB^*.

We further extend the presented approach by incorporating the application of hyper–binary resolution [Bac02a, Bie09a]. Several aspects of AB^* are evaluated in Section 3.4. Even when using AB^* after all other simplification techniques are finished, a considered clause may often be tightened further. In Chapter 8, we indicate some directions for future research that are related to AB^* and simplification in general.

Chapter 4

Beyond Unit Propagation

The tremendous improvement in SAT solving has made SAT solvers a core engine for many real–world applications. Deliberate engineering of the original CDCL algorithm has enhanced state–of–the–art solvers and enabled them to tackle huge and difficult SAT problems. Different types of solvers have been engineered for different types of SAT instances (i.e. real–world, random, handmade). Solvers for real–world applications make the small amount of effort spent on decision making worthwhile, as they can perform searches very rapidly and, in particular, propagate assignments quickly.

The main ingredient of any conflict–driven SAT solver is clearly a fast implementation of BCP. For the bulk of SAT instances, more than 80% of the runtime is spent on unit propagation [MMZ$^+$01, Kot10a]. On the other hand, relatively little computational effort is spent on choosing decision variables. This constitutes ongoing research to improve the speed of BCP [MMZ$^+$01, Bie08b, CHS09]. For CDCL based SAT solvers, BCP corresponds exactly to unit propagation. If all literals of a clause $C \in \mathcal{F}$ but one are falsified by an assignment τ, the remaining literal λ is asserted to be *true* (see Section 2.2.2).

The fundamental unit propagation technique has now been highly optimised [CHS09] and nearly exploited for further speedups. Solver engineering has recently taken the line of making improvements at other stages of solving. As mentioned in the previous chapter, simplification techniques may not only be applied within preprocessing but also in so–called inprocessing in between searches [Bie09a, Bie09b]. Moreover, conflict analysis has been further improved in different ways [ABH$^+$08, HS09, PD10].

In this work, also published in [KK11a], we are aiming for further improvements in propagation from a different angle. Our motivation is to increase the number of implications that follow from one decision. As shown by Williams *et al.* [WGS03a], many industrial SAT instances may only require a very small set of variables that are chosen as decision variables. The difficulty is to find such a set of variables [Int03, DGS07, KKS08b]. However, an increase in the average number of implications that are caused by one decision comes along with an even smaller set of variables that have to be chosen as decision variables. Moreover, a smaller number of decisions and a greater number of implications may reduce the dependency on felicitous branching decisions.

The approach here focuses on clauses that are not unit under the partial assignment τ but may still be used to deduce further assignments by an enhanced version of BCP. It might be the case that the set of unassigned literals $\mathtt{U}_\tau(C)$ of clause C has a common implication λ_p. If so, λ_p can be propagated even though C is not unit. In this process, we utilise all binary clauses of a formula to check whether some literals have common implications. Consider clause $C = (\lambda_1 \vee \lambda_2 \vee \lambda_3 \vee \lambda_4)$ and the partial assignment $\tau = \{\overline{\lambda_1}, \overline{\lambda_2}\}$. Obviously, one of the two literals, λ_3 or λ_4, has to be assigned in order to satisfy clause C. If there are additional binary clauses, such as $C_1 = (\overline{\lambda_3} \vee \lambda_0), C_2 = (\overline{\lambda_4} \vee \lambda_5)$ and $C_3 = (\overline{\lambda_5} \vee \lambda_0)$, any of the assignments λ_1 and λ_2 implies the assignment λ_0 by unit propagation. However, this may happen after several other decisions. An improved version of BCP could assign λ_0 immediately.

In this chapter, we present two approaches to enhancing the widely applied unit propagation technique by the idea explained above. We propose efficient ways to utilise more reasoning in the main component of current SAT solvers, so as to improve the power of BCP. Following the concept of algorithm engineering, the implementation of the first approach motivates the cost–efficient second approach, even though quality decreases. The chapter is organised as follows. In the next section, related work regarding the utilisation of binary clauses is presented. Section 4.2 explains the idea of enhanced propagation in more detail and presents two approaches on how to realise such propagation. In Section 4.3, both approaches are evaluated.

4.1 Related work

The application of more advanced reasoning in SAT solving has been studied in several different contexts [GT93, Bac02b]. The so–called look–ahead solvers [LA97, Heu08a, BHvMW09] aim to improve the quality of branching decisions and thus to guide the solver's search. One of the basic ideas is to propagate both assignments of a variable ν_p before a decision on ν_p is finally made. In doing so, further reductions may be applied to the formula

[LA97, HM04]. As described in Section 3.2.1, this idea can also be applied in preprocessing to learn unit or binary clauses [Ber01]. Look–ahead solvers are particularly successful for handmade and unsatisfiable random instances [Sat11]. Our approach differs in this respect, and unit propagation itself is extended rather than the decision process.

Recent work [ABH$^+$08, PD10, HS09] improves on the quality of clause learning. Pipatsrisawat and Darwiche focus on some particular missed implications that are not caught by unit propagation [PD10]. Conflict analysis is modified to learn some additional clauses in order to improve the quality of subsequent propagation. Our approach aims to tackle the missed implications already at each propagation step.
Our extension of unit propagation is eminently based on the set of binary clauses in a formula. The special treatment of binary clauses is studied in several works [GT93, Bra01, ZS02, Bac02a, BW03, GS05]. Bacchus introduces the concept of hyper–binary resolution in combination with the computation of the complete binary closure. Applying his approach after each decision at each level of the search captures the implications generated by our approach. However, this turns out to be too time consuming for today's SAT problems. See Section 3.3.1 for a more detailed description of hyper–binary resolution, and Section 3.3.2 for an exhaustive and effective application of the technique.

The preprocessor 2-Simplify [Bra01, Bra04] considers the binary implication graph[1] (BIG) for preprocessing. As well as using some techniques such as the detection of equal variables, the method inspects non–binary clauses to check whether the literals of a clause imply a common assignment. This is based on the concept of hyper–resolution that goes back to Robinson [Rob83] and constitutes a special case of hyper–binary resolution, as presented by Bacchus *et al.* [Bac02a, BW03]. However, in 2-Simplify the complete binary closure is created [Bra01], whereas the implementations proposed by Baccchus [Bac02a] and Biere [Bie09a] are more efficient.

4.2 Enhancing BCP

In this chapter, we aim to improve BCP in terms of the number of implications that can be derived from one decision. This is contrary to one common focus in the literature on how to improve the speed of unit propagation [MMZ$^+$01, CHS09]. In particular, since the implementation of Chaff [MMZ$^+$01] and MiniSat [ES03], much effort has been spent to speed up unit propagation. For many industrial SAT instances poor decisions dur-

[1]Note that the general term "implication graph" is ambiguous within different research works in the area of SAT solving. The binary implication graph is not completely related to the implication graph used in CDCL solving (see Section 2.2.4).

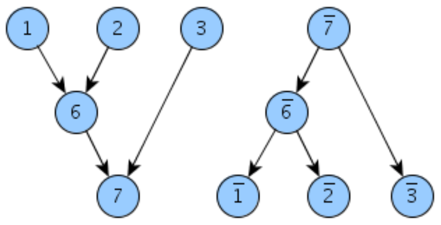
ing CDCL can be compensated by very fast unit propagation. Likewise, sophisticated reasoning is deferred in favour of simple but fast unit propagation. In this section, we introduce two ideas for extending classical BCP. An evaluation and comparison of both implementations is given in Section 4.3.

4.2.1 General observations on clauses and implications

In classical unit propagation, any clause $C_j \in \mathcal{F}$ can imply the value for at most one variable. As described in Section 2.2.2 this applies if all literals of C_j but one are falsified by a partial assignment τ of the variables. However, it may happen that C_j does not have to become unit until the value of a variable can be implied. This goes along with the fact that C_j may directly imply values for more than one variable. Consider the following example: Given a clause $C_5 = (\lambda_1 \vee \lambda_2 \vee \lambda_3 \vee \lambda_4 \vee \lambda_5)$ and a partial assignment τ such that $\overline{\lambda_4}, \overline{\lambda_5} \in \tau$. Apparently, unit propagation cannot be applied for clause C_5 since there is more than one literal unassigned in C_5. Let us also assume there are the binary clauses shown in Figure 4.1 (a).

$$\begin{aligned} C_1 &= (\overline{\lambda_1} \vee \lambda_6) \\ C_2 &= (\overline{\lambda_2} \vee \lambda_6) \\ C_3 &= (\overline{\lambda_3} \vee \lambda_7) \\ C_4 &= (\overline{\lambda_6} \vee \lambda_7) \end{aligned}$$

(a)



(b)

Figure 4.1: Implication graph induced by binary clauses (BIG)

As already stated above, binary clauses constitute a special constraint in CNF. Any binary clause can be understood as two implications. By applying the idea of unit propagation, one can claim that if one of the two literals is *false*, the value of the other variable is implied. This constitutes a BIG as used by Aspvall *et al.* [APT79] to solve 2–CNF formulae. Each variable $\nu_p \in \mathcal{V}$ in the formula is represented by two vertices, p and $\overline{p}$, in the graph, one for the state of ν_p being *true* (p) and the second for the state of ν_p being *false* ($\overline{p}$). Each binary clause $(\lambda_p \vee \lambda_q)$ induces two directed edges $(\overline{p} \to q), (\overline{q} \to p)$ as shown in Figure 4.1 (b). With this, a BIG contains complementary components that may be connected or not.

Consider clause C_5 from above. We know that one of the three literals λ_1, λ_2 and λ_3 has to fulfil C_5 since the others are falsified under τ. Considering the BIG, all three literals (λ_1, λ_2 and λ_3) imply literal λ_7, since there is a chain of implications (a path) from all three vertices $1, 2, 3$ to 7, and thus $(\lambda_1 \vee \lambda_2 \vee \lambda_3) \xrightarrow{\text{UP2}} \lambda_7$. Hence, the partial assignment τ implies the assignment λ_7, $\tau \xrightarrow{\text{UP2}} \lambda_7$. We generalise the idea in the following definition.

Definition 1. *Let $C^* \in \mathcal{F}$ be a clause that is not satisfied by the partial assignment τ, so $\mathtt{T}_\tau(C^*) = \emptyset$. An unassigned literal $\lambda_i \notin \tau$ (also $\overline{\lambda_i} \notin \tau$) is called an inevitable assignment if $\mathtt{U}_\tau(C^*) \xrightarrow{\text{UP2}} \lambda_i$. In other words, each literal in $\mathtt{U}_\tau(C^*)$ implies the assignment λ_i by the propagation of binary clauses $\in \mathcal{F}_2$.*

In this example, clause C_5 is the triggering clause C^* whose literals can be split into the two sets $\mathtt{F}_\tau(C^*)$ and $\mathtt{U}_\tau(C^*)$. As defined in Section 2.1, $\mathtt{F}_\tau(C^*)$ contains all literals that are falsified by the partial assignment τ (λ_4, λ_5) and $\mathtt{U}_\tau(C^*)$ contains those for which the variables are unassigned $(\lambda_1, \lambda_2, \lambda_3)$. Clearly, a triggering clause C^* can never contain a literal that is *true* under τ and thus $\mathtt{T}_\tau(C^*) = \emptyset$. Moreover, $|\mathtt{U}_\tau(C^*)| > 1$ since $|\mathtt{U}_\tau(C^*)| = 1$ constitutes the specification of common unit propagation. The clauses C^*, where an implication $\mathtt{U}_\tau(C^*) \xrightarrow{\text{UP2}} \lambda_x$ can be deduced without any falsified literals ($\mathtt{F}_\tau(C^*) = \emptyset$), can be detected by preprocessing techniques that utilise hyper–resolution [Bra01, BW03].

In the next two subsections (4.2.2 and 4.2.3), we present two ideas on how to recognise the potential for inevitable implications more or less efficiently. In Section 4.2.4, we describe how to incorporate such implications within a CDCL SAT solver. The general course of actions for detecting an inevitable implication during BCP can be summarised by the following four steps:

1. Restrict the set of all clauses $\mathcal{F}$ to a reasonable subset $\mathcal{F}_T \subseteq \mathcal{F}$, which will be considered as triggering clauses (4.2.5).

2. Detect a clause $C^{*'} \in \mathcal{F}_T$ in the formula that may be used to trigger inevitable implications (Sections 4.2.2 and 4.2.3).

3. Determine at least one inevitable implication triggered by $C^{*'}$. It may be more than one implicant or none if the previous step allows for false positives (4.2.4).

4. Apply the additional implication without elementary changes in the data structure (4.2.4).

Detecting cases of inevitable implications allows for further implications that are beyond unit propagation. However, BCP is a highly critical part in CDCL solving and therefore requires fast execution.

4.2.2 A matrix–based approach

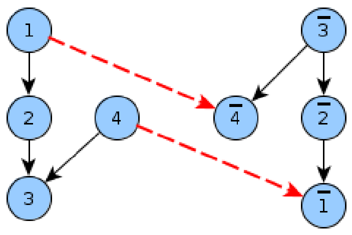
To enhance BCP as described above, the algorithm has to determine if all unassigned variables of a given clause have some common implication. This is equal to a reachability problem in the BIG. Additional computation, e.g. breadth–first search, during BCP is beyond question. Clearly, an adjacency matrix of vertices (i.e. literals) could allow for very rapid computation. However, industrial SAT problems can have up to 10 million variables, which clearly makes a quadratic matrix infeasible. To cope with a high number of literals by allowing random matrix access, we formulate the following property:

Property 3. *Given a directed acyclic graph $G = (V_G, E_G)$, two vertices $a, b \in V_G$ reach a common vertex iff a sink $s \in V_G$ and two paths $(a \rightharpoonup s)$ and $(b \rightharpoonup s)$ exist. A sink is any vertex without outgoing edges.*

We denote the set of all sinks in G as $\sigma \subseteq V_G$. With Property 3, reachability information is only required for pairs of vertices $a, s \in V_G$ where one vertex is a sink $(s \in \sigma)$ and the other is an inner vertex $(a \in V_G \setminus \sigma)$. Recall that a path $(a \rightharpoonup s) \in G$ represents a binary implication $\lambda_a \xrightarrow{\text{UP2}} \lambda_s$.

Although Property 3 is evident, in practice, this already drastically reduces the size of a reachability matrix (cf. Section 4.3). Moreover, for real–world SAT problems, the BIG often decomposes into several (disconnected) components $\gamma_j \in V$. Hence, the functionality of the adjacency matrix can be achieved by holding several independent $n_j \times s_j$ matrices, with j being the index, s_j the number of sinks and n_j the number of inner vertices of the j-th component γ_j. We refer to the component in which a literal λ_p is represented as $\gamma(\lambda_p)$. An example of a reachability matrix is given in Figure 4.3 (a). Note that the indices of sinks and inner nodes are consistently assigned within each component.

Property 3 requires an acyclic directed graph. The removal of strongly connected components (SCCs) can be combined with equivalence reasoning. Literals belonging to the same SCC are identical and can be replaced by one representative literal as described by Brafman [Bra01]. Thus by computing SCCs, the algorithm detects equivalent literals and, furthermore, achieves the requirements of Property 3. In general, two vertices i and $\bar{i}$ that represent the states $true$ and $false$ for variable $\nu_i \in \mathcal{V}$ in the formula $\mathcal{F}$ may be contained in two different components. However, it is also possible that i and $\bar{i}$ are contained in the same component γ_j for some variable ν_i. This does not contradict to the fact that γ_j is not strongly connected. See Figure 4.2 for an example.



$$C_1 = (\overline{\lambda_1} \vee \lambda_2)$$
$$C_2 = (\overline{\lambda_2} \vee \lambda_3)$$
$$C_3 = (\overline{\lambda_4} \vee \lambda_3)$$
$$C_4 = (\overline{\lambda_4} \vee \overline{\lambda_1})$$

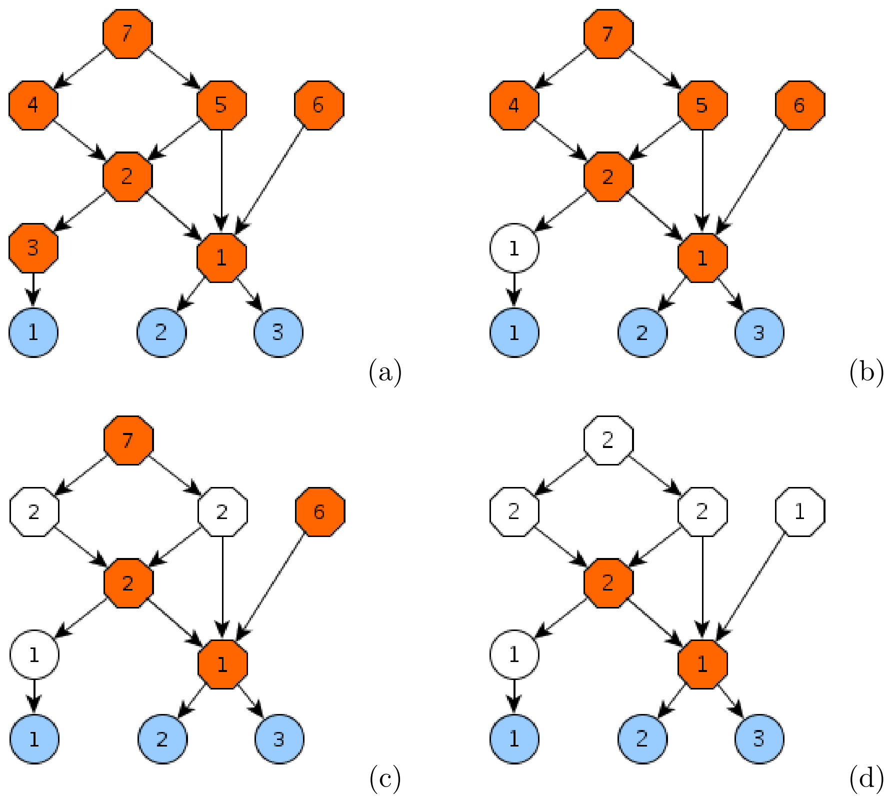
Figure 4.2: Connected components in the BIG. The binary clauses C_1, C_2 and C_3 build a BIG where the two vertices of one variable (representing $true/false$) are placed in different components (black solid edges). Adding clause C_4 causes two more edges (red dashes) that connect the two components. The resulting component is not strongly connected and does not have any directly contradicting path, such as λ_j implying $\overline{\lambda_j}$.

First and foremost, the aim of the matrices is to predict whether or not a set $V'_G \subset V_G$ of vertices reaches a common vertex. At first, it is not important to exactly know the sink that can be reached when starting from vertices in V'_G. Moreover, the matrices are only consulted to provide answers into one direction (e.g. "Do some vertices have a common successor?") but not the other way around ("Is a sink reached by some particular inner vertices?"). This allows for further lossless compression of a reachability matrix by Property 4.

Property 4. *Let $N(v)$ be the adjacent vertices of $v \in V_G$. If any inner vertex $v \in V_G \setminus \sigma$ reaches exactly the same sinks as one of its successors $w \in N(v)$ then v can adopt the reachability information of w. We call v an epigone of w.*

In particular, with Property 4, any epigone v of w can be reduced: if w is a sink then v becomes a sink with the same sink ID as w. Otherwise, v becomes an inner vertex sharing the same column as w in the reachability matrix. In Figure 4.3 the compression of a reachability matrix is shown by an example. In the uncompressed version (a) on the left, the inner vertex (octagon) with ID 3 reaches only sink 1. As an epigone of sink 1, the inner vertex 3 can be treated as equal and can thus be compressed as depicted in Figure 4.3 (b).

This reduction is applied iteratively. The inner vertices 4 and 5 in the uncompressed version are both epigones of the inner vertex with ID 2 and can thus be treated as equal to 2 (see Figure 4.3 (b) $\rightarrow$ (c)). This detects the inner vertex 7 as an epigone of 2, since all its neighbours are epigones of 2, and hence it only reaches the same sinks as the inner vertex 2 does.



	inner vertices						
sinks↓	1	2	3	4	5	6	7
1		x	x	x	x		x
2	x	x		x	x	x	x
3	x	x		x	x	x	x

(a) uncompressed

	inner	
sinks↓	1	2
1		x
2	x	x
3	x	x

(d) compressed

Figure 4.3: Sink matrix of one component. Inner vertices are drawn as orange octagons; sinks are drawn as light blue ellipses. In (d), the complete compression of the matrix by reduction of epigone vertices is given. Epigone vertices have transparent fill colours.

The complete reduction is depicted in Figure 4.3 (d). Thus the compressed reachability matrix reduces from seven to two columns (Figure 4.3). The compression algorithm is a modified depth–first search procedure, starting from the sinks up the DAG to its roots. It is outlined in Algorithm 4.1.

Algorithm 4.1: Create a compressed reachability matrix

```
   Require Connected subgraph $\gamma$ of the (directed) BIG without
   strongly connected (sub)components
   Return Compressed reachability matrix $M_\gamma$ for $\gamma$
   Function compress ($\gamma$)
       foreach $v \in V_\gamma$ do sinkId($v$) $\leftarrow$ innerId($v$) $\leftarrow \infty$
4      $P \leftarrow \{v \in V_\gamma$ : outdeg($v$) $= 0\}$
       $M_\gamma \leftarrow \emptyset$          | at most $|P|$ rows and $\leq |V_\gamma| - |P|$ columns
       $c \leftarrow r \leftarrow 0$                       | index for columns and rows
7      foreach $v \in P$ do sinkId($v$) $\leftarrow r$; $++r$
8      while $P \neq V_\gamma$ do
9          $v \leftarrow$ choose one from $\{v \in \{V_\gamma \setminus P\}$ : targets($v$) $\subseteq P\}$
10         column($v$) $\leftarrow \vec{0}$                  | new empty matrix column
           foreach $w \in$ targets($v$) do
12             if sinkId($w$) $\neq \infty$ then
                   bit(column($v$), sinkId($w$)) $\leftarrow 1$
               else
15                 column($v$) $\leftarrow$ column($v$) $\cup$ column($w$)
           if only one bit $k$ is set in column($v$) then
17             sinkId($v$) $\leftarrow k$             | $v$ is an epigone of sink $k$
           else if $\exists\ w \in$ targets($v$) : column($v$) = column($w$) then
19             innerId($v$) $\leftarrow$ innerId($w$)         | $v$ is epigone of $w$
           else
21             $M_\gamma \leftarrow M_\gamma \cup$ column($v$)       | append column($v$) to $M_\gamma$
               innerId($v$) $\leftarrow c$; $++c$
23         $P \leftarrow P \cup \{v\}$
       return $M_\gamma$
```

The function compress listed in Algorithm 4.1 expects a connected subcomponent γ of the BIG as input with $V_\gamma \subseteq V_G$ vertices and $E_\gamma \subseteq E_G$ edges. We assume γ to be a directed acyclic graph, since equivalence reasoning has replaced all SCCs. In line 4, the set P is initialised with all sinks of the component (vertices with outdegree 0) and holds all vertices that have already been processed. The compressed matrix M_γ that will be returned has one row for each sink and thus $|P|$ rows in total. The number of columns depends on the number of epigones found but is limited by the number of inner vertices $|V_\gamma| - |P|$. The sink IDs are assigned consecutively in line 7; all other sink IDs are invalid (∞).

The main loop in line 8 handles all unprocessed vertices. An unprocessed vertex v is chosen (line 9) such that all its adjacent target vertices are already finished. Since γ is connected and acyclic, there is always such a vertex unless $P = V_\gamma$. In line 10, a new empty column is created that may or may not be appended to the resulting matrix M_γ. Subsequently, all adjacent target vertices w of v are inspected. Where w is a sink or an epigone of a sink with ID k, the k-th bit is set in the new column (line 12). Otherwise, if w is an inner vertex the two columns are unified in line 15.

At this point, the compression checks are applied. If only one bit is set in the new column, then v is initialised to be an epigone of w (line 17) and the column is abolished. Note that at least one bit is set in $\texttt{column}(v)$, since v is not a sink itself. If $\texttt{column}(v)$ is equal to some other column of a target vertex w of v then v becomes an epigone of w and references $\texttt{column}(w)$ (line 19). If all columns of the matrix M_γ were always considered at this point, a possibly better compression of M_γ could be achieved at the expense of runtime. However, checking the adjacent vertices of v can be incorporated efficiently into the initialisation of $\texttt{column}(v)$ in lines 12 and 15. In both cases, it can be checked within the (union) operation whether $\texttt{column}(v)$ is actually changed or if v reaches the same sinks as w. Finally, if no compression is possible, $\texttt{column}(v)$ is appended to the matrix M_γ in line 21. In the end, vertex v is marked to be processed in line 23.

In practice, the reachability matrices can further be compressed by omitting leading and ending blank entries in each column. Two additional integers can give the range for each column. We only apply this for a column if the overhead of the additional integers does not exceed the saving of memory.

Utilising components and matrices

Considering the primal function of a reachability matrix, a given clause C has to be investigated with respect to an inevitable implication. More precisely, C can be split into the two sets of literals $\texttt{F}_\tau(C)$ and $\texttt{U}_\tau(C)$ of *false* or unassigned literals under the present partial assignment τ respectively. The pivotal question is whether C may be used to trigger an implied assignment that is beyond unit propagation.

At first stage, all literals in $\texttt{U}_\tau(C)$ have to belong to the same component. Otherwise, there is no chance of finding a common implication that can be reached by paths from all literals of $\texttt{U}_\tau(C)$. Secondly, the reachability matrix of the corresponding component can be utilised to check whether all literals in $\mathtt{U}_{\mathcal{T}}(C)$ reach at least one common sink. The worst case for this check may require an almost complete comparison of $|\mathtt{U}_{\mathcal{T}}(C)|$ columns of the matrix. However, if one literal in $\mathtt{U}_{\mathcal{T}}(C)$ is a sink itself, the worst case scenario reduces to $|\mathtt{U}_{\mathcal{T}}(C)|$ bit operations. Due to the compression of matrices many literals are marked as sinks in practice. Thus it is very likely to have one sink within a large set $\mathtt{U}_{\mathcal{T}}(C)$. Note that the matrix may still be misleading, since it may happen that all common successors are already assigned. This issue is discussed in Section 4.2.4.

Maintenance of reachability matrices

Before the initialisation of the solver, the vertex of each literal λ_i is assigned to its own component $\gamma(\lambda_i)$. Whenever a binary clause $(\lambda_i \vee \lambda_j)$ is added, the two components $\gamma(\overline{\lambda_i})$ and $\gamma(\lambda_j)$, of the related vertices are merged if they are different. Analogously, the two complementary components $\gamma(\lambda_i)$ and $\gamma(\overline{\lambda_j})$ may be merged. A merge can be done in constant time by holding a first and last vertex for each component. The affected components are marked as being "dirty". Dirty components are updated periodically. Therefore, newly created SCCs are removed as described above. Subsequently, a new matrix is computed which requires one depth–first search execution on this component as listed in Algorithm 4.1. Note that components may also be split when vertices are removed from the graph. This applies whenever a unit clause has been learnt. Out–of–date information about the reachability of variables may generate incorrect information about inevitable implications. However, this is caught by the computation for an explicit inevitable implicant as described in Section 4.2.4.

4.2.3 A convenient alternative

The reachability matrix approach constitutes a feasible way to enhance the widely used unit propagation. However, the maintenance of components and their matrices requires considerable computational effort. Moreover, the compression factor varies for different instances. In this subsection, we present an alternative method that approximates the reachability matrix in a practical sense but considerably outperforms the previous approach.

The basic idea is to cache reachability information on–the–fly while usual unit propagation is performed. Whenever a variable ν_i is assigned a value $b \in \{true, false\}$, all unit implications that are caused by binary clauses are propagated first. In this step, values are assigned to exactly those variables ν_j whose corresponding vertices j or $\overline{j}$ are successors of i (if $b = true$) or $\overline{i}$ (if $b = false$) in the BIG. Due to the way how edges are created in the BIG we know that whenever there is a path from vertex i to j, there is also a path from $\overline{j}$ to vertex $\overline{i}$. Hence, when the unit propagation of binary clauses starts from λ_i (i.e. vertex i), we initialise a *sink–tag* with opposite polarity $\overline{\lambda_i}$. For each literal $\lambda_i \xrightarrow{\text{UP2}} \lambda_j$ (i.e. vertex j) that is implied, we mark the opposite vertex $\overline{j}$ with the current sink–tag. In doing so, all vertices in the graph that have a path to vertex $\overline{i}$ are marked with the corresponding sink–tag to store the implication $\overline{\lambda_j} \xrightarrow{\text{UP2}} \overline{\lambda_i}$.

The idea of sink–tags extends the concept of binary dominators introduced by Biere for lazy hyper–binary resolution [Bie09a, Bac02a, BW03]. However, dominators are attached to variables instead of literals. See Section 3.3.1 for a more detailed description of hyper–binary resolution.

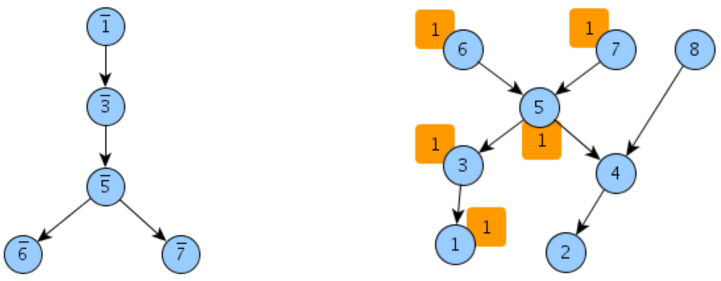
Algorithm 4.2 outlines the procedure for propagating binary clauses immediately after assigning a variable value during a search. Figure 4.4 illustrates this approach for one example. Assume the following binary clauses are given: $C_1 = (\overline{\lambda_3} \vee \lambda_1)$, $C_2 = (\overline{\lambda_5} \vee \lambda_3)$, $C_3 = (\overline{\lambda_6} \vee \lambda_5)$ and $C_4 = (\overline{\lambda_7} \vee \lambda_5)$. When assigning literal $\overline{\lambda_1}$ (i.e. the value $false$ to variable ν_1), the propagation of binary clauses implies the assignment $\overline{\lambda_1} \xrightarrow{\text{UP2}} \overline{\lambda_3}$ by clause C_1. In doing so, the solver gets to know that an assignment of λ_3 implies the assignment $\lambda_3 \xrightarrow{\text{UP2}} \lambda_1$ due to the characteristic of binary clauses. This information is detected on–the–fly during propagation and is stored as reachability information for literal λ_3 by setting its sink–tag to λ_1.

With the assignment of $\overline{\lambda_3}$, clause C_2 asserts the assignment $\overline{\lambda_3} \xrightarrow{\text{UP2}} \overline{\lambda_5}$ and thereby detects a path from vertex $\overline{1}$ to vertex $\overline{5}$. At the same time, the solver gets to know that there is also a path from vertex 5 to vertex 1 (via vertex 3), and thus detects the implication $\lambda_5 \xrightarrow{\text{UP2}} \lambda_3 \xrightarrow{\text{UP2}} \lambda_1$ since propagation started with literal $\overline{\lambda_1}$ and only binary clauses are considered so far. Hence, literal λ_5 stores the sink–tag λ_1. The assignments $\overline{\lambda_6}$ and $\overline{\lambda_7}$ asserted by clauses C_3 and C_4 detect the implications $\lambda_6 \xrightarrow{\text{UP2}} \lambda_1$ and $\lambda_7 \xrightarrow{\text{UP2}} \lambda_1$ analogously. The sink–tags for λ_6 and λ_7 keep this information.

The left–hand side of Figure 4.4 shows the propagation of binary clauses using the clauses of the example above. The propagation detects a subtree of the BIG. On the right–hand side of Figure 4.4, the reachability information for the BIG is depicted. The propagation of $\overline{\lambda_1}$ assigns sink–tag λ_1 to the vertices shown in the right graph.

At a later point in solving, the question may arise whether literals λ_6 and λ_7 have a common binary implication, i.e. whether vertices 6 and 7 have a common successor in the BIG. The sink–tags would indicate the implications $\lambda_6 \xrightarrow{\text{UP2}} \lambda_1$ and $\lambda_7 \xrightarrow{\text{UP2}} \lambda_1$. However, when asking about a common successor of vertices 7 and 8, the sink–tag approach would miss the common successor unless the sink–tags are changed by unit propagation starting from $\overline{\lambda_2}$ or $\overline{\lambda_4}$.

Clearly, the sink–tag approach only caches one possible target that can be reached by a vertex. Furthermore, the target does not necessarily have to be a real sink. It may be any vertex in the graph. As a consequence, this approach does not require the removal or detection of SCCs. Note, that the sink–tag λ_i for a literal λ_j is stored in the hope that it will be useful at a later point in the search, when λ_j is unassigned by a different partial assignment. However, it can additionally be used to detect hyper–binary resolution as described by Biere [Bie09a] and explained in Section 3.3.1.



(a) Unit propagation of binary clauses (b) Sink–tags set during BCP

Figure 4.4: The left graph shows the implications of normal unit propagation of binary clauses when assigning literal $\overline{\lambda_1}$. During propagation, the sink–tag λ_1 can be set in the BIG for all literals that are complementary to the literals reached by propagation.

Algorithm 4.2 formalises the procedure of assigning sink–tags during propagation, as described above. Given a literal λ_p to be assigned, the function propagates all binary implications immediately. In line 6, the sink–tag T is initialised to the complement of the propagated literal, $\overline{\lambda_p}$. As long as there are literals to be propagated, the next literal λ_q is dequeued in line 8. The complement of each propagated literal is assigned a possibly new sink–tag value in line 10. This is because the implication $\lambda_p \xrightarrow{\text{UP2}} \lambda_q$ also indicates the existence of the implication $\overline{\lambda_q} \xrightarrow{\text{UP2}} \overline{\lambda_p} = T$. This information may be used at a later point in solving.

Each binary clause containing literal $\overline{\lambda_q}$ is considered for propagation in line 11. If a conflict arises in line 12, the function returns an empty set and the unit clause $(\overline{\lambda_p})$ can be learnt outside. If λ_k is unassigned (line 13), the current binary clause asserts the assignment λ_k and continues propagation. In line 16, the non–empty set R of all propagated literals is returned.

Algorithm 4.2: Assignment of a variable and caching of sink–tags

```
   Require Literal $\lambda_p$ that has to become true, partial assignment $\tau$
   Return A set of literals $R$ that are assigned or $\emptyset$ if a conflict arises
   Function assign ($\lambda_p$)
       $\tau \leftarrow \tau \cup \lambda_p$                          | assign literal $\lambda_p$
       $Q \leftarrow \{\lambda_p\}$                            | initialise queue
6      $T \leftarrow \overline{\lambda_p}$                          | initialise sink-tag
       while $Q \neq \emptyset$ do
8          $\lambda_q \leftarrow Q$.dequeue()             | remove next element of $Q$
           $R \rightarrow R \cup \lambda_q$        | resulting set of assigned literals
10         sinkTag($\overline{\lambda_q}$) $\leftarrow T$                | set sink-tag for $\overline{\lambda_q}$
11         foreach $\lambda_k : (\overline{\lambda_q} \vee \lambda_k) \in \mathcal{F}_2$ do
12             if $\overline{\lambda_k} \in \tau$ then return $\emptyset$              | conflicting
13             if $\lambda_k \notin \tau$ then
                   $\tau \leftarrow \tau \cup \lambda_k$                | assign literal $\lambda_k$
                   $Q \leftarrow Q \cup \lambda_k$
16     return $R$                     | $R \neq \emptyset$ contains at least $\lambda_p$
```

Utilising sink–tags in BCP

Sink–tags can be utilised in a similar way to components and reachability matrices. However, sink–tags represent and replace both components and bits in a reachability matrix. We observe the following property:

Property 5. *Whenever unit propagation is completed, the sink–tag λ_i of an unassigned literal $\lambda_j \notin \tau$ (and also $\overline{\lambda_j} \notin \tau$) can never be falsified by the current partial assignment τ. We have $\overline{\lambda_i} \notin \tau$.*

Proof. Assume the contrary, that $\overline{\lambda_i} \in \tau$ and λ_i is a sink–tag for an unassigned literal λ_j. Since literal λ_j is unassigned, we have $\lambda_j \notin \tau$ and $\overline{\lambda_j} \notin \tau$. Due to the initialisation of sink–tags, there is a binary implication $\lambda_j \xrightarrow{\text{UP2}} \lambda_i$, and due to the nature of binary clauses, there is also an implication $\overline{\lambda_i} \xrightarrow{\text{UP2}} \overline{\lambda_j}$. Thus $\tau \xrightarrow{\text{UP2}} \overline{\lambda_j}$ causes $\overline{\lambda_j} \in \tau$ and contradicts the assumption. □

Note that a valid sink–tag λ_i of an unassigned literal may be *true* by the partial assignment and thus $\lambda_i \in \tau$ is not contradictory.

To check whether a clause C with a set of unassigned literals $\mathtt{U}_\tau(C)$ may trigger some inevitable implications, the sink–tags of the literals in $\mathtt{U}_\tau(C)$ are consulted. If all literals in $\mathtt{U}_\tau(C)$ have the same sink–tag λ_i, an inevitable implication $\mathtt{U}_\tau(C) \xrightarrow{\text{UP2}} \lambda_i$ may be found. We differentiate three approaches:

Optimistic: The state of the sink–tag λ_i is not considered. If λ_i is assigned in τ, the algorithm still applies a breadth–first search to find common binary implications of the unassigned literals in $\mathtt{U}_\tau(C)$.

Pessimistic: The breadth–first search (see Section 4.2.4) for common binary implications of the unassigned literals is only applied if the sink–tag λ_i is unassigned.

Lazy & pessimistic: If the sink–tag λ_i is unassigned, it is taken as the only common implication of the unassigned literals in $\mathtt{U}_\tau(C)$. This option may dismiss valuable information. On the other hand, there is no need for an additional breadth–first search.

The effect of the different heuristic approaches is evaluated in Section 4.3.

4.2.4 Inevitable implications with CDCL

Identifying the possibility for an inevitable implication during SAT solving, however, is an important part of allowing for an extension of common unit propagation. When using components and reachability matrices and the sink–tag approach, both approaches, may indicate an inevitable implication that is not valid or useless at the current state of the solver. This may be due to out–of–date information of components (see Section 4.2.2) or simply a result of partial assignments that are not considered by the BIG. An example is given in Table 4.5.

Partial assignment:	τ	$=$	$\overline{\lambda_1}, \overline{\lambda_2}, \lambda_3$
Candidate clause for inevitable implication:	C^*	$=$	$(\lambda_4 \vee \lambda_5 \vee \lambda_1 \vee \lambda_2)$
Binary clauses:	C_1	$=$	$(\overline{\lambda_4} \vee \lambda_3)$
	C_2	$=$	$(\overline{\lambda_5} \vee \lambda_6)$
	C_3	$=$	$(\overline{\lambda_6} \vee \lambda_3)$

Table 4.5: Inevitable implications as redundant clauses

Clause C^* in Table 4.5 may be detected by any of the approaches described here to trigger an inevitable implication. The partial assignment τ splits C^* into $\mathtt{F}_\tau(C^*) = \lambda_1, \lambda_2$ and $\mathtt{U}_\tau(C^*) = \lambda_4, \lambda_5$. All literals in $\mathtt{U}_\tau(C^*)$ reach a common sink λ_3 in the BIG. However, ν_3 is already assigned the value *true*. Thus the information about further implications is redundant. Whenever the possibility for an inevitable implication is indicated for a particular clause, an additional breadth–first search in the BIG is required to find common non–redundant implications. Even for the pessimistic sink–tag approach where a non–redundant implication is already available, the breadth–first search may be applied additionally. This is because the breadth–first search often reveals several different inevitable implications at once. Only the lazy pessimistic heuristic omits the breadth–first search and one implication is detected for a triggering clause.

Once a non–redundant inevitable implication is known, it is desirable to apply the enhanced propagation without elementary changes in the data structures. Therefore a few resolution steps are applied to generate a new clause which is unit under the current partial assignment τ and hence allows for normal unit propagation. The created clause may be treated as temporary clause and may be removed by the next execution of the garbage collector.

In order to clarify the idea, consider the clause $C_5 = (\lambda_1 \vee \lambda_2 \vee \lambda_3 \vee \lambda_4 \vee \lambda_5)$ from the example in Section 4.2.1, and the binary clauses $C_1 = (\overline{\lambda_1} \vee \lambda_6)$, $C_2 = (\overline{\lambda_2} \vee \lambda_6)$, $C_3 = (\overline{\lambda_3} \vee \lambda_7)$ and $C_4 = (\overline{\lambda_6} \vee \lambda_7)$ (see Figure 4.1). When the method has detected that the partial assignment τ with $\tau = \overline{\lambda_5}, \overline{\lambda_4}$ allows for an inevitable implication of λ_7, an additional clause C_5' is created via resolution, as in hyper–resolution (see Section 3.3.1 and [Bac02a]). The resolution steps are analogous to the path in the implication graph. Resolving clauses C_5 and C_1 on variable ν_1 results in clause $(\lambda_6 \vee \lambda_2 \vee \lambda_3 \vee \lambda_4 \vee \lambda_5)$. Further resolutions with clauses C_2, C_3 and C_4 finally deduce clause $C_5' = (\lambda_7 \vee \lambda_4 \vee \lambda_5)$.

By construction, C_5' contains at most one unassigned literal. However, it may happen that C_5' is already fulfilled as described in Table 4.5. Of course, this can be checked before the actual creation of C_5'. On the other hand, according to Property 5, it can never happen that all literals in C_5' are falsified by τ when C_5' is not created before common unit propagation is completed. The following lemma formalises the realisation of inevitable propagation.

Lemma 4.2.1. *Given the partial assignment τ, if a clause C^* triggers an inevitable assignment $\lambda_i \notin \tau$, a new clause $C^{*'} = (\lambda_i \vee \mathtt{F}_\tau(C^*))$ can be resolved. Clause $C^{*'}$ is unit under τ and asserts the assignment λ_i by common unit propagation.*

Proof. By Definition 1, clause C^* is not satisfied by τ, i.e. $\mathtt{T}_\tau(C^*) = \emptyset$. Due to the inevitable assignment λ_i with clause C^*, for each literal $\lambda_j \in \mathtt{U}_\tau(C^*)$, the implication $\lambda_j \xrightarrow{\text{UP2}} \lambda_i$ holds. Thus, the binary clause $C_j = (\overline{\lambda_j} \cup \lambda_i)$ is implicitly given. Moreover, for each literal $\lambda_j \in \mathtt{U}_\tau(C^*)$, clause C^* can be resolved with the implicit binary clause C_j. This deduces clause $C^{*'} = (\lambda_i \vee \mathtt{F}_\tau(C^*))$. By definition, literal $\lambda_i \notin \tau$ is unassigned and we also have $\overline{\lambda_i} \notin \tau$ due to Property 5. Given that all literals in $\mathtt{F}_\tau(C^*)$ are falsified by τ, clause $C^{*'}$ is unit under the partial assignment τ and thus asserts the assignment λ_i. □

With both approaches, it may happen that, given a clause $C_5 = (\lambda_1 \vee \lambda_2 \vee \lambda_3 \vee \lambda_4 \vee \lambda_5)$ and an assignment τ with $\tau = \overline{\lambda_4}, \overline{\lambda_5}$, the remaining literals share one common implication. Unlike the example that uses the binary clauses from Figure 4.1, consider the binary clauses $C_1 = (\overline{\lambda_1} \vee \lambda_6)$, $C_2 = (\overline{\lambda_2} \vee \lambda_6)$ and $C_6 = (\overline{\lambda_6} \vee \lambda_3)$. Now the common implication of all unassigned literals $\mathtt{U}_\tau(C_5)$ is equal to one of the literals in $\mathtt{U}_\tau(C_5)$, namely λ_3. Both literals λ_1 and λ_2 imply λ_3. With Lemma 4.2.1, the inevitable assignment λ_3 is realised by deducing the additional clause $C_5' = (\lambda_3 \vee \lambda_4 \vee \lambda_5)$. In this special case, the deduced clause C_5' subsumes the triggering clause C_5 and therefore constitutes a harder constraint than C_5. Since C_5' is a valid constraint, we can simply remove the literals λ_1 and λ_2 from C_5 and thus prune the search space.

Corollary 4.2.2. *Given the partial assignment τ, if a clause C^* triggers an inevitable assignment λ_i and we have $\lambda_i \in \mathtt{U}_\tau(C^*)$, then all literals in $\mathtt{U}_\tau(C^*) \setminus \{\lambda_i\}$ can be removed from C^*.*

Corollary 4.2.2 follows from Lemma 4.2.1 and the fact that $C^{*'} = (\lambda_i \vee \mathtt{F}_\tau(C^*))$ subsumes C^*, since $\lambda_i \in C^*$ and obviously $\mathtt{F}_\tau(C^*) \subset C^*$. The recognition of this kind of subsumption comes without any extra computational effort. Table 4.7 in the next section indicates that these subsumptions are found quite frequently.

4.2.5 Putting it all together

In the previous sections of this chapter, it has been shown how different components of enhancing unit propagation can be realised separately. In this section, the single components are assembled in one approach. At first, we discuss the remaining issue, namely how to choose clauses to be checked for their ability to trigger an inevitable assignment.

Candidates for inevitable assignments

In general, new inevitable assignments may be possible whenever the partial assignment τ is changed by the solver. Hence, for a complete application of enhanced propagation, any assignment or decision of a variable $\nu_i \leftarrow b$ ($b \in \{true, false\}$) would entail a check of all clauses where ν_i is contained with opposite polarity $\neg b$. Furthermore, this would require two lists to be held for each variable with all clauses the variable occurs in. With the learning of additional clauses in any CDCL solver, such lists tend to become extremely long. Considering that the average length of learnt clauses increases for long solver runs, the additional memory and the time overhead to traverse such lists requires too many resources.

A convenient compromise is to use the data structure that is available in a CDCL solver anyway. During normal BCP, each clause has two watched literals [MMZ+01]. Whenever a literal λ_i is falsified, each clause C where λ_i is one of the watched literals is inspected to check whether it is unit under τ. If, for a clause $C \in \mathcal{F}$, another unassigned or *true* literal $\lambda_q \in C$ can be found, λ_q takes over to watch C. Otherwise C is unit under the current assignment τ and unit propagation applies, as described in Section 2.2.2.

At this point, we enhance normal unit propagation. For an inspected clause C that is not yet fulfilled by the partial assignment τ, we check whether the remaining unassigned literals $\in \mathtt{U}_\tau(C)$ belong to the same component, or if the unassigned literals have the same sink–tag, depending on the underlying approach (4.2.2, 4.2.3). For early detection of conflicts, unit propagation is finished until those clauses are finally checked for triggering inevitable assignments.

Algorithm 4.3 outlines the interaction of the separate fragments. Unlike usual unit propagation, there is an extra queue E for enhanced propagation (line 5). Queue Q with literals to propagate is always handled before E (line 7). For the next literal λ_q to be propagated, the list of clauses where $\overline{\lambda_q}$ is one watched literal (line 9 *et seq.*) is traversed. For each inspected clause C^*, λ_p indicates the other watched literal (line 11). If clause C^* is already satisfied by λ_p, the algorithm continues with the next clause (line 12). If there is another literal λ_k in C^* that is either *true* or still unassigned, λ_k becomes the new watched literal for C^* in line 14.

In line 16, clause C^* is tested to see whether it may be a candidate for an inevitable assignment. First of all, C^* must not be satisfied by τ, i.e. $\mathtt{T}_\tau(C^*) = \emptyset$. Depending on the implemented approach (matrix–based, as described in Section 4.2.2, or based on sink–tags as in Section 4.2.3), it is tested to see whether $\mathtt{U}_\tau(C^*)$ may have some common implication. If all literals in $\mathtt{U}_\tau(C^*)$ seem to have a common sink, C^* is put into E for a later inspection.

If clause C^* is unit under τ, literal λ_p is assigned in line 19 and enqueued for propagation. If a conflict arises within the function `assign`, the resulting set P is empty and the unit clause (λ_p) is returned as a conflicting clause in the next line, in order to learn the unit clause $(\overline{\lambda_p})$ outside. Note that if the sink–tag approach is implemented, some sink–tags are updated within the function `assign`, as described in Algorithm 4.2. As in standard BCP, if all literals in the clause C^* are falsified by τ (line 21) a conflict has been found and C^* is returned.

If unit propagation is completed, Algorithm 4.3 checks for inevitable assignments. If queue E is not empty and the next clause C^* in E is not yet satisfied by τ (line 22), then clause C^* is inspected again. Note that even

Algorithm 4.3: Enhanced propagation in CDCL

```
   Require Partial assignment $\tau$, literal $\lambda_s$ to be propagated
   Return A conflicting clause for $\tau$ or $\emptyset$ if propagation was successful
   Function enhancedBCP ($\lambda_s$)
       $Q \leftarrow \{\lambda_s\}$                                   | BCP queue
5      $E \leftarrow \emptyset$        | candidate clauses for enhanced propagation
       while $Q \neq \emptyset$ or $E \neq \emptyset$ do
7          while $Q \neq \emptyset$ do
               $\lambda_q \leftarrow Q$.dequeue()          | next literal to propagate
9              $W_{\overline{q}} \leftarrow$ watchedOf($\overline{\lambda_q}$)        | clauses with watched $\overline{\lambda_q}$
               foreach $C^* \in W_{\overline{q}}$ do
11                 $\lambda_p \leftarrow$ otherWatched($C^*, \overline{\lambda_q}$)
12                 if $\lambda_p \in \tau$ then break         | consider next in $W_{\overline{q}}$
                   if $\exists\ \lambda_k \in \mathtt{U}_\tau(C^*) \cup \mathtt{T}_\tau(C^*) \setminus \{\lambda_p\}$ then
14                     $W_k \leftarrow W_k \cup C^*$                  | link new watched
                       $W_{\overline{q}} \leftarrow W_{\overline{q}} \setminus C^*$
16                     if $C^*$ may trigger an inevitable assignment then
                           $E \leftarrow E \cup C^*$          | Sections 4.2.2, 4.2.3
                   else if $\overline{\lambda_p} \notin \tau$ then
19                     $P \leftarrow$ assign ($\lambda_p$)             | assign literal
                       if $P \neq \emptyset$ then $Q \leftarrow Q \cup P$ else return ($\lambda_p$)
21                 else return $C^*$                     | $C^*$ is conflict
22         if $E \neq \emptyset \wedge \mathtt{T}_\tau(C^*) = \emptyset : C^* \leftarrow E$.dequeue() then
23             $S \leftarrow$ common sinks of literals in $\mathtt{U}_\tau(C^*)$     | BFS or lazy
               foreach $\lambda_i \in S$ do
25                 $\mathcal{F} \leftarrow \mathcal{F} \cup (\lambda_i \vee \mathtt{F}_\tau(C^*))$           | add new clause
26                 if $\lambda_i \in \mathtt{U}_\tau(C^*)$ then mark $C^*$ as subsumed
27                 $P \leftarrow$ assign ($\lambda_i$)                | assign literal
                   if $P \neq \emptyset$ then $Q \leftarrow Q \cup P$ else return ($\lambda_i$)
       return $\emptyset$
```

though it may be ensured that $\mathtt{T}_\tau(C^*) = \emptyset$ before C^* is enqueued into E in line 16, a subsequent assignment may satisfy C^* in the meantime. In line 23, the common sinks of the unassigned literals in $\mathtt{U}_\tau(C^*)$ are computed. Except for the lazy sink–tag heuristic where S contains only the sink–tag this is done by applying a breadth–first search. Recall that the fast checks, which either use the reachability matrix or sink–tags, have been applied in line 16 before the clause was actually enqueued. As described in Section 4.2.4, for all inevitable implications, a new clause is C' created (line 25). By default, C' is marked as redundant and can be removed by later garbage collection. If a newly generated clause C' subsumes the original clause C^*, C^* is marked as redundant in line 26. If C^* is not already redundant, the subsuming clause C' has to become non–redundant to avoid losing constraints. The inevitable implication λ_i is enqueued and assigned in line 27, analogous to line 19.

To realise inevitable implications, additional clauses are created, as described in Section 4.2.4 and shown in line 25 of Algorithm 4.3. This is done to allow for normal conflict analysis, where any implied assignment is expected to have an asserting clause. To avoid the creation of additional clauses, normal conflict analysis could also be adapted to allow for different kinds of asserting clauses. Another approach could aim to create all possible clauses in a preprocessing step, in order to catch all inevitable assignments. However, this could cause the creation of several redundant clauses that are never used for propagation. Recall that the clauses that have to be created are not restricted in any way. In particular, a clause is not only added if the resolvent is a binary clause, as applies in hyper–binary resolution.

4.3 Evaluation

The previous section describes an idea for how to enhance classical unit propagation by the application of inevitable assignments. In Section 4.3.1, we evaluate the two different approaches of using a matrix or sink–tags to detect inevitable assignments during BCP. Afterwards, the different heuristics of the sink–tag approach are examined in more detail.

4.3.1 Matrix versus sink–tag approach

The following comparison between the matrix approach and the sink–tag approach is based on tests that were performed on about 500 benchmarks of the SAT–Race 2008 and the industrial track of the SAT competitions of 2007 and 2009 [Sat11, Sat10]. Trivial instances that can be solved by preprocessing were removed. The average number of variables per instance is 115,500. The different solver versions are all based on the CDCL part of our SAT solver SApperloT without preprocessing.

Qualitative improvement by inevitable implications

The concept of inevitable implications aims for the improvement of BCP. Consequently, the extension of propagation should go along with an increased number of implications that can be deduced from one decision. Figure 4.6 (a) compares the effect of enhanced propagation on the average number of decisions that have to be made to solve an instance. The plot depicts four different configurations: The leftmost bar represents the application of normal unit propagation within CDCL solving. On average, more than 950,000 decisions are required to solve an instance. Using the matrix approach to detect inevitable implications during BCP produces a much smaller average number of required decisions to solve an instance, as illustrated by the second bar. The application of the sink–tag approach also reduces the average number of decisions compared to the application of pure unit propagation. However, both the optimistic and the pessimistic sink–tag heuristic (the third and fourth bars of Figure 4.6 (a)) require clearly more decisions on average than the matrix approach. The different sink–tag heuristics are compared in more detail in Section 4.3.2.

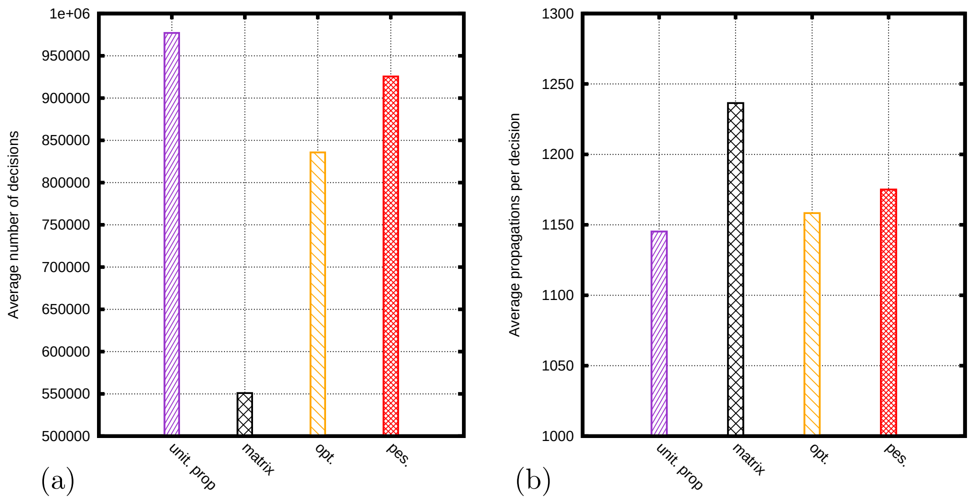


Figure 4.6: The effect of unit propagation alone, the matrix approach, optimistic sink–tags and pessimistic sink–tags on decisions and propagations

The plot in Figure 4.6 (b) illustrates the power of BCP in terms of the average number of propagations that can be deduced by one decision. The number of propagations includes both the number of unit propagations and the number of inevitable implications. The matrix approach (second bar) clearly exhibits the greatest number of propagations per decision when averaging over all instances. The application of sink–tags (third and fourth bars) shows an improvement over unit propagation (first bar) but is far behind the matrix approach.

Table 4.7 compares different aspects of the matrix and sink–tag approaches. Moreover, we distinguish between the optimistic and the pessimistic heuristic of the sink–tag approach. The average and the maximal values that are listed for each configuration and aspect consider all solved instances.

		matrix		opt. sink–tags		pess. sink–tags	
		∅	max	∅	max	∅	max
BCP	inevitable impl.	135,780	2,788,834	81,902	1,782,459	25,981	1,275,854
	inev./dec. [%]	63	1582	34	1341	7	226
generated	long	118,862	2,788,834	72,654	1,755,513	25,761	1,275,854
	binaries	16,816	235,042	9100	152,728	219	6238
	units	101	2722	147	4386	1	247
	subsume	14,038	496,460	701	18,340	640	28,780
	subsume [%]	9.6	70.9	5	96.7	8.1	100

Table 4.7: Enhanced propagation and the creation of clauses

The first row indicates the number of inevitable implications that were detected to solve an instance. The second row relates the number of inevitable implications to the number of decisions needed to solve an instance. For both aspects, the matrix approach clearly outperforms the sink–tag heuristics. On average, an inevitable implication was detected for more than 63% of the decisions. Recall that in all approaches, inevitable implications are only applied when unit propagation has finished. Therefore, each detection of an inevitable implication can be seen as a saved decision in a search.
The subsequent rows state the number of generated clauses needed to realise the extension of BCP, as described in Section 4.2.4. Most created clauses have more than two literals (long clauses in the third row). However, the number of binary clauses created (fourth row) is not negligible. In Section 4.2.4, we point out that a clause that triggers an inevitable implication may be subsumed by the generated clause. The last two rows of Table 4.7 indicate the frequency of the creation of subsuming clauses. Remarkably, on average, at least 5% of the generated clauses are detected as subsuming their triggering clause. In the matrix approach, this applies to almost 10% of the clauses on average.

For the matrix approach, there are some more interesting issues. For each instance, we measured the biggest matrix that was created for a component. On average, this biggest matrix required 870.64 MB without applying Property 4. The application of Property 4 achieves a reduction to an average size of 502.47 MB.

Comparing the runtime of both approaches

Even though the matrix approach clearly outperforms the sink–tag approach in terms of quality, it has a drawback in the costly maintenance of its components and matrices. The cactus plot in Figure 4.8 compares the runtime of both approaches. Each plotted curve represents the performance of one

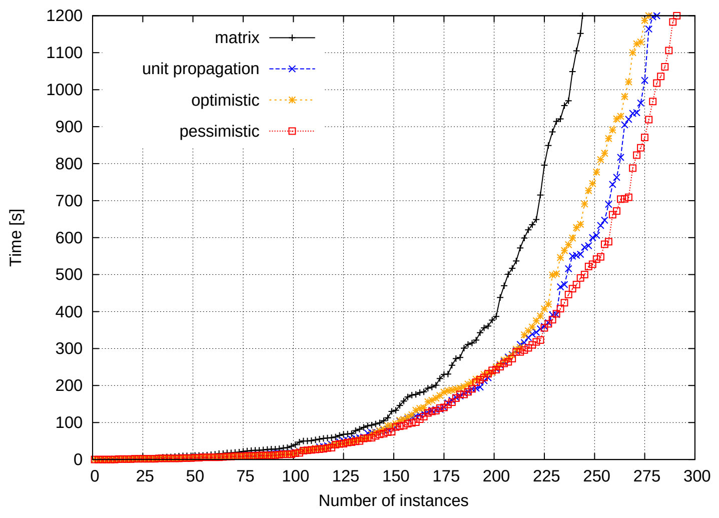


Figure 4.8: Runtime comparison of four configurations

solver configuration. For each configuration, the solved instances are sorted by their runtime in ascending order. Thus, x instances can be solved when the runtime per instance is restricted to y seconds. Note that the plot zooms in on the subset of about 500 instances that could be solved within the time bound of 1200 seconds.

Given the same amount of time, the matrix approach (black curve) clearly solves the least number of instances. Due to the cheaper computation of sink–tags, the alternative approach performs much better. However, the optimistic sink–tag heuristic (orange) cannot compete to standard CDCL (blue). For the pessimistic heuristic (red), it pays to extend propagation only when a valid common successor definitely exists (i.e. unassigned variables that have a common unassigned sink–tag λ_i ($\nu_i \notin \tau$) have at least one common and valid implication λ_i). It clearly solves more instances than standard CDCL with pure unit propagation.

On average, the maintenance of a matrix required 989 depth–first search computations. However, for some instances, the maintenance of the matrix required between 40% and 80% of the total runtime. Even though the improvement in the quality of BCP is clearly observable for the matrix approach, the sink–tag approach is more attractive when it comes to runtime.

4.3.2 Heuristics of the sink–tag approach

The previous evaluation put its focus on the comparison between the matrix and the sink–tag approach. In the following, we study the three different heuristics of the sink–tag approach, *optimistic, pessimistic,* and *lazy and pessimistic* (lazy for short). All tests are based on the set of 300 SAT benchmarks from the SAT competition 2011 [Sat11].

Quality of different heuristics

To evaluate the quality of the different sink–tag heuristics, we analyse different aspects of their application at different decision levels. This is motivated by the fact that all heuristics consider the binary clauses of the formula. The higher the decision level, the more variables assigned by the partial assignment τ and the more binary clauses that are likely to be satisfied by τ. Moreover, the application of a heuristic could easily be restricted to a certain range of decision levels.

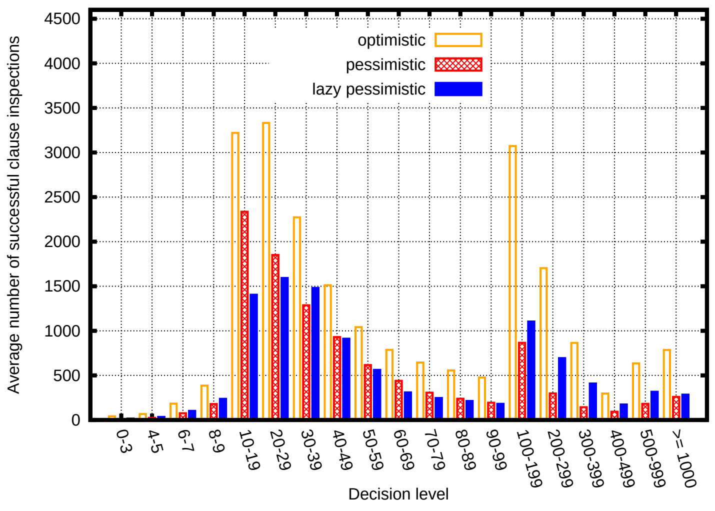


Figure 4.9: Sink–tag heuristics at different decision levels

Figure 4.9 compares the application of the sink–tag heuristics at different decision levels. Decision levels are grouped together, where the group size increases for higher decision levels. Each heuristic is represented by a bar for each group of decision levels. The y–axis indicates the number of clauses whose inspection triggered at least one inevitable implication. If a clause triggered several inevitable implications simultaneously (cf. Algorithm 4.3 from line 23), it is counted only once in this plot.

It is clearly visible that the optimistic heuristic (orange bar) detects the most clauses that trigger inevitable implications, independent of the depth of the search. Moreover, all sink–tag heuristics detect inevitable implications even at higher decision levels. An interesting observation is the difference between the lazy and the pessimistic heuristics. For several decision levels, the lazy heuristic exhibits a higher number than the pessimistic heuristic. The difference between the heuristics reflect the costs of computing all inevitable implications for one triggering clause. To this end, the pessimistic heuristic additionally applies a breadth–first search, which is skipped by the lazy heuristic. In turn, the number of inspected clauses decreases compared to the lazy heuristic due to the effort spent on the breadth–first search. The effect is distinctly recognisable for higher decision levels.

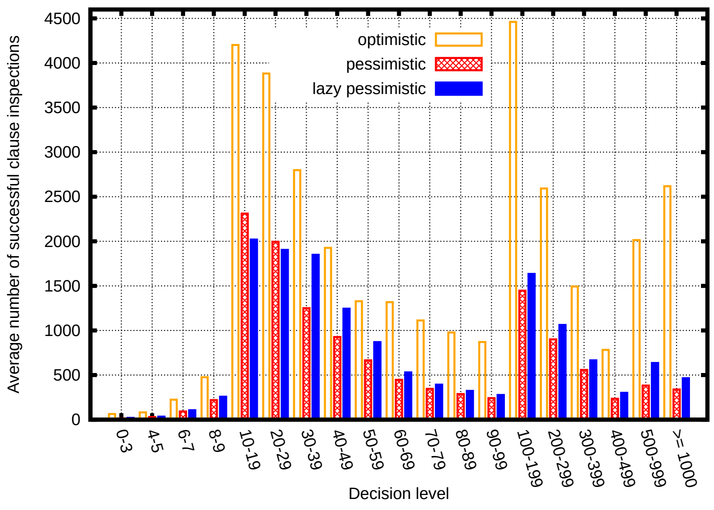


Figure 4.10: Sink–tag heuristics at different decision levels without separate propagation of binary clauses

In Section 4.2.3, we describe how sink–tags are set during the propagation of binary clauses. To enable sink–tags to be used for both, the detection of inevitable implications and for lazy hyper–binary resolution, all binary clauses are propagated first. This is required for the application of lazy hyper–binary resolution [Bie09a]. However, for the pure application of inevitable implications, this is not necessarily required, even though more sink–tags may be set when all binary clauses are propagated first. Figure 4.10 is set out like Figure 4.9 and illustrates the detection of inevitable implications without the separate propagation of binary clauses. Note that the solver configurations depicted in Figure 4.9 propagate binary clauses first but do not apply lazy hyper–binary resolution. Thus the comparison of Figures 4.10 and 4.9 illustrates the effect of propagating binary clauses separately on the detection of inevitable implications. It is evident that, on average, inevitable implications are detected more often at almost all levels. The effect is surprisingly high, especially since the sink–tags are updated more frequently when binary clauses are propagated first. The optimistic heuristic improves considerably for the higher decision levels in particular. Furthermore, the superiority of the lazy heuristic compared to the pessimistic heuristic is observable at almost all decision levels.

Expenses for better quality

The detection of inevitable implications by the optimistic sink–tag heuristic is clearly superior to the pessimistic and the lazy heuristics. However, its application is also more costly in terms of superfluous computations. Figure 4.11 compares the success of the optimistic heuristic to the number of superfluous computations. The plotted values are based on the solver configuration that does not propagate binary clauses separately (represented by the orange bar in Figure 4.10).

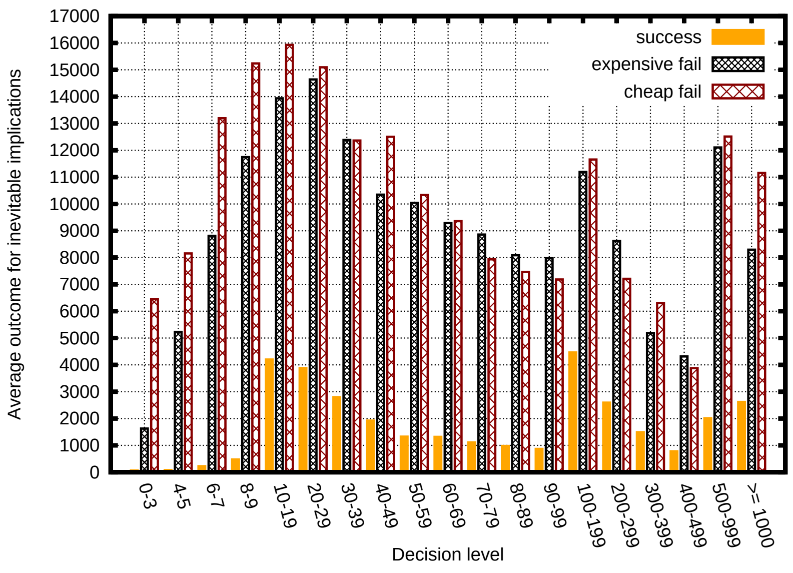


Figure 4.11: Success and failures with optimistic sink–tags

Figure 4.11 uses the grouping of decision levels on the x–axis, as in Figure 4.10, but applies a different scale on the y–axis. The orange bar indicates the number of successful inspections of a clause for which at least one inevitable implication was computed. It is averaged over all solved instances.

The black bar indicates the number of expensive superfluous computations per decision level. In particular, these are the cases where a breadth–first search was applied but no inevitable implication was possible (i.e. $S = \emptyset$ after the BFS in line 23 of Algorithm 4.3). Recall that this may only happen for the optimistic heuristic. The pessimistic heuristic applies BFS only if the sink–tag itself is unassigned and is thus known to be a common implication. The maroon bar states the average number of cheaper superfluous computations. A clause that was enqueued as triggering clause during BCP is found to be satisfied when unit propagation is completed (line 22 of Algorithm 4.3). This case may occur for all sink–tag heuristics but is much less costly than a superfluous BFS computation.

Figure 4.11 shows that cheap failures happen most frequently at most decision levels. However, all sink–tag heuristics have to cope with this overhead, since inevitable implications are not applied before unit propagation is completed. Certainly, the expensive failures are much more critical. The plot indicates that these superfluous computations occur almost as often as the cheap failures. The number of cases where the breadth–first search is successful (orange bar) is considerably smaller. For many levels, more than four out of five BFS computations are superfluous. It is surprising that the failure rate is particularly high at low decision levels. Recall that a successful BFS computation may, however, compute several inevitable implications.
Figure 4.10 illustrates that the optimistic heuristic detects more than double the number of inevitable implications than the other heuristics at many decision levels. However, it has to compensate for the high number of expensive superfluous computations to be convincing in terms of runtime.

Inevitable implications and generated clauses

In the previous analyses, each successful inspection of a clause is counted only once. However, the optimistic and the pessimistic sink–tag heuristics may find several inevitable implications for one triggering clause. On average, the optimistic sink–tag heuristic computes 10.7 and the pessimistic sink–tag heuristic computes seven inevitable implications for one triggering clause. Obviously, the lazy heuristic applies exactly one inevitable implication per triggering clause.

Figure 4.12 considers the application of inevitable implications and the corresponding generation of clauses to realise these implications, as described in Section 4.2.4. As in Figure 4.10, each sink–tag heuristic is represented by one bar. The values are based on the same solver configurations as used in Figure 4.10, where binary clauses are not propagated separately. The x–axis groups different sizes of clauses. A bar at x with a height of y indicates that, on average, y clauses with size x were created by the corresponding heuristic.

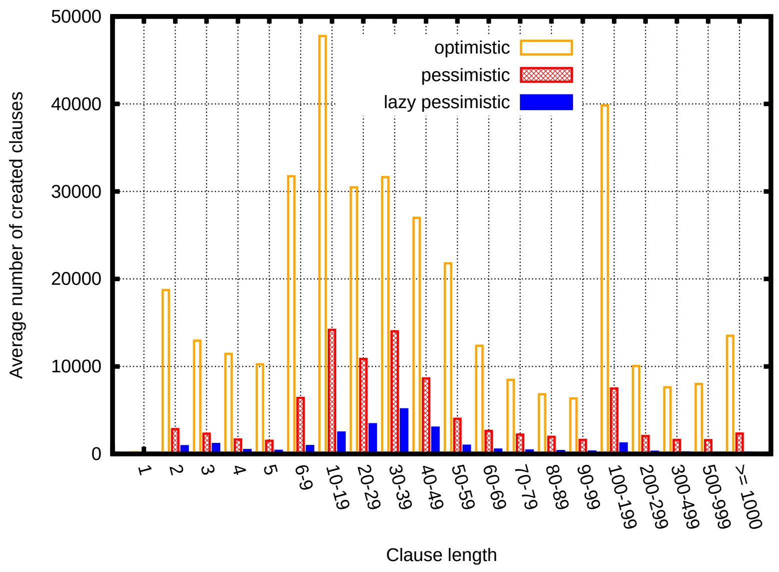


Figure 4.12: Average number and size of clauses to realise inevitable implications

As indicated by the previous analyses, the optimistic sink–tag heuristic detects most clauses that trigger at least one inevitable implication. As Figure 4.12 illustrates, it also deduces the greatest number of inevitable implications and thus creates the most extra clauses. It is observable that remarkably many long clauses are created, especially compared to the lazy heuristic. On the one hand, the importance of a clause can rarely be measured solely by its number of literals. On the other hand, creating a high number of long clauses may harm the solver in terms of resource requirements. The clearly observable differences between the pessimistic and the lazy heuristics reflect the number of inevitable implications that are applied for one triggering clause by the pessimistic heuristic.

Enhanced propagation

The detection of inevitable implications may increase the average number of implications that can be deduced by one decision. In Section 4.3.1, we studied the effect of the matrix approach and the sink–tag approach on the average number of propagations per decision. Figure 4.13 analyses the effect for different configurations and sink–tag heuristics.
The plot is divided into two parts to allow for different scales on the y–axis, which indicates the number of implications per decision. Each solver configuration is represented by one curve. A point (x, y) in the plot states that,

for x instances, at least y implications can be deduced by one decision. To make the differences visible, the plots zoom in on the first 80 of 300 instances.

Figure 4.13 distinguishes between six different solver configurations. The black and the blue curves represent the pure application of unit propagation in CDCL. In the latter configuration (blue), binary clauses are propagated separately before longer clauses are considered for unit propagation. The next three configurations (maroon, red and orange) represent the three sink–tag heuristics (optimistic, pessimistic and lazy) without the separate propagation of binary clauses. The last configuration (purple) represents the lazy sink–tag heuristic where binary clauses are propagated separately.

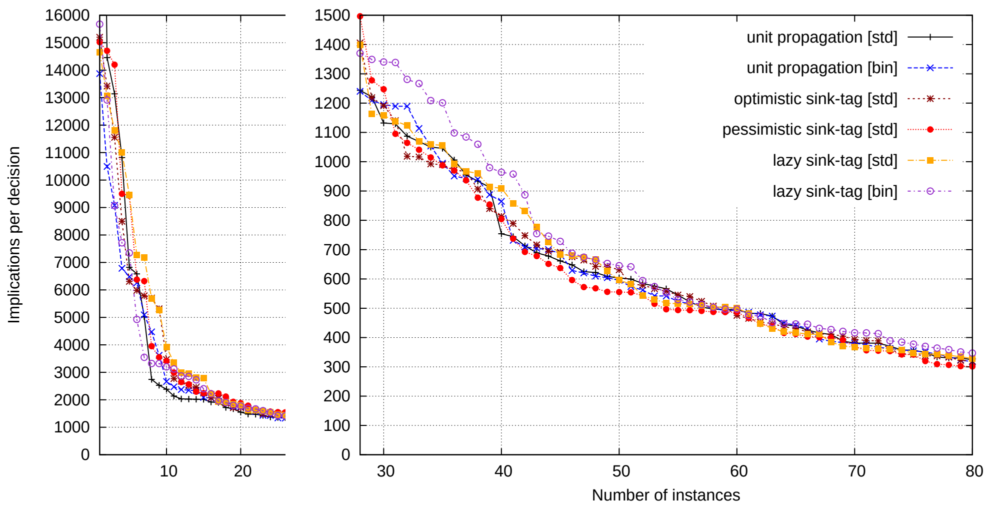


Figure 4.13: Average number of propagations per decision for different configurations and sink–tag heuristics

Both configurations that apply the lazy sink–tag heuristic indicate a high number of propagations per decision. However, the separate propagation of binary clauses exhibits comparably small values for most of the first 25 instances but improves for the later instances (purple). The pure application of unit propagation rarely produces the most propagations per decision, though the optimistic and the pessimistic sink–tag heuristics are often worse.
The plots illustrate that there is no configuration that is superior to all other configurations on the entire benchmark set. However, differences in the plots are not negligible since, on average, they apply for each decision during a search. At some points, the best configuration deduces 200 more implications per decision than the worst configuration at that point. In particular, the advantage of the lazy heuristic over the pessimistic heuristic has to be considered. Even though the latter deduces clearly more inevitable implications

(cf. Figure 4.12), the lazy heuristic detects more clauses that trigger an inevitable implication (cf. Figure 4.10). After all, the lazy heuristic seems to be more powerful due to its faster application.

Runtime

We finally compare the runtime for the application of the different sink–tag heuristics. Figure 4.14 depicts a cactus plot of the same solver configurations that are used in Figure 4.13. The plot omits the instances that have not been solved within a timeout of 8000 seconds.
We tested several different configurations for the three sink–tag heuristics. In particular, we restricted the application of the sink–tag heuristics to certain decision levels. As mentioned at the beginning, all solver configurations do not apply additional inprocessing techniques or special optimisation, in order to focus on the presented techniques themselves and to keep the number of side–effects low.

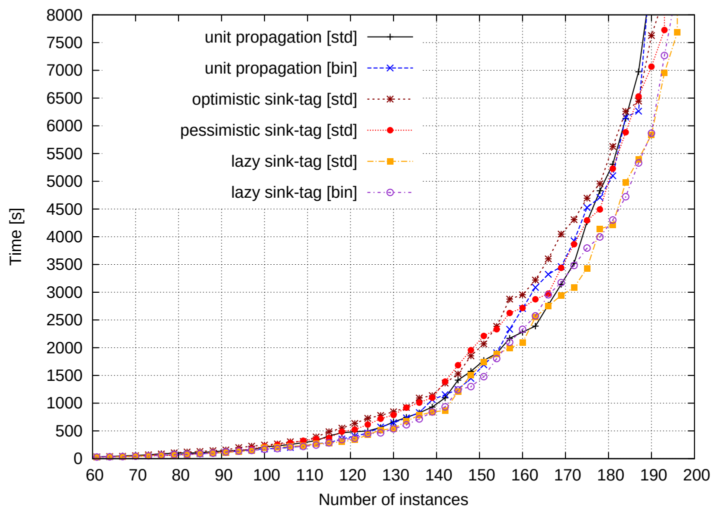


Figure 4.14: Runtime of different heuristics

Overall, the lazy sink–tag heuristics clearly perform best. As Figure 4.14 indicates, the final success is independent of whether binary clauses are propagated separately or not. It is observable that the application of the lazy sink–tag heuristic can clearly improve solving. The pure application of unit propagation (black and blue curves) solves fewer instances for almost all time bounds. Ultimately, the lazy heuristic solves six more instances when each solver is allowed to run for 8000 seconds per instance.

4.4 Summary

Boolean constraint propagation is one of the most critical parts in modern SAT solvers. BCP is equal to unit propagation for almost all state–of–the–art conflict–driven SAT solvers that focus on industrial SAT instances. Most of the runtime for solving an instance is spent on unit propagation.
In this chapter, we proposed an enhancement of classical unit propagation. Unlike unit propagation, a clause may imply an assignment of a variable even though it does not have to be unit under the partial assignment τ. We call an assignment inevitable if all unassigned literals of a clause, which is not satisfied under τ, eventually imply that assignment.

To detect some inevitable implications, we considered the set of binary clauses of a formula. In this chapter, we presented two different approaches for detecting inevitable implications. Both approaches were explained in detail and were evaluated in Section 4.3. The first approach uses a compressed reachability matrix to detect inevitable implications. It is more thorough than the second approach that stores so–called sink–tags during the propagation of binary clauses. These sink–tags can then be used at a later point in search to detect inevitable implications.

The practical evaluation has shown that the first approach is clearly better in terms of quality whilst the latter approach is superior in terms of runtime and resource requirements.
We proposed three different heuristics for the sink–tag approach and analysed their implementations in greater detail. Ultimately, the most superficial heuristic, which we call the lazy sink–tag heuristic, turned out to be the most efficient. Its application clearly improves on the pure application of unit propagation.

The concept of inevitable implications is revisited in Chapter 8, where we conclude the work on the enhancement of unit propagation and give some directions for future research.

Chapter 5

SAT Solving with Reference Points

Conflict–driven SAT solving has proven to be very successful on a wide range of benchmarks. In particular, instances that originate from real–world applications which are modelled as SAT problems are primarily tackled by CDCL solvers (Section 2.2). Many problems, ranging from hardware and software verification [Vel02, IYG$^+$08], planning [KS92] and automotive product configuration [KS00] to haplotype inference in bioinformatics [LMS06] use a state–of–the–art CDCL solver as back–end technology.
Due to the success of CDCL solvers within these areas and the continuous improvements in CDCL implementations [Sat11], a smaller portion of SAT research addresses the question whether a different solving technique can beat CDCL in its main discipline. As already pointed out at the beginning of Chapter 3, small modifications of CDCL are often far more effective in practice than complicated and costly changes to the conflict–driven approach. However, there are approaches that motivate more in–depth reasoning to guide SAT solver searches. Certainly, more reasoning often requires more information and comes along with more complex data structures. In many cases, this may cause strong concepts to be inapplicable in practice.

In contrast to the predominant CDCL technique, Goldberg suggested a new SAT solving approach [Gol08a, Gol08b] that operates on complete assignments of the variables of a CNF formula. Unlike conflict–driven solving, decisions for branching are based on the set of clauses that are falsified by the assignment. The algorithm is therefore called Decision Making with a Reference Point (DMRP). The concept of DMRP connects some aspects of local search heuristics with branching and backtracking searches.

Using a complete assignment — a so–called reference point — for decision making demands more information during the search and hence involves the maintenance of more complex data structures. Therefore the DMRP approach has not managed to match state–of–the–art SAT solvers in industrial applications. In [Kot10a], the DMRP algorithm is analysed from a practical point of view, and we present an implementation that utilises a suitable data structure to realise the DMRP approach in practice. The effect and quality of decisions is comparable to CDCL solving. Moreover, we show how DMRP can be combined with CDCL solving to compete with the performance of state–of–the–art solvers and even improve on some families of industrial instances.

This chapter is organised as follows: In Section 5.1, we describe previous work in the domain of reference points and DMRP solving. Section 5.2 examines the DMRP approach from a practical point of view and we present a competitive implementation for this approach. In Section 5.3, we combine CDCL and DMRP to create a new hybrid approach. An experimental evaluation is presented in Section 5.4.

5.1 Related Work

In this chapter, two different SAT solving approaches are used. The state–of–the–art conflict–driven solving approach (CDCL) and the more recent DMRP approach that operates on complete assignments.

CDCL is based on the GRASP algorithm [MSS99], which extends the original DPLL branch–and–bound procedure [DP60, DLL62] by the idea of learning from conflicting assignments. Moreover, conflicts are analysed to jump over parts of the search space that would cause further conflicts. There are several improvements to the original algorithm. In particular, the two watched literals scheme and the VSIDS (see Section 2.2.5) variable selection heuristic [MMZ$^+$01] allow for a fast execution of unit propagation, since the maintenance of the required data structure can be implemented efficiently. For a detailed description of conflict–driven solving, we refer the reader to Section 2.2.

Complete assignments

Goldberg has analysed the properties and use of complete variable assignments in several works. A complete assignment to the variables of a formula $\mathcal{F}$ is called a point and is referred to as $\mathcal{P}$ throughout this chapter.
Goldberg introduces the notion of *stable point sets* [Gol02]. In this context, the *neighbourhood* of a point is considered. This is done with the aim of restricting the number of points that have to be inspected in order to find a model for $\mathcal{F}$ or to prove unsatisfiability.

The neighbourhood of a point $\mathcal{P}$ is related to clauses that are falsified by $\mathcal{P}$. In order to do so, the set of all possible points $\in \{false, true\}^{|\mathcal{V}|}$ of a formula $\mathcal{F}$ is divided into the set of points $S_{\mathcal{F}}$ that constitute a model for $\mathcal{F}$ and the remaining points $U_{\mathcal{F}}$ that falsify at least one clause of $\mathcal{F}$, such that $S_{\mathcal{F}} \cup U_{\mathcal{F}} = \{false, true\}^{|\mathcal{V}|}$ and $S_{\mathcal{F}} \cap U_{\mathcal{F}} = \emptyset$. Consider a clause $C_{\mathcal{P}}$ that is falsified by $\mathcal{P} \in U_{\mathcal{F}}$, i.e. none of the literals of $C_{\mathcal{P}}$ is contained in $\mathcal{P}$. The 1-neighbourhood of $\mathcal{P}$ defines a set of points $1Nb(\mathcal{P}, C_{\mathcal{P}})$ with respect to clause $C_{\mathcal{P}}$. A point $\mathcal{P}'$ is in the 1-neighbourhood $1Nb(\mathcal{P}, C_{\mathcal{P}})$ of $\mathcal{P}$ if $\mathcal{P}$ and $\mathcal{P}'$ differ in exactly one variable value and $C_{\mathcal{P}}$ is satisfied by $\mathcal{P}'$. If point $\mathcal{P}'$ satisfies all clauses of $\mathcal{F}$, then $\mathcal{P}' \in S_{\mathcal{F}}$ constitutes a model for $\mathcal{F}$. Otherwise, there is at least one clause $C_{\mathcal{P}'}$ that is falsified by $\mathcal{P}' \in U_{\mathcal{F}}$.

To generate a *completed set of points*, the neighbourhood relationship is extended further. A transport function $g : U_{\mathcal{F}} \mapsto \mathcal{F}$ maps each point $\mathcal{P} \in U_{\mathcal{F}}$ to one particular clause $C_{\mathcal{P}}$ that is falsified by the point $\mathcal{P}$. The neighbourhood $N(\mathcal{P}_0)$ of a point $\mathcal{P}_0 \in U_{\mathcal{F}}$ with respect to the transport function g can now be created transitively. For each point, $\mathcal{P}_i \in N(\mathcal{P}_0)$ the 1-neighbourhood of $\mathcal{P}_i$ with respect to clause $g(\mathcal{P}_i)$, $1Nb(\mathcal{P}_i, g(\mathcal{P}_i))$, is merged into $N(\mathcal{P}_0)$. More formally, the set of points $N(\mathcal{P}_0)$ is referred to as being stable with respect to the transport function g if $1Nb(\mathcal{P}_i, g(\mathcal{P}_i)) \subseteq N(\mathcal{P}_0) \; \forall \; \mathcal{P}_i \in N(\mathcal{P}_0)$ and if $N(\mathcal{P}_0) \subseteq U_{\mathcal{F}}$.

A crucial proposition in [Gol02] is that the existence of a stable point set of formula $\mathcal{F}$ proves the unsatisfiability of $\mathcal{F}$. By exploring the neighbourhood of a chosen point, either a model is found or a stable set of points is constituted, which, in turn proves unsatisfiability. Note that we have only roughly outlined the basic idea here, with a modified notation to some extent, and we refer the reader to the original work [Gol02, Gol05] for a detailed description.

Decision making with a reference point

The DMRP approach extends the idea of complete assignments. A set of different points is inspected by a branching algorithm. When using a complete assignment to guide the search for a satisfying model of the formula $\mathcal{F}$, the complete assignment is called a reference point. The DMRP approach and some previous ideas have been proposed by Goldberg in several works [Gol06, Gol08a, Gol08b]. Even though DMRP uses BCP with backtracking and learning from conflicting assignments, it is not a simple variant of CDCL. A crucial difference from CDCL solvers is the ubiquitous existence and use of a reference point. The algorithm aims to modify the current reference point $\mathcal{P}$ to $\mathcal{P}'$ in order to satisfy a clause under consideration. Furthermore, it is important that all clauses satisfied by $\mathcal{P}$ remain satisfied by the modified reference point $\mathcal{P}'$.

Algorithm 5.1: Outline of the DMRP approach

```
Require Formula $\mathcal{F}$ in CNF, a reference point $\mathcal{P}$ and any two
timeout criteria $T_1$ and $T_2$
Return A reference point that satisfies $\mathcal{F}$, UnSat or Unknown.
Function solveDMRP ($\mathcal{F}, \mathcal{P}, T_1, T_2$)
4      $\mathcal{M} \leftarrow \{C \in \mathcal{F} \ : \ C$ falsified by $\mathcal{P}\}$
       while $\neg T_1$ do
6          if $\mathcal{M} = \emptyset$ then return $\mathcal{P}$
7          $C_0 \leftarrow$ remove any clause from $\mathcal{M}$
8          $\mathcal{P}' \leftarrow$ dmrpTryModifyPoint($\mathcal{F} \setminus \mathcal{M}, C_0, \mathcal{P}, T_2$)
9          if $\mathcal{P}'$ = UnSat then
               return UnSat
11         else if $\mathcal{P}'$ = Unknown then
               $\mathcal{M} \leftarrow \mathcal{M} \cup \{C_0\}$                | try another clause
13         else
               $\mathcal{P} \leftarrow \mathcal{P}'$                        | adapt reference point
               $\mathcal{M} \leftarrow \{C \in \mathcal{F} \ : \ C$ falsified by $\mathcal{P}\}$
       return Unknown
```

Algorithm 5.1 gives an overview of the DMRP search, though our notation varies from the original notation of Goldberg [Gol08a]. In line 4, the set of clauses $\mathcal{M}$ that are falsified by the given point is initialised. The algorithm searches for a modification of the point that satisfies all clauses of $\mathcal{F}$. If such a point is found within the given time, it is returned as a model in line 6.

In line 7, a clause C_0 is chosen from the set of currently falsified clauses $\mathcal{M}$. In doing so, C_0 becomes the basis for the next search. One invocation in line 8 of the DMRP subsolver (listed in Algorithm 5.2) considers the subformula $\mathcal{F} \setminus \mathcal{M}$. It aims to find a modified reference point $\mathcal{P}'$ that satisfies all clauses in the subformula $\mathcal{F} \setminus \mathcal{M}$ together with C_0. It may happen that the empty clause is learnt (line 9) or that the search within the DMRP subsolver times out (line 11). The latter case causes the surrounding algorithm to call the DMRP subsolver with another falsified clause in the next pass of the loop.

Note that a modified reference point $\mathcal{P}'$ may have an arbitrary Hamming distance to the previous point $\mathcal{P}$. However, the stepwise modification of $\mathcal{P}$ within the function `dmrpTryModifyPoint` considers different points that differ only in one variable assignment between two consecutive steps. If a point $\mathcal{P}'$ is found by the DMRP subsolver, $\mathcal{P}$ is replaced (in line 13) and set $\mathcal{M}$ is adapted accordingly.

Algorithm 5.2: Subroutine to search for a better reference point

```
Require Subformula $\mathcal{F}$, a clause $C$ that shall be satisfied by a
modification of the current point $\mathcal{P}$, and a timeout criteria $T$
Return A modification of $\mathcal{P}$ that satisfies $\mathcal{F} \cup C$ , or UnSat if the
empty clause is learnt, or Unknown if $T$ is exceeded
Function dmrpTryModifyPoint $(\mathcal{F}, C, \mathcal{P}, T)$
4     $\mathcal{P}_t \leftarrow \mathcal{P}, \quad \mathcal{D} \leftarrow \{C\}$
      while $\neg T$ do
6         if $\mathcal{D} = \emptyset$ then return $\mathcal{P}_t$          | valid modification of $\mathcal{P}$
7         $C \leftarrow$ choose any clause from $\mathcal{D}$
8         $\lambda_q \leftarrow$ choose flip candidate $\lambda_q \in C$        | with restriction
9         $< res, \mathcal{P}_t > \leftarrow$ bcpWithPoint$(\mathcal{F}, \lambda_q, \mathcal{P}_t)$
          while $res =$ Conflict do
11            $lma \leftarrow$ analyseConflict$(\mathcal{F}, res)$
              if $lma = \emptyset$ then return UnSat
13            $\mathcal{P}_t \leftarrow$ backtrackResetPoint$(\mathcal{F}, lma)$
14            $< res, \mathcal{P}_t > \leftarrow$ learnAndPropagate$(\mathcal{F}, lma, \mathcal{P}_t)$
15        $\mathcal{D} \leftarrow \{C \in \mathcal{F} \ : \ C$ falsified by $\mathcal{P}_t\}$
      return Unknown
```

Algorithm 5.2 constitutes the elementary part of the DMRP approach. A given reference point has to be modified to satisfy the given set of clauses $\mathcal{F} \cup C$. Different points are explored by selecting one literal of the current point to be flipped. This has quite some similarity to local search approaches. However, it is crucial that the search for the modified point is applied in such a way that the algorithm may eventually detect that the given set of clauses cannot be satisfied. On the contrary, this does not apply in common local search algorithms.

The search is organised in a branching procedure that allows for backtracking to former states. This is realised by using a trail where different temporarily modified versions of points $\mathcal{P}_t$ are implicitly stored. The stack $\mathcal{D}$ holds all clauses of $\mathcal{F} \cup C$ that are falsified by the current temporary point $\mathcal{P}_t$ (line 4). If a temporary point $\mathcal{P}_t$ satisfies all clauses in $\mathcal{F} \cup C$, it is a valid new point and is returned in line 6. Otherwise, a clause C that is falsified by the temporary point $\mathcal{P}_t$ is chosen by a heuristic in line 7. Based on this clause C, a literal $\lambda_q \in C$ is selected to be flipped within the next temporary point in line 8. A proposed heuristic chooses λ_q from among the literals in C, such that the assignment $\mathcal{P}_t \setminus \{\overline{\lambda_q}\} \cup \{\lambda_q\}$ (where the value of ν_q is flipped) satisfies a maximal number of clauses in $\mathcal{D}$. A literal $\lambda_q \in C$ is only considered for flipping if it has not been already flipped within the current trail. This avoids arbitrary flipping of variable values back and forth, and will be explained in more detail in the next section.

In line 9, the consequences of changing the value of variable ν_q are propagated. This may imply the modification of further variable values. If a conflict arises during propagation, it is analysed in line 11 and a lemma *lma* is created. If the empty clause is learnt, it is not possible to modify $\mathcal{P}$ to satisfy all given clauses. Otherwise, the lemma is used to backtrack to a former point in search (line 13). This resets the temporary point $\mathcal{P}_t$ to a previous state, which also implicitly resets the trail. The lemma is finally added to $\mathcal{F}$ in line 14 and the consequences are propagated. Finally, in line 15, the stack $\mathcal{D}$ is adapted according to the new temporary point $\mathcal{P}_t$. Different aspects of the algorithm and implementation details are handled in the next section.

Boundary points

The concept of *boundary points* was also introduced by Goldberg [Gol09] and analyses another perspective of points and the ability to prove unsatisfiability by the use of point sets. Goldberg and Manolios suggest a template procedure for using boundary points in SAT solving [GM10]. As in DMRP, for a given point $\mathcal{P}$, we consider the set of clauses $\mathcal{M}_{\mathcal{P}} \subseteq \mathcal{F}$ that are falsified by $\mathcal{P}$. The clause set $\mathcal{M}_{\mathcal{P}} \neq \emptyset$ is called a λ_i–boundary point if there is a literal λ_i that is contained in every clause of $\mathcal{M}_{\mathcal{P}}$.
If there is such a λ_i–boundary point for point $\mathcal{P}$ then obviously $\mathcal{P}$ contains literal $\overline{\lambda_i}$. Moreover, the modified point $\mathcal{P}' = \mathcal{P} \setminus \overline{\lambda_i} \cup \lambda_i$ that flips the value for variable ν_i satisfies all clauses in $\mathcal{M}_{\mathcal{P}}$. Furthermore, consider the set of clauses $\mathcal{M}_{\mathcal{P}'}$ that are falsified by the new point $\mathcal{P}'$. If $\mathcal{M}_{\mathcal{P}'}$ is empty, a model of $\mathcal{F}$ is found. Otherwise, $\mathcal{M}_{\mathcal{P}'}$ constitutes a $\overline{\lambda_i}$–boundary point, since we have $\mathcal{M}_{\mathcal{P}} \cap \mathcal{M}_{\mathcal{P}'} = \emptyset$ and thus $\overline{\lambda_i}$ is contained in every clause of $\mathcal{M}_{\mathcal{P}'}$. With this, a λ_i–boundary point is either one flip (of variable ν_i) away from a model of the formula or it has a so–called *twin boundary point*. This observation shows a way of eliminating boundary points.

For the following, we assume formula $\mathcal{F}$ to be unsatisfiable. Consider a point $\mathcal{P}$ with a λ_i–boundary point and its twin, a $\overline{\lambda_i}$–boundary point for $\mathcal{P}'$. The sets $\mathcal{M}_{\mathcal{P}} \cap \mathcal{M}_{\mathcal{P}'} = \emptyset$ are disjoint and non–empty. Choosing one clause from each set $C \in \mathcal{M}_{\mathcal{P}}$ and $C' \in \mathcal{M}_{\mathcal{P}'}$ allows for a resolution of C and C' on variable ν_i. The resolvent is neither contained in $\mathcal{M}_{\mathcal{P}}$ nor in $\mathcal{M}_{\mathcal{P}'}$, and thus eliminates both points.
It can be further proven that adding the resolvents of a resolution proof eventually eliminates all boundary points. Moreover, if an unsatisfiable formula $\mathcal{F}$ has a λ_i–boundary point, then any resolution proof (that $\mathcal{F}$ is unsatisfiable) has a resolution on variable ν_i [Gol09]. Since the main focus of this chapter is on DMRP solving, we do not go deeper into the details of boundary point theory.

5.2 A closer look at DMRP

This section analyses the DMRP approach, especially the search routine listed in Algorithm 5.2, from a practical point of view. A crucial part addresses the fact that branching decisions are based on the set of clauses that are falsified by a current point $\mathcal{P}_t$. It requires the solver to know this set of clauses. This could be realised analogously to the implementation of local search approaches [SLM92, Fuk04] where the solver keeps track of clauses that change their state from `satisfied` to `falsified` and *vice versa*, whenever the value of a variable changes. However, for any variable ν_q, this requires the solver to know all clauses where literal λ_q (or $\overline{\lambda_q}$) occurs. Since the introduction of the two watched literals scheme [MMZ$^+$01], most CDCL solvers do not maintain complete occurrence lists of variables or literals.
In this section, we present a data structure that allows for a fast computation of the most frequently required information in the DMRP approach by simultaneously avoiding the maintenance of complete occurrence lists.

5.2.1 Different states of variables

In CDCL solvers, each variable $\nu_q \in \mathcal{V}$ can actually have three values: $val(\nu_q) \in \{true,\ false,\ unknown\}$. In general, any variable whose value is known has either been chosen as decision variable or its value is implied by BCP. To undo decisions and their implications both types of assignments (decisions and implications) are placed on a stack (often called a trail) in the order they are assigned [ES03, Bie08b].

In the DMRP algorithm, we introduce two different kinds of values expressed by the functions *pval* and *tval*. The DMRP algorithm maintains a reference point $\mathcal{P}$ which is an assignment of all the variables in the formula. Hence, for any variable ν_q in the formula, the reference point $\mathcal{P}$ either contains λ_q ($\nu_q = true$) or its negation, $\overline{\lambda_q}$ ($\nu_q = false$). For a variable ν_q, we refer to its value in $\mathcal{P}$ as $pval(\nu_q) \in \{true, false\}$. The second kind of value, $tval(\nu_q)$, is used to state a temporary modification of $pval(\nu_q)$. The default of $tval(\nu_q) \in \{true, false, ref\}$ is ref, which indicates that the corresponding variable is not affected by the current temporary modification of the reference point $\mathcal{P}_t$, and hence the value given by $pval(\nu_q)$ is valid. During the search for a modification of $\mathcal{P}_t$ to $\mathcal{P}_t'$ (Algorithm 5.2) that reduces the set of falsified clauses $\mathcal{M}$ to $\mathcal{M}' \subset \mathcal{M}$, the temporary value $tval(\nu_q) \neq ref$ hides $pval(\nu_q)$ for any variable ν_q.
We use similar shortcuts as used for partial assignments (see Section 2.1). For any variable ν_q, we say that $pval(\lambda_q)$ is true if $pval(\nu_q) = true$ and $pval(\overline{\lambda_q})$ is true if $pval(\nu_q) = false$, and analogously for $tval(\nu_q)$. Moreover, we have $tval(\lambda_q) = ref$ iff $tval(\nu_q) = ref$. Hence, we have $\lambda_q \in \mathcal{P}_t$ iff $tval(\lambda_q)$ is true, or $tval(\lambda_q) = ref$ and $pval(\lambda_q) = true$.

5.2.2 Clauses satisfied by the reference point

In addition to standard SAT solving, the algorithm has to maintain a reference point $\mathcal{P}$. Obviously, if all clauses in $\mathcal{F}$ are satisfied by the reference point $\mathcal{P}$, the algorithm has found a model for the formula. Hence, for the remainder of this section, we assume the set of clauses $\mathcal{M}$, which contains all clauses falsified by $\mathcal{P}$, to be non–empty.

After some initialisation of $\mathcal{P}$, the set $\mathcal{M}$ can be computed by simply traversing the set of all clauses $\mathcal{F}$. However, while the algorithm tries to modify $\mathcal{P}$ in order to satisfy more clauses of $\mathcal{M}$, we have to keep track of the clauses in $\mathcal{F} \setminus \mathcal{M}$ that become temporarily falsified by a temporarily modified reference point $\mathcal{P}_t$. These clauses are put onto a stack $\mathcal{D}$, which is described further below. The first matter is how to compute the clauses that are falsified by a modification of the reference point.

Similar to the concept of watched literals [MMZ+01], for each clause C in $\mathcal{F} \setminus \mathcal{M}$, we choose one literal $\lambda_q \in C, \lambda_q \in \mathcal{P}_t$ to take responsibility for C regarding its satisfiability by the current point $\mathcal{P}_t$. By definition, for any clause in $\mathcal{F} \setminus \mathcal{M}$, at least one such literal $\lambda_q \in C$ has to exist with $\lambda_q \in \mathcal{P}_t$. We refer to the set of clauses for which a literal λ_q is responsible as $R(\lambda_q)$. Let us assume that literal $\lambda_q \in \mathcal{P}_t$ is not temporarily modified. Thus, we have $tval(\nu_q) = ref$ and $pval(\nu_q) = true$. If the value of variable ν_q is flipped to modify $\mathcal{P}_t$ to $\mathcal{P}'_t$, $tval(\nu_q)$ becomes $false$ to indicate $\overline{\lambda_q} \in \mathcal{P}'_t$. This requires all clauses in $R(\lambda_q)$ to be traversed. For each clause $C \in R(\lambda_q)$, a new literal from the modified reference point $\mathcal{P}'_t$ has to be found that takes responsibility for C. Note that, in addition to the responsibilities regarding the reference point, there are also two literals per clause that watch this clause in the sense of the usual two watched literals scheme [MMZ+01]. This is necessary to notice whenever a temporary modification ($tval$) generates a unit clause or completely falsifies a clause C, such that no further decisions to modify $\mathcal{P}_t$ can satisfy C. Let the set of clauses that are watched by a literal λ_q be $W(\lambda_q)$. We examine the different cases in more detail below.

The complete procedure of assigning a temporary value to a variable is outlined in Algorithm 5.3. The following explanation references the respective parts of Algorithm 5.3. Whenever the value of a variable is changed while searching for a modified reference point $\mathcal{P}'_t$ ($tval$ is changed), we have to take care of both $W(\lambda_q)$ and $R(\lambda_q)$ of the corresponding literal λ_q that became false by $tval$ (i.e. $tval(\overline{\lambda_q})$ became true). When examining $W(\lambda_q)$ (line 4), the usual four cases as in CDCL may happen for any affected clause C that is watched by λ_q and some other literal $\lambda_w \in C$. For these cases, only the values of $tval$ play a role:

W.1 C is already satisfied by the appropriate temporary assignment of the other watched literal λ_w (line 6).

W.2 Another literal $\lambda_j \in \{C \setminus \lambda_w\}$ with $tval(\lambda_j) \neq false$ can watch C (line 7). Note that this step does not consider whether C is satisfied by the point $\mathcal{P}'_t$. Assume that C is falsified by $\mathcal{P}'_t$. If at least two literals in C are not temporarily modified, a later decision on C could then be made to satisfy C by another modification of $\mathcal{P}'_t$.

W.3 All other literals in $\{C \setminus \lambda_w\}$ are falsified with respect to $\mathcal{P}'_t$. Hence, clause C is unit regarding the temporary modifications by $tval$, and $tval(\lambda_w)$ has to be set to $true$ to satisfy C (line 11).

W.4 If, in the second case above, $tval(\lambda_w)$ is already set to $false$, a conflicting assignment is generated and the algorithm jumps back to resolve the conflict (line 10).

In addition, the DMRP algorithm has to detect the clauses that are no longer satisfied by the modified point $\mathcal{P}'_t$. Each clause C of $R(\lambda_q)$ for a literal λ_q whose value $tval(\nu_q)$ changes has to be inspected. The following update is done after the list $W(\lambda_q)$ has been examined successfully.

R.1 If $tval(\nu_q)$ equals $pval(\nu_q)$, nothing has to be done. This case may apply when $tval(\lambda_q)$ is implied by unit propagation, as in case W.3 above (line 13).

R.2 $tval(\nu_q)$ differs from $pval(\nu_q)$ and another literal in C can be found to take responsibility for C. This might be any literal $\lambda_j \in C$ with $\lambda_j \in \mathcal{P}'_t$. This applies if $tval(\lambda_j) = true$ (R.2.1), or if $tval(\lambda_j) = ref$ and $pval(\lambda_j) = true$ (R.2.2). In the latter case (from line 17), C is removed from $R(\lambda_q)$ and put into $R(\lambda_j)$, since λ_j can take responsibility for C even if the temporary modification is undone.
If, in the first case (line 15), $tval(\lambda_j) = true$, C remains in $R(\lambda_q)$ since $tval(\lambda_j)$ was obviously assigned before the current change in the value of ν_q in the reference point. C is satisfied by the modified point $\mathcal{P}'_t$ unless the temporary assignment of ν_j (and ν_q) is undone.

R.3 $tval(\nu_q)$ differs from $pval(\nu_q)$ but no other literal $\lambda_j \in C$ satisfies C under the current temporary point $\mathcal{P}'_t$. In that case, C is put on the stack $\mathcal{D}$, which keeps track of all clauses that are falsified by the current temporary reference point (line 20). Note that since $W(\lambda_q)$ is examined first, there are at least two literals $\lambda_i, \lambda_j \in C$ for which $tval(\lambda_i) = tval(\lambda_j) = ref$ and $pval(\lambda_i) = pval(\lambda_j) = false$. If this does not hold, one of the cases W.3, W.4 or R.2 will apply. At a later point in the search, Algorithm 5.2 may choose one of these literals in line 8 as the basis for a decision.

Algorithm 5.3: Update data structure

```
Require A variable $\nu_q \in \mathcal{V}$ where $tval(\nu_q)$ has been changed from $ref$
to $b \in \{true, false\}$. W.l.o.g. we assume that $tval(\overline{\lambda_q}) = true$.
Return a conflicting clause $C$ or OK if no conflicts arise
Function onChangeOfVariableTVal $(\overline{\lambda_q})$
4     forall $C \in W(\lambda_q)$ do
          $\lambda_w \leftarrow$ otherWatched$(C, \lambda_q)$
6         if $tval(\lambda_w) = true$ then continue                          | W.1
7         if $\exists\ \lambda_j \in C:\ tval(\lambda_j) \neq false \wedge \lambda_j \neq \lambda_w$ then
              $W(\lambda_j) \leftarrow W(\lambda_j) \cup \{C\}$                   | link new watched
              $W(\lambda_q) \leftarrow W(\lambda_q) \setminus \{C\}$                | W.2
10        else if $tval(\lambda_w) = false$ then return $C$                  | W.4
11        else $tval(\lambda_w) \leftarrow true$                             | W.3
13    if $tval(\nu_q) = pval(\nu_q)$ then return OK                          | R.1
      forall $C \in R(\lambda_q)$ do
15        if $\exists\ \lambda_j \in C : tval(\lambda_j) = true$ then
              continue                                                       | R.2.1
17        else if $\exists\ \lambda_j \in C:\ tval(\lambda_j) = ref\ \wedge pval(\lambda_j) = true$ then
              $R(\lambda_q) \leftarrow R(\lambda_q) \setminus \{C\}$
              $R(\lambda_j) \leftarrow R(\lambda_j) \cup \{C\}$                    | R.2.2
20        else push $C$ at $\mathcal{D}$                                       | R.3
      return OK
```

Note that this implementation allows for backtracking without any updates of the sets $R(\lambda_q)$ of any literal λ_q. The responsibility list $R(\lambda_q)$ has to be examined only when the $tval(\nu_q)$ of a variable ν_q changes its value from ref to $true$ or $false$, but not for the opposite case, which applies for backtracking. Moreover, the data structure is sound in the sense that no clause that gets falsified by $\mathcal{P}$ will be missed.
Regarding the implementation of responsibility lists, a further aspect can be considered. For a literal $\lambda_q \in \mathcal{L}$, the list $R(\lambda_q)$ is only meaningful if $pval(\lambda_q) = true$. To save memory, responsibility lists can thus be associated to variables in practice.

5.2.3 Keeping track of temporarily falsified clauses

While trying to modify a reference point $\mathcal{P}_t$ to $\mathcal{P}'_t$, with the aim of reducing the set of falsified clauses from $\mathcal{M}$ to $\mathcal{M}' \subset \mathcal{M}$, a data structure $\mathcal{D}$ is used. $\mathcal{D}$ stores those clauses that get falsified by a temporary reference point $\mathcal{P}_t$. In Section 5.2.2, we described the procedure for and conditions of adding clauses to $\mathcal{D}$. In this section, the design of the data structure $\mathcal{D}$ is presented.

Three main demands have to be met, whereas the third issue is addressed in Section 5.2.4 and is only mentioned briefly here:

- Backjumping over parts of the temporary modification (due to a conflict — see case W.4) has to be very fast with the least possible overhead to update $\mathcal{D}$.
- Clauses that are falsified by the current point $\mathcal{P}_t$ have to be found in reasonable time without having to traverse the clause's literals at each look–up in the data structure.
- When a temporarily falsified clause C is chosen from $\mathcal{D}$ by the decision procedure in line 7 of Algorithm 5.2, the algorithm aims to satisfy clause C by another modification of the reference point. Let the set $\Lambda_C = \{\lambda_i \in C \ : tval(\lambda_i) = ref\}$ contain those literals of C whose value in $\mathcal{P}_t$ may possibly be modified to satisfy C. Due to the watched literals scheme (as stated in case R.3 above), we have $|\Lambda_C| \geq 2$. The data structure has to allow for the computation of the applied heuristic regarding which literal to choose from Λ_C. Originally, this is the literal $\lambda_q \in \Lambda_C$, which is contained in most of the clauses in $\mathcal{D}$.

Fast backjumping

The overall data structure is depicted in Figure 5.1. Since the first issue above is fundamental, we use a stack to realise $\mathcal{D}$, which has one entry pL_d for each decision level d. Each entry basically points to a set of clauses L_d that are falsified by the current point $\mathcal{P}_t$ at this level. In addition, each clause set L_d has a flag that indicates if the referred clauses still have to be considered as belonging to $\mathcal{D}$.
This allows for very fast backjumping. For each level d we jump back, the corresponding flag in L_d is set to false, the set of clauses in L_d is deleted and pL_d is popped from the stack. This means a negligible overhead compared to backjumping in CDCL solving. Note that an entry pL_d that is removed from the stack does not completely destroy the referenced set L_d. Importantly, this allows other data to still refer to L_d. Invalid references may be updated lazily later on. Frequent garbage collection destroys invalid clause sets L'_d unless they are referenced by some other object. Another important advantage of this implementation will become even more evident below.

Finding clauses falsified by $\mathcal{P}_t$

To find the clauses in $\mathcal{D}$ that still have to be satisfied by further modifications of the current point $\mathcal{P}_t$, all valid clause sets L_d have to be traversed. These are exactly the sets L_d that are referenced by the entries pL_d of the stack $(0 \leq d < |\mathcal{D}|)$.

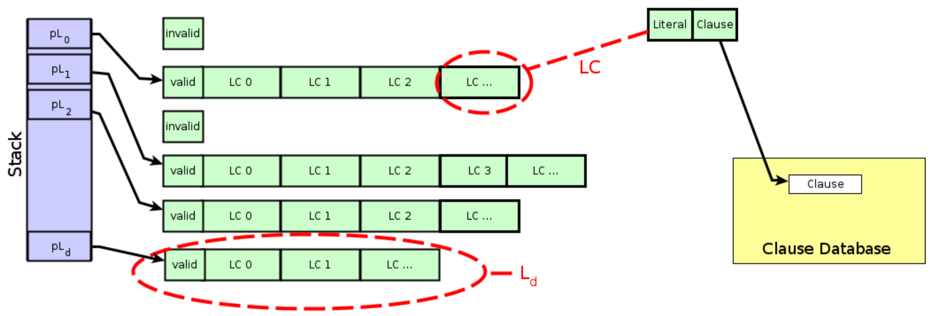


Figure 5.1: Basic data structure for storing falsified clauses

We do not remove any satisfied clause C from any set L_d that is still flagged as being valid. Otherwise, this would effectively require us to put such clauses back into L_d whenever the satisfying modification to C is undone. Instead, we also keep a literal for each clause in L_d as a kind of representative. Thus, any entry in L_d (besides the flag) is a tuple which consists of a clause C and one representative literal $\lambda_r \in C$. In the following, a tuple is wrapped into an object of type $LC =< \lambda_r, C >$.
At some point during a search, a clause C that is referred to by some set L_d may be detected as being satisfied by the temporary point $\mathcal{P}_t$ by a literal $\lambda_r \in C$, $\lambda_r \in \mathcal{P}_t$ (equivalently $tval(\lambda_r) = true$). In that case, $\lambda_r \in C$ becomes the representative literal for C. We define the concept of representative literals more formally below:

Definition 2. *For any clause C a literal $\lambda_r \in C$ is called a **representative literal** if the following holds: $\lambda_r \in \mathcal{P}_t \Rightarrow C$ is satisfied by $\mathcal{P}_t$.*

The procedure `getFalsified` is listed in Algorithm 5.4. The search for a falsified clause starts at the position where the previous call to the function found a falsified clause. If a conflict arises between two consecutive calls of the function, the stored position d is invalid and is reset to $d = \infty$ during conflict analysis. For this reason, d and c may have to be changed in line 5. To find clauses in $\mathcal{D}$ that are still falsified by the current point $\mathcal{P}_t$, the stack $\mathcal{D}$ is traversed from its top to its bottom. This prefers the most recently added clauses for decisions, as it is motivated by the success of the BerkMin solver [GN07].
The first clause that is found to be falsified by $\mathcal{P}_t$ is taken as the basis for the next branching decision. In line 8, c is decremented to access the next element $LC =< \lambda_r, C >$ at position c of the clause list L_d. If the representative literal λ_r is set in the current point $\mathcal{P}_t$ (line 10), clause C is satisfied and the algorithm continues with the next clause without actually touching C. Otherwise, the literals of clause C have to be inspected by the procedure `isSatisfied` invoked in line 11. If a literal $\lambda_q \in C$ can be found that satisfies

Algorithm 5.4: Find next clause falsified by $\mathcal{P}_t$

```
   Require Current reference point $\mathcal{P}_t$, stack $\mathcal{D}$, the level $d$ and index $c$
   into $L_d$ where the last clause was found
   Return The level $d$ in $\mathcal{D}$ and the index into $L_d$ where the next
   falsified clause is located, or $< \infty, \infty >$ if all clauses are satisfied by $\mathcal{P}_t$
   Function getFalsified ($\mathcal{P}_t, \mathcal{D}, d, c$)
       if $\mathcal{D} = \emptyset$ then return $< \infty, \infty >$
5      if $d \geq |\mathcal{D}|$ then $d \leftarrow |\mathcal{D}| - 1$; $c \leftarrow |L_d|$          | $d$ was reset
       while $true$ do
           while $c > 0$ do
8              $c \leftarrow c - 1$                                        | decrement first
               $< \lambda_r, C > \leftarrow L_d[c]$                      | element at index $c$ in $L_d$
10             if $\lambda_r \in \mathcal{P}_t$ then continue
11             $\lambda_q \leftarrow$ isSatisfied ($\mathcal{P}_t, C$)
               if $\lambda_q \neq \emptyset$ then
13                 $L_d[c] \leftarrow < \lambda_q, C >$                   | set representative
14             else return $< d, c >$                             | falsified clause found
15         if $d = 0$ then return $< \infty, \infty >$                     | all satisfied
           $d \leftarrow d - 1$; $c \leftarrow |L_d|$                        | continue with next level

18 Function isSatisfied ($\mathcal{P}_t, C$)
       foreach $\lambda_q \in C$ do
           if $\lambda_q \in \mathcal{P}_t$ then return $\lambda_q$
       return $\emptyset$
```

C with respect to $\mathcal{P}_t$, the representative literal can be set appropriately (in line 13) and the next clause is checked. If C is not satisfied by the reference point, a falsified clause is found for the next decision and the algorithm returns a reference in line 14. If all clauses of one level have been handled, the next level is chosen. If no level is left, the algorithm returns $< \infty, \infty >$ in line 15 to indicate that all clauses in $\mathcal{D}$ are satisfied by the reference point $\mathcal{P}_t$.

A crucial issue within the search for a falsified clause in $\mathcal{D}$ is the use of representative literals. The state of the representative literal is always tested, before the entire clause is inspected. On the one hand, this ensures that a satisfied clause is not inspected twice unless a conflict causes a modification of the point that makes this necessary (when the assignment to the particular representative literal is undone). On the other hand, significant changes of $\mathcal{P}_t$ to the satisfiability state of any clause are not missed. The latter issue would require extra maintenance if only Boolean flags were used to mark satisfied clauses instead of representative literals.

5.2.4 Computation of the MakeCount of variables

Given a clause C^* that is falsified under the current point $\mathcal{P}_t$, the heuristic has to compute the literal of $\Lambda_{C^*} = \{\lambda_q \in C^* \ : tval(\lambda_q) = ref\}$, which satisfies most of the clauses in $\mathcal{D}$ (or, optionally, $\mathcal{D} \cup \mathcal{M}$) when the value of ν_q is flipped in $\mathcal{P}_t$. To compute the so–called *MakeCount* of a variable, we extend the data structure of Figure 5.1.

The extended data structure allows for a lazy computation of the Make-Count of a variable. It is depicted in Figure 5.2 and is organised as follows: Each variable ν_i that is not yet affected by the temporary modification of the reference point $\mathcal{P}_t$ $(tval(\nu_i) = ref)$ is associated with a list Ω_i of elements of type M. An element M represents a clause in $\mathcal{D}$ that can be satisfied by flipping the current value of ν_i in the point $\mathcal{P}_t$. We refer to the k-th element of list Ω_i as $M_i[k]$. Due to the laziness of the data structure, it might be that an element is out–of–date. More precisely, each element $M_i[k]$ (representing some clause C') in the list Ω_i of variable ν_i consists of two fields: The first field references the set L_d of clauses in $\mathcal{D}$, which C' is contained in. Note that L_d can still be referenced from the list Ω_i even if the reference to L_d is removed from $\mathcal{D}$ within backjumping. The second field is an index into L_d that indicates the particular clause C' (i.e. the respective element LC) that can be satisfied by flipping the value of ν_i in $\mathcal{P}_t$.

Whenever case $R.3$ from above applies for a clause C, stack $\mathcal{D}$ is extended by C. The procedure is outlined in Algorithm 5.5. A pointer to C and some representing literal $\lambda_r \in C$ are wrapped into an object LC in line 4 of Algorithm 5.5. This data is appended to the set of clauses L_d, which is referenced from the topmost entry of the stack $\mathcal{D}$ (line 6). At this point, we also add an entry M to the MakeCount lists Ω_i of variables ν_i which can be chosen to satisfy C later on. These are the variables for which $\lambda_i \in \Lambda_C$. This is done by the loop in line 8.

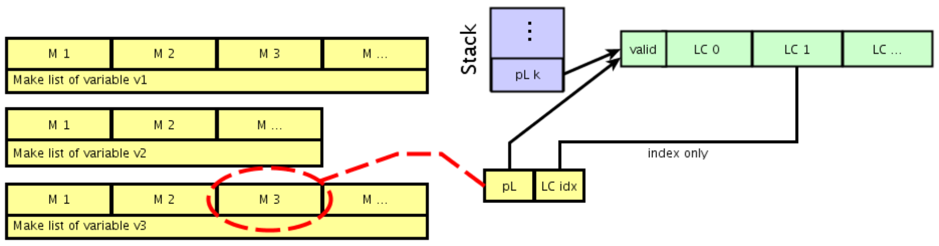


Figure 5.2: Computation of the *MakeCount* of variables. Extension of Figure 5.1.

Algorithm 5.5: Push falsified clause on the stack

```
Require Clause $C$ that has to be pushed on $\mathcal{D}$.
Function pushStack ($C$)
      $d \leftarrow |\mathcal{D}| - 1$                      | recent decision level
4     $LC \leftarrow < C[0], C >$            | wrap clause with representative
      $c \leftarrow |L_d|$                      | next position for $LC$ in $L_d$
6     $L_d[c] \leftarrow LC$                  | add clause to $\mathcal{D}$ at level $d$
      $M \leftarrow < L_d, c >$          | wrap reference for MakeCount lists
8     foreach $\lambda_i \in C$ do
          if $tval(\lambda_i) = ref$ then $\Omega_i \leftarrow \Omega_i \cup M$
```

When computing the valid MakeCount from the possibly out–of–date information, different cases may appear:

M.1 It might be that an element $M_i[k]$ refers to a clause that has been removed from $\mathcal{D}$. In that case, the flag of the structure L_d that is referenced by $M_i[k]$ has been invalidated during backtracking. Hence, this case can be realised immediately and $M_i[k]$ may be deleted from Ω_i.

In the following cases, it is ensured that the clause referenced by an element $M_i[k]$ is still contained in $\mathcal{D}$. We refer to the referenced clause as $C_{d,c}$ and the respective structure $LC_{d,c}$ and assume that the referenced clause is added at $L_d[c]$ by Algorithm 5.5 in line 6.

M.2 We first assume $C_{d,c}$ to be satisfied by a modification of the point $\mathcal{P}_t$. Recall, that what we actually get from $M_i[k]$ is a reference to $LC_{d,c}$, which wraps the clause $C_{d,c}$ and a representing literal λ_r of $C_{d,c}$. We can distinguish between two cases:

M.2.1 $C_{d,c}$ might have been considered by the procedure `getFalsified` in Algorithm 5.4 to find falsified clauses in $\mathcal{D}$. In this case, the representative literal λ_r has been set so that by checking the state of λ_r in $\mathcal{P}_t$, we know that $C_{d,c}$ is satisfied and we have finished.

M.2.2 If $C_{d,c}$ has not been considered by Algorithm 5.4 yet, the satisfiability state of the clause has to be computed by inspecting its literals. Given that $C_{d,c}$ is satisfiable under $\mathcal{P}_t$, a literal $\lambda_r \in C_{d,c}$ that satisfies $C_{d,c}$ will be found and will be made the representative literal for this clause in $LC_{d,c}$. This allows for a fast detection of the satisfiability of $C_{d,c}$ later on and will relieve Algorithm 5.4 from checking all literals of $C_{d,c}$. The representative literal ensures that for each temporary point $\mathcal{P}_t$, there is, at most, one traversal through all literals of a clause to recognise that this clause is satisfied by $\mathcal{P}_t$.

M.3 If C is falsified by the current point $\mathcal{P}_t$, this is recognised by a check of all literals in $C_{d,c}$.

Algorithm 5.6: Compute the MakeCount of a variable

```
   Require Current point $\mathcal{P}_t$, variable $\nu_i$, minimum MakeCount $m$
   Return MakeCount of $\nu_i$ if it is at least $m$
   Function makeCountOf ($\mathcal{P}_t, \nu_i, m$)
       $u \leftarrow |\Omega_i|$                      | number of unchecked elements in $\Omega_i$
       $n \leftarrow 0$                               | falsified clauses in $\Omega_i$
6      foreach $< L_d, c > \ \in \Omega_i$ do
7          if $n + u < m$ then return 0               | $m$ cannot be reached
           $u \leftarrow u - 1$                       | one remaining element less
           if $L_d$ is invalid then
10             $\Omega_i \leftarrow \Omega_i \setminus < L_d, c >$   | remove invalid element
               continue
           $< \lambda_r, C > \leftarrow L_d[c]$       | get referenced element $LC$
13         if $\lambda_r \in \mathcal{P}_t$ then continue
14         $\lambda_q \leftarrow$ isSatisfied ($\mathcal{P}_t, C$)   | Algorithm 5.4 line 18
           if $\lambda_q \neq \emptyset$ then
16             $L_d[c] \leftarrow < \lambda_q, C >$   | set representative
17         else $n \leftarrow n + 1$;                 | one more falsified
       return $n$
```

Algorithm 5.6 sketches the cases listed above. To compute the Make-Count of a given variable ν_i, each element of the MakeCount list Ω_i has to be inspected (line 6). For practical applications, the computed value is only of interest if it exceeds a minimum value m. In line 7, this issue is used to terminate early if the required minimum MakeCount m can no longer be reached.

If case M.1 applies and the referenced set of clauses L_d is no longer valid, this clause is removed from the MakeCount list Ω_i in line 10. The representative literal of a clause C may already indicate that C is satisfied by $\mathcal{P}_t$, as described in case M.2.1 and shown in line 13. The inspection of the literals of C in line 14 may detect that the clause is satisfied by $\mathcal{P}_t$ and case M.2.2 applies. The representative literal can then be set appropriately in line 16. Otherwise, one more falsified clause is counted in line 17.

The realisation of the set $\mathcal{D}$ and the data structure to compute Make-Counts of variables follows the idea of maintaining data structures lazily. This avoids the storage of complete occurrence lists for literals. MakeCount lists do not require any update operations on backjumping. Even though indices in M become undefined when the referenced set L_d is cleared during backjumping, this is not problematic, since an index is only used after L_d is asserted as still being valid by its flag. The fast propagation and backjumping is clearly realised at the expense of the costly detection of falsified clauses in case M.3. This fact can be observed in Algorithm 5.6 in line 14 when calling the function `isSatisfied`, where, at worst, all literals have to be inspected.

Algorithm 5.7: Compute variables with the best MakeCount

```
   Require Current point $\mathcal{P}_t$, clause $C$
   Return Set of variables in $C$ with the best MakeCount
   Function bestVars ($\mathcal{P}_t, C$)
       $m \leftarrow 0$                                      | best MakeCount
       $R \leftarrow \emptyset$                              | variables with best MakeCount
 6     foreach $\lambda_i \in C$ do
 7         if $tval(\lambda_i) \neq ref$ then continue
           $x \leftarrow$ makeCountOf ($\mathcal{P}_t, \nu_i, m$)
 9         if $x > m$ then $R \leftarrow \{\nu_i\}$; $m \leftarrow x$      | better than all
10         else if $x = m$ then $R \leftarrow R \cup \{\nu_i\}$           | equal candidate
       return $R$
```

As in Algorithm 5.6, the size of a list Ω_i of a variable ν_i gives an upper bound $\widehat{\Omega_i}$ on the valid MakeCount of ν_i. The computation of a MakeCount can terminate early if a minimum value cannot be reached anymore (line 7 of Algorithm 5.6). This issue can be used when the variable ν_i with the highest MakeCount has to be found within a set of candidates. Algorithm 5.7 describes the procedure of finding a set of variables from a given clause C that have the highest MakeCounts. The best flip candidate, as required in line 8 of Algorithm 5.2, can be chosen among the set of candidates returned by Algorithm 5.7.
For a given clause C, Algorithm 5.7 inspects all literals of the set Λ_C (lines 6 and 7). Depending on the computed MakeCount, a variable may replace all collected variables in line 9 or extend the set of variables that have an equal MakeCount in line 10. The requirement for the minimum MakeCount m can be utilised even better when the variables ν_i are inspected in decreasing order (in line 6) with respect to the size of the MakeCount lists Ω_i.

Better detection of falsified clauses

The computation of the MakeCount of a variable and the procedure of finding falsified clauses in $\mathcal{D}$, as presented above, have to face the costs associated with fast unit propagation and backjumping. This is desirable for the following reason:

For industrial SAT benchmarks, one decision may imply many other assignments by unit propagation. For some benchmarks, the average number of implications by unit propagation per decision ranges up to thousands. Moreover, undoing decisions after conflict analysis requires all implied assignments to be undone. Higher costs for decision making may be compensated by fast propagation and backjumping.

Nevertheless, in order to use more features in a DMRP search, such as the elimination of boundary points, more information is required during the search. In this regard, we first analyse the main drawbacks of Algorithm 5.4 and Algorithm 5.6, and present a modification of the data structure depicted in Figure 5.2.

Within both procedures for finding the next falsified clause in $\mathcal{D}$ (Algorithm 5.4) and computing the valid MakeCount of a variable (Algorithm 5.6), the representative literal λ_r of the clause C may indicate that the considered clause is satisfied by the current reference point $\mathcal{P}_t$. However, the converse argument of Definition 2 is not valid. For this reason, all literals of C have to be inspected by the subroutine `isSatisfied` if $\lambda_r \notin \mathcal{P}_t$.

Making representative literals more meaningful

In order to avoid the invocation of the subroutine `isSatisfied` to ensure that a clause is falsified by the current reference point, we define a stronger concept of representative literals:

Definition 3. *For any clause C, a representative literal $\lambda_r \in C$ is called* ***liable*** *if the following holds: $\lambda_r \in \mathcal{P}_t \Leftrightarrow C$ is satisfied by $\mathcal{P}_t$.*

Obviously, if it can be ensured that all representative literals are liable, any call to the function `isSatisfied` in Algorithm 5.4 and Algorithm 5.6 is superfluous. With this, accessing the referenced clause in both algorithms is superfluous as well. The literals of a referenced clause are only accessed when the clause is known to be falsified by $\mathcal{P}_t$ and is chosen for a decision during the search in line 7 of Algorithm 5.2.

We examine the process of DMRP for one clause by the following short example. Consider the initial reference point $\mathcal{P} = \{\overline{\lambda_1}, \overline{\lambda_2}, \overline{\lambda_3}, \overline{\lambda_4}, \ldots\}$ and clause $C = (\lambda_1 \vee \overline{\lambda_2} \vee \lambda_3 \vee \lambda_4)$. We assume that at decision level d_i, the point is changed to $\mathcal{P}_i = \{\overline{\lambda_1}, \lambda_2, \overline{\lambda_3}, \overline{\lambda_4}, \ldots\}$. Algorithm 5.3 will detect that C is falsified by the temporary point $\mathcal{P}_i$ and puts a reference to C into $\mathcal{D}$ (with representative literal λ_1), as described in Algorithm 5.5. In particular, the MakeCount lists Ω_1, Ω_2 and Ω_4 of variables ν_1, ν_3 and ν_4 are extended by one entry (we assume that $tval(\nu) = ref$ for these variables). Assume that at a later decision $d_k > d_i$, the point is temporarily modified to

$\mathcal{P}_k = \{\overline{\lambda_1}, \lambda_2, \overline{\lambda_3}, \lambda_4, \ldots\}$. Now C is satisfied by $\mathcal{P}_k$, but this is not realised immediately. Another decision at level $d_m > d_k$ may modify the point to $\mathcal{P}_m = \{\overline{\lambda_1}, \lambda_2, \lambda_3, \lambda_4, \ldots\}$. If, at a level $d_n > d_m$, the clause C is inspected by the subroutine `isSatisfied`(C) (see Algorithm 5.4, line 18), the representative literal will be set to λ_3. Unless the search jumps back below level d_m, the representative literal λ_3 will always indicate that clause C is satisfied by the point. However, if the search jumps back to a level d_l with $d_k < d_l < d_m$, clause C is still satisfied by literal λ_4 but the representative literal $\lambda_3 \notin \mathcal{P}_l$ cannot indicate this. Clearly, choosing λ_4 as representative for C would have been a better choice.

An obvious solution for overcoming a bad choice for the representative literal could modify the function `isSatisfied` in line 18 of Algorithm 5.4 to always inspect all literals and return the satisfying literal with the lowest decision level as the representative, if one exists. Therefore, consider that literal λ_4 (instead of λ_3) is chosen as representative at level d_n and the search will continue. Unless the search jumps back below level d_k, the representative literal will indicate that C is satisfied by the point. Assume that conflict analysis causes to jump back to level d_j with $d_i < d_j < d_k$. Clause C is falsified again by the point $\mathcal{P}_j$. Assume a decision at a higher level $d_q > d_j$ will then modify the point to $\mathcal{P}_q = \{\lambda_1, \lambda_2, \overline{\lambda_3}, \overline{\lambda_4}, \ldots\}$. With this, clause C is satisfied by $\mathcal{P}_q$ but the representative literal λ_4 does not indicate this. Hence, a better choice for the representative literal may avoid some invocations of the function `isSatisfied` but is still not liable. These observations motivate the following property.

Property 6. *Assume that for any satisfied clause C that was put into $\mathcal{D}$ at level d_j, the representative literal $\lambda_l \in C$ is certain to be at the lowest possible decision level d_l, such that C is satisfied by $\mathcal{P}_l$ but not by any point $\mathcal{P}_k$ at level $d_j \leq d_k < d_l$. By this assurance, the representative literal is liable.*

As long as the assignments of level d_l are not undone, literal λ_l is in the modified reference point and clearly indicates that clause C is satisfied. If, on the other hand, the search jumps back to a level below d_j, the reference to C is invalidated during backjumping. If the search jumps back to a level d_k, with $d_j \leq d_k < d_l$, clause C is falsified by $\mathcal{P}_k$ and literal $\lambda_l \notin \mathcal{P}_k$.
To ensure Property 6, the representative literals can be set whenever a new assignment is made. Algorithm 5.8 lists the procedure that has to be called after unit propagation for all freshly assigned literals.

Algorithm 5.8 uses the fact that the MakeCount list Ω_i of a variable ν_i contains all clauses that can be satisfied by an assignment $tval(\nu_i) \leftarrow \overline{pval(\nu_i)}$. For a newly assigned variable ν_i, all elements in Ω_i are inspected in line 3. As in Algorithm 5.6, the iteration over Ω_i can optionally be used

Algorithm 5.8: Mark satisfied clauses

```
  Require Current point $\mathcal{P}_t$, recently assigned literal $\lambda_i \in \mathcal{P}_t$
  Function markSatisfied ($\mathcal{P}_t, \lambda_i$)
3     foreach $< L_d, c > \ \in \Omega_i$ do
          if $L_d$ is invalid then
5             $\Omega_i \leftarrow \Omega_i \backslash < L_d, c >$;          | remove invalid element
          else
              $< \lambda_r, C > \leftarrow L_d[c]$          | get referenced element $LC$
8             if $\lambda_r \notin \mathcal{P}_t$ then
                  $L_d[c] \leftarrow < \lambda_i, C >$          | set representative
```

to remove invalid entries in line 5. In line 8, it is crucial to change the representative literal only if the current one is not set in the point $\mathcal{P}_t$. Invoking Algorithm 5.8 for all freshly assigned variables ensures that the representative literal of any clause has the lowest possible decision level. With Property 6, the representative literals are liable and any call to the function `isSatisfied` in Algorithm 5.4 and Algorithm 5.6 is superfluous.

As already mentioned above, this allows a faster execution of Algorithm 5.4 to obtain falsified clauses in $\mathcal{D}$, and of Algorithm 5.7 to compute the best MakeCount. The backjumping procedure during a search does not have to be adapted. However, this is realised at the expense of unit propagation. In Section 5.3.2, this computation of the MakeCount is used to allow for the elimination of boundary points within a DMRP search.

5.2.5 Learning

A DMRP search aims to modify the reference point to one that falsifies fewer clauses of the formula. Based on some falsified clauses, decisions are made about changing a variable's value in the current point. If a trail of decisions leads to a conflict, where some previous assignments in the trail are contradictory, the conflict is analysed as in CDCL. This is mentioned in case W.4 and listed in line 11 of Algorithm 5.2. Related to conflict analysis and learning, two aspects have to be considered for the DMRP approach. Both issues are particularly important for the invariants within the implementation of the DMRP approach.

L.1 Whenever a unit clause $C = (\lambda_u)$ is learnt, the search jumps back to decision level 0 within a DMRP search (line 13 of Algorithm 5.2), assigns λ_u in the temporary point $\mathcal{P}_t$ and propagates all implications of this assignment. These forced assignments may imply that the temporary $\mathcal{P}_t$ differs from the initial valid point $\mathcal{P}$.

If this applies in line 14 of Algorithm 5.2, $\mathcal{P}_t$ is returned immediately to modify the valid point permanently in the invoking function. This modification of $\mathcal{P}$ to $\mathcal{P}'$ (in line 13 of Algorithm 5.1) exhibits a major difference from previously described modifications: The set of clauses $\mathcal{M}'$ falsified by $\mathcal{P}'$ does not necessarily have to be a subset of $\mathcal{M}$ — the set of clauses falsified by the previous reference point $\mathcal{P}$.

L.2 For any learnt lemma C^*, with $|C^*| > 1$ that is generated when a conflict is analysed, the data structure has to be updated properly. In particular, C^* has to be linked to obey the invariants of $\mathcal{M}$ and $\mathcal{D}$. We prove the following property.

Property 7. *Let Algorithm 5.2 be invoked with clause C_0 falsified by the currently valid reference point $\mathcal{P}$. Assume that clause C_0 is the only clause in the set $\mathcal{F} \setminus \mathcal{M} \cup \{C_0\}$ that is falsified by $\mathcal{P}$. With this, any lemma generated by the function* `analyseConflict` *(in line 11 of Algorithm 5.2) contains at least one literal λ_q with $\lambda_q \in \mathcal{P}$ ($pval(\lambda_q) = true$). With this, the generated lemma cannot belong to the current set $\mathcal{M}$.*

Proof. We prove Property 7 by the construction of learnt lemmas within conflict analysis, as described in Section 2.2.4. The function `dmrpTryModifyPoint` presented in Algorithm 5.2 considers the clause set $\mathcal{F} \setminus \mathcal{M} \cup \{C_0\}$. A modification of the valid reference point $\mathcal{P}$ is sought that satisfies all clauses in this set, particularly C_0, which is the only clause in the given set that is falsified by $\mathcal{P}$. By definition, any clause C_i in the set $\mathcal{F} \setminus \mathcal{M}$ is satisfied by $\mathcal{P}$ and contains at least one literal $\lambda_q \in C_i$ that is assigned in the point $\lambda_q \in \mathcal{P}$ (equivalently $pval(\lambda_q) = true$).
Clause C_0 is the base for the topmost decision in line 7 of Algorithm 5.2 since, by assumption, it is the only falsified clause in $\mathcal{D}$. Hence, C_0 is satisfied by the first modification of the point $\mathcal{P}_t$ and cannot be asserting for a later assignment of a variable. When the search runs into a conflict, there is a conflicting clause $C_c \neq C_0$ that has all literals assigned as $false$ ($tval(\overline{\lambda_q}) = true \; \forall \; \lambda_q \in C_c$).

As described in Section 2.2.4, the lemma C^* is generated by recursively resolving variables (that are not decisions) from the conflicting clause C_c by using the asserting clauses. C_c is the first version ($C_0^* = C_c$) of the generated lemma C^*. Given that $C_c \in \mathcal{F} \setminus \mathcal{M}$, there is at least one literal $\lambda_q \in C_0^*$ that is assigned in the valid point $\mathcal{P}$ ($pval(\lambda_q) = true$, $\lambda_q \in \mathcal{P}$). Let λ_q be one literal $\in C_i^*$ with $\lambda_q \in \mathcal{P}$. If any literal $\lambda_p \in C_i^*, \lambda_p \neq \lambda_q$ is resolved from C_i^*, the resolvent C_{i+1}^* still contains literal $\lambda_q \in \mathcal{P}$. If, on the other hand, λ_q is resolved by the use of the asserting clause $C_q \neq C_0$ for the assignment $\overline{\lambda_q}$, clause C_q obviously contains literal $\overline{\lambda_q} \notin \mathcal{P}$. Since C_q is contained in the set $\mathcal{F} \setminus \mathcal{M}$, there has to be a literal $\lambda_s \in C_q$ that is in the valid point $\mathcal{P}$. With $\lambda_s \in \mathcal{P}$, we have $\lambda_s \neq \overline{\lambda_q}$ and the new resolvent C_{i+1}^* contains literal $\lambda_s \in \mathcal{P}$.

By induction, the final lemma C^* contains at least one literal that is in $\mathcal{P}$. Hence C^* is satisfied by $\mathcal{P}$ and does not belong to the set $\mathcal{M}$. $\square$

The issue presented in L.1 mentions the fact that a unit clause that may be learnt within Algorithm 5.2 can permanently modify the reference point $\mathcal{P}$. This does not contradict the proven Property 7. Consider the following example: Let the reference point be $\mathcal{P} = \{\overline{\lambda_1}, \overline{\lambda_2}, \overline{\lambda_3}, \overline{\lambda_4}, \overline{\lambda_5}, \overline{\lambda_6} \ldots\}$. The formula contains the clauses $C_0 = (\lambda_1 \vee \lambda_2 \vee \lambda_3)$, $C_1 = (\overline{\lambda_1} \vee \lambda_4 \vee \lambda_6)$ and $C_2 = (\overline{\lambda_1} \vee \lambda_5 \vee \lambda_6)$. We also have $C_0 \in \mathcal{M}$ and $C_1, C_2 \notin \mathcal{M}$. Algorithm 5.2 is called with clause C_0.
Assume that eventually, the unit clause $(\overline{\lambda_3})$ is learnt. This is possible with Property 7 since $(\overline{\lambda_3})$ contains one literal of $\mathcal{P}$. The reference point $\mathcal{P}$ is not modified and thus the search continues. Assume that the unit clause $(\overline{\lambda_2})$ is learnt. Again, this is consistent with Property 7. However, C_0 becomes a unit itself and implies the assignment of λ_1 at level 0. The valid reference point $\mathcal{P}$ has to be modified, and therefore Algorithm 5.2 returns, as described in L.1. By the modification of the reference point to $\mathcal{P} = \{\lambda_1, \overline{\lambda_2}, \overline{\lambda_3}, \overline{\lambda_4}, \overline{\lambda_5}, \overline{\lambda_6} \ldots\}$, both clauses C_1 and C_2 are falsified by $\mathcal{P}$ and hence belong to $\mathcal{M}$. In return, C_0 will remain satisfied by any later point, unless $\mathcal{F}$ is proven to be unsatisfiable.

In the second issue, L.2, linking learnt non–unit clauses into the data structure during one execution of the function `dmrpTryModifyPoint` is realised based on the following property.

Property 8. *Let Algorithm 5.2 be invoked with clause C_0 being falsified by the currently valid reference point $\mathcal{P}$. Any generated lemma contains at least one literal in $\mathcal{P}$.*

Property 8 follows immediately from Property 7 since the precondition, namely that C_0 is the only clause falsified by $\mathcal{P}$, is ensured at invocation. Moreover, whenever the search continues after a conflict arises and a lemma is added to the formula $\mathcal{F}$, the precondition of Property 7 is ensured again. Any generated lemma C^* with $|C^*| > 1$ can be linked into $\mathcal{D}$ by choosing the literal $\lambda_s \in C^*, \lambda_s \in \mathcal{P}$ which was assigned at the highest decision level d (i.e. most recently). By Property 8, at least one such literal λ_s has to exist. Literal λ_s takes responsibility for C^*: $R(\lambda_s) \leftarrow R(\lambda_s) \cup \{C^*\}$.
The functions `backtrackResetPoint` and `learnAndPropagate` in lines 13 and 14 of Algorithm 5.2 determine a new temporary point $\mathcal{P}_t$. If $\lambda_s \notin \mathcal{P}_t$, the lemma C^* is also appended to the list L_d that is referenced by the stack $\mathcal{D}$ for decision level d and is considered for the MakeCounts as described in the previous section. These two actions ensure a proper update of the entire data structure and no more special treatments are needed.

5.3 Combining DMRP and CDCL in a Hybrid Solver

In Algorithm 5.1, we assume that the initial reference point is given from outside. In the original paper [Gol08a], reference points are chosen at random and the process attempts to modify them by a call to the function `solveDMRP`. If no result can be computed within a certain amount of time (i.e. a number of conflicts) `solveDMRP` will be invoked with a new initial point. This is similar to local search restarts but with the difference that the DMRP algorithm itself carefully considers how to modify a reference point. However, the choice of the initial point is crucial for the algorithm as presented in Section 5.4.
As already mentioned in Section 2.2.5, CDCL solvers perform restarts quite frequently. At a restart, the activity values of variables or literals are kept and a subset of the learnt clauses is carried along for the next start. However, the current partial assignment (all literals in the trail) is almost completely rejected, even though phase saving [PD07a] keeps some information. This motivates a hybrid approach that alternates between the CDCL and the DMRP algorithms. The DMRP approach offers the convenient possibility of taking a closer look at the drawbacks of a partial assignment before it is rejected. It may focus on the clauses falsified under the assignment that is given by the cached values of the phase saving heuristic.

The implementation that is shown in Algorithm 5.9 combines both approaches by the use of the Luby *et al.* restart strategy [LSZ93], which has proven itself successful in both theory and practice. The Luby strategy assumes that an algorithm does not have any external information and thus does not know when it is best to perform a restart. In that case, the available computation time is shared almost equally among different restart strategies [LSZ93]. The function `maxConflictCount` in Algorithm 5.9 returns the number of conflicts for the next run due to the Luby strategy. This number is the product of a constant factor f and the next number of the sequence $(1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, 1, \ldots)$ (see [LSZ93] for details).
The function `chooseAlgo` decides on which algorithm to use for the next run. On average, we achieved the best results when running the DMRP algorithm exactly for the smallest conflict limit (when $cl = f$).

Since the DMRP algorithm requires a reference point, i.e. an assignment to all variables, the last partial assignment of the CDCL solver has to be extended to a complete assignment (`extendPartialAssignmToRefPoint`). This is done by continuing the previous CDCL search with the last partial assignment. However, within this execution, only binary clauses are considered during the search until all variables are assigned a value. This assignment constitutes the initial reference point for the DMRP algorithm. In this phase, the solver may also realise that the formula is unsatisfiable.

Algorithm 5.9: The hybrid approach

```
Require Formula $\mathcal{F}$ in CNF
Return Sat, UnSat or Unknown
Function solveHybrid ($\mathcal{F}$)
    $last \leftarrow$ CDCL,    $res \leftarrow$ Unknown
    while $res =$ Unknown do
        $cl \leftarrow$ maxConflictCount()                | use Luby strategy
        $algo \leftarrow$ chooseAlgo($cl$)               | apply CDCL or DMRP ?
        if $algo =$ DMRP then
            if $last =$ CDCL then
                $< res, \mathcal{P} > \leftarrow$ extendPartialAssignmToRefPoint()
                if $res =$ UnSat then return $res$
            $res \leftarrow$ solveDMRP($\mathcal{F}, \mathcal{P}, cl, cl$)
        else  $res \leftarrow$ solveCDCL($\mathcal{F}, cl$)
        $last \leftarrow algo$
    return $res$
```

For the case where the partial assignment is empty (e.g. at the start of the algorithm), this function simply computes a reference point that satisfies all binary clauses. Taking care of binary clauses at first is also motivated by the work of Zheng and Stuckey [ZS02] and Bacchus [Bac02b], where the idea of primarily focusing on binary clauses has improved solving for some families of instances. This also guarantees an additional invariant for our data structure, specifically that a binary clause can neither be contained in the set $\mathcal{M}$ nor in the stack $\mathcal{D}$.

5.3.1 Adjustments for the hybrid approach

In addition to standard CDCL solving, each clause of the formula is assigned an activity value that is set to zero at the beginning. Whenever a clause is involved in a conflict (i.e. if it is used for resolution during the generation of a lemma) its activity value is increased. In some solvers (e.g. [ES03]), this technique is commonly used to clear the clause database of inactive learnt clauses periodically. Our hybrid solver maintains an activity value for every clause.

The activity value of a clause is taken into account when the next clause from the set $\mathcal{M}$ has to be chosen (line 7 of Algorithm 5.1) to be handled by the function `dmrpTryModifyPoint`. Clauses with high activity values are preferred for the next attempt at modifying the current reference point. However, if the call to `dmrpTryModifyPoint` times out for a chosen clause C, other clauses with high activity are preferred for the next invocation of Algorithm 5.2.

In contrast to the original DMRP algorithm, the conflict limit (timeout) for the function `solveDMRP` depends on the success of its subroutine `dmrpTryModifyPoint` in line 8 of Algorithm 5.1. If the current reference point $\mathcal{P}$ can be improved, the initial conflict limit is reset. The solver also differs in the computation of the MakeCount of a variable. The MakeCount of a variable may consider only the clauses currently in $\mathcal{D}$ to get the greatest level of local improvement. Optionally, all clauses in $\mathcal{D} \cup \mathcal{M}$ may be considered to make decisions more globally. For variables that have the same MakeCount, ties can be broken in favour of different issues, which is explained in more detail in Section 5.4.

5.3.2 Using boundary points

The concept of boundary points was also introduced by Goldberg [Gol09] and is briefly described in Section 5.1. Goldberg and Manolios propose a complete template algorithm that uses boundary points for SAT solving [GM10]. We do not aim to apply the complete procedure but instead use the concept of boundary points within DMRP.

Consider the DMRP algorithm using the MakeCount of variables for decision making. Thus in line 8 of Algorithm 5.2, the maximal MakeCount of all literals in the chosen clause C is computed. If there is a literal $\lambda_q \in C$ that is contained in all clauses that are falsified by the temporary point $\mathcal{P}_t$, the condition of a λ_q-boundary is nearly fulfilled.
Clearly, a flip of the value of variable ν_q produces a temporary point $\mathcal{P}'_t$ that satisfies all previously falsified clauses. Moreover, it may not falsify new clauses. In that case, the function `dmrpTryModifyPoint` returns the found point $\mathcal{P}'_t$ and we have finished. If, on the other hand, some new clauses are falsified by $\mathcal{P}'_t$, both points $\mathcal{P}_t$ and $\mathcal{P}'_t$ constitute twin boundary points. The sets of falsified clauses $\mathcal{M}$ and $\mathcal{M}'$ both contain at least one clause. By choosing one clause from each of the sets $\mathcal{M}$ and $\mathcal{M}'$, the boundary point can be eliminated by resolution on variable ν_q. A heuristic could choose the most active clause from each set. The elimination of boundary points is a reasonable option according to the propositions in [Gol09], some of which are mentioned in Section 5.1.

The most crucial issue is to detect whether a literal λ_q is contained in all clauses falsified by the point $\mathcal{P}'_t$, as required above. This can be realised by simple counting if the alternative approach of marking falsified clauses is used, as presented in Section 5.2.4. A DMRP search can keep an additional counter σ for the number of actually falsified clauses in $\mathcal{D}$. Whenever a clause is put into $\mathcal{D}$ by Algorithm 5.5, the counter σ is increased. On the other hand, if the function `markSatisfied` detects the satisfaction of a previously falsified clause (Algorithm 5.8, line 8 *et seq.*), the counter σ is decreased. To allow for inexpensive backjumps to previous states in the search, the counter has to be backed up for each decision level d before a new decision is made ($\sigma_d \leftarrow \sigma$). A backjump to level d can thus simply reset the counter $\sigma \leftarrow \sigma_d$. If the valid MakeCount for a variable ν_q, computed by Algorithm 5.6, is equal to σ, all falsified clauses must contain variable ν_q. As described above, the subsequent flip of variable ν_q may yield a λ_q-boundary point that can then be eliminated.

5.4 Evaluation

In this section, we study the performance of the DMRP approach using industrial SAT benchmarks. The first part considers the pure application of DMRP. The subsequent part analyses the hybridisation of DMRP and CDCL, as described in Section 5.3.

5.4.1 Reference points and boundary points

We illustrate the progress of an entire DMRP run for one SAT instance in Figure 5.3. In order to fit the entire progress into one plot, we chose the rather easy instance *cmu-bmc-barrel6.cnf* with 2306 variables and 8931 clauses from the application of bounded model checking [BCCZ99]. The illustrated configuration of the DMRP algorithm requires $13,335$ decisions to prove the formula to be unsatisfiable. The plot depicts several properties and is organised as follows:
The x–axis reflects the progress of solving. A vertical line through a point $(x_i, 0)$ illustrates the state of the solver after x_i decisions. The red line states the number of original clauses (not learnt clauses) that are falsified by the current reference point $\mathcal{P}$. The light grey line states the number of original clauses falsified by the temporary point $\mathcal{P}_t$. The values of both lines are related to the y–axis on the left–hand side. To state the depth of the search at the same time, a second y–scale is used, which is printed on the right–hand side. The current decision level is depicted by the purple line in the upper part of the plot. Finally, a blue point is printed whenever a boundary point was found and eliminated.

The beginning of the search procedure is typical for the DMRP approach. The initial reference point falsifies a relatively high number of clauses: The red line starts at $(0, 793)$ and decreases rapidly within the first few decisions. After 69 decisions, there are 158 clauses; after 726 decisions, there are less than 20 clauses that are falsified by the point.

A closer look at the lower left corner of the plot shows that the temporary point sometimes exhibits smaller values than the reference point. This is due to a strategy of the solver that we introduced for the following reason. We observed that highly frequent but minor modifications of the reference point often produce a big computational overhead (from line 13 in `solveDMRP`, Algorithm 5.1). In these cases, it turned out to be more effective to modify the reference point not immediately but after a delay of one decision in the search. This bundles many small modifications into one bigger modification of the reference point, particularly after the initialisation of a new point that is then improved steadily. However, not all improvements of the reference point are detected by one decision. As the upper part of the plot indicates, the decision level reaches a depth of 84 at one stage to find an improvement of the reference point. Overall, the depth of the search is quite low, since the instance is rather easy to solve.

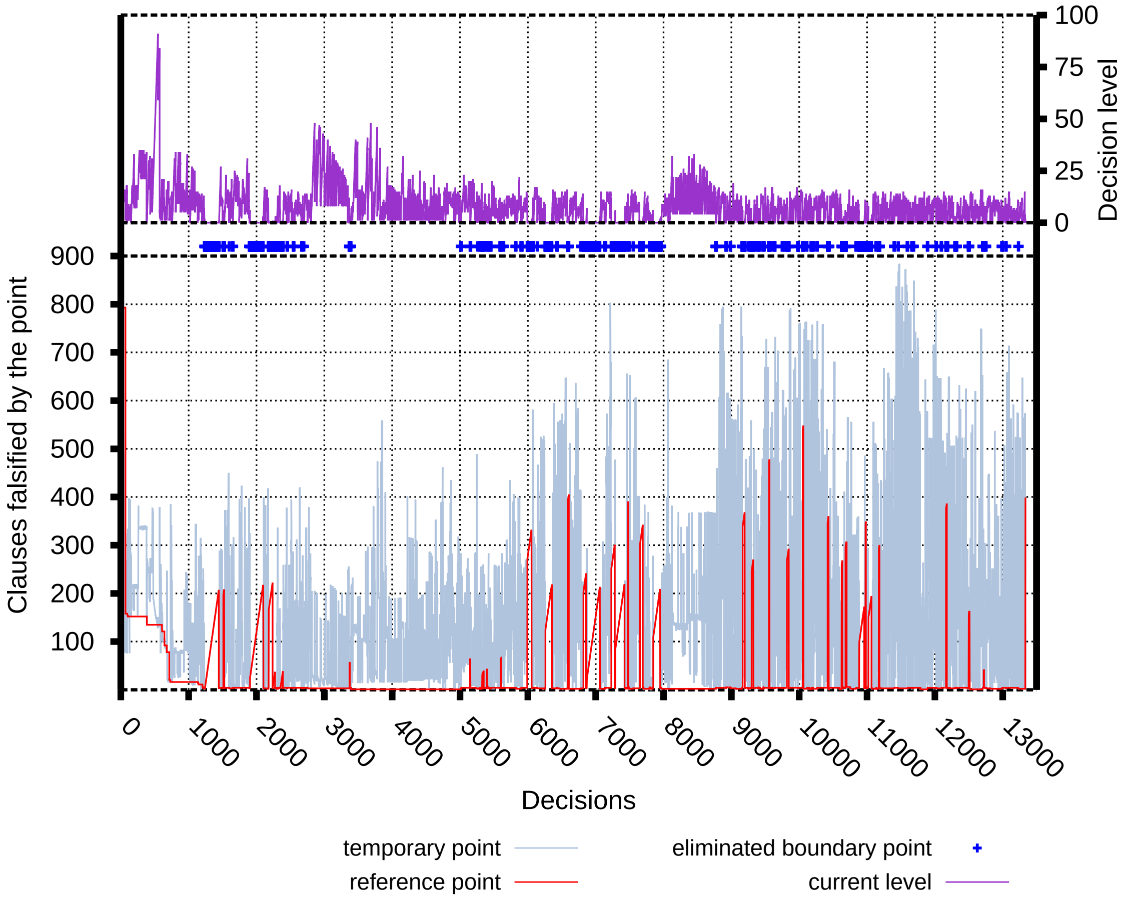


Figure 5.3: Progress of DMRP on the benchmark *cmu-bmc-barrel6*

After 1200 decisions, the reference point falsifies only very few clauses (less than 10). Henceforward, many boundary points are found on–the–fly, as described in Section 5.3.2. The blue points in the plot state that there are some phases in the search where several boundary points are eliminated. In total, 1957 boundary points are eliminated during the entire solving process. It is clearly visible in the plot that the reference point deteriorates from time to time. This happens whenever the solver learns a unit clause, which forces a modification of the reference point. Moreover, the elimination of a boundary point creates a new clause that is also considered by the grey and the red curves in the plot.

The latter case is visible when several boundary points are eliminated in a sequence of decisions. From decision 6870 to decision 7060, the number of clauses falsified by the reference point increases from 22 to 212. In each step, a boundary point is eliminated and the effect is clearly visible by the slope of the red curve.

An interesting issue is the behaviour of the grey curve, even though it is impossible to follow it exactly. Between decision $11,000$ and decision $12,000$, the reference point falsifies only a few clauses (3) while the temporary point falsifies several clauses (up to 883). This is characteristic of the DMRP procedure, especially for harder instances as search progresses. At the same time, for the plotted instance, the decision levels are quite small as indicated by the purple curve. In the end, 2418 modifications of the reference point are required before the empty clause is learnt.

The application of DMRP, as presented above, indicates a high number of boundary point eliminations. However, there are also many instances where no boundary point is eliminated during the search process. Note that we do not search for boundary points in particular within DMRP but only eliminate them if they are detected on–the–fly. Figure 5.4 depicts the elimination of boundary points for five DMRP configurations. These configurations differ in some parameter settings, such as the probability for random decisions and the selection strategy for decision clauses. However, the specific differences are not important here. The numbers are based on the results for 140 benchmarks of the SAT competition 2011, which were solved by at least one of the plotted configurations. The left–hand plot states the total number of boundary point eliminations and the right–hand plot relates this number to the number of decisions needed to solve an instance. On the x–axes, instances are sorted by their corresponding values in descending order. Hence a point (x, y) in the plot (a) states that for x instances, at least y boundary points were eliminated. In plot (b), (x, y) states that for x instances, the average number of eliminations per decision is at least $y/1000$. Both y–axes use a logarithmic scale.

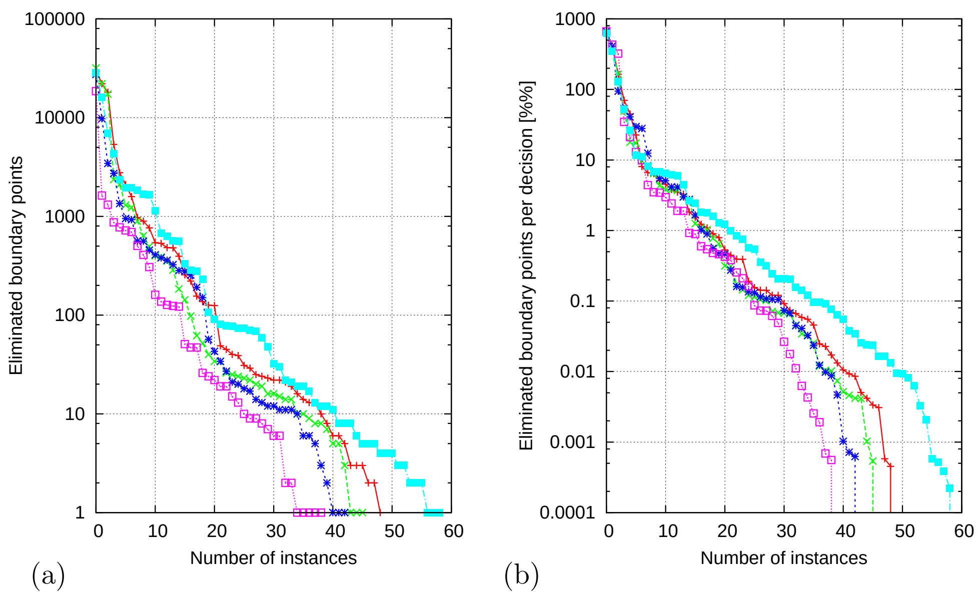


Figure 5.4: Boundary point elimination within DMRP

Clearly, there are some instances where many boundary points were found on–the–fly and could be eliminated. Plot (a) indicates that there are 20 benchmarks for which the best configuration eliminated more than 100 boundary points. Moreover, the right–hand plot indicates that for 20 benchmarks, a boundary point was eliminated at least every 1000th decision by the best configuration. However, considering that both plots zoom in on the best 60 instances, the overall application of boundary point elimination is fairly low. Moreover, this does not justify the alternative computation of MakeCounts, to count the total number of clauses that are falsified by the temporary point at each decision level (see Section 5.3.2).

5.4.2 Runtime comparison

For the evaluation presented in the remainder of this chapter, we ran our solver for all industrial (application) benchmarks of the SAT competitions and the SAT–Races of the years 2006–2009 [Sat10, Sat11]. By removing trivial instances, which can be solved by the preprocessor, the benchmark set contains 564 instances. Each instance was preprocessed in advance and the timeout for each solver run was set to 1200 seconds. As a reference and to validate the results, we have run our CDCL solver SApperloT using the Luby restart strategy (without DMRP) and MiniSat2.0 [ES03, ES12].

Figure 5.5 depicts a cactus plot to compare the runtime of several different solver configurations. It illustrates the results of different configurations of the hybrid approach. Furthermore, two configurations apply the DMRP procedure alone. The configurations differ in the following regards, which are related to the details of decision making: As mentioned above, the Make-Count of a variable may consider all clauses in $\mathcal{D} \cup \mathcal{M}$ (global) or only clauses in $\mathcal{D}$ (local). If two variables have the same MakeCount, ties are broken in favour of the variable ν_i that:

(act) has the highest activity value, similar to the VSIDS heuristic [MMZ+01].

(bc) has the smallest set $R(v)$ (cf. Section 5.2.2). This can be seen as a simple approximation of the *BreakCount* of a variable. Unlike the MakeCount, the BreakCount of a variable ν_j states the number of clauses that are falsified by a flip of the value of variable ν_j.

(dc) was chosen least often for DMRP decisions. This avoids always flipping the same variables back and forth in different calls to `solveDMRP`.

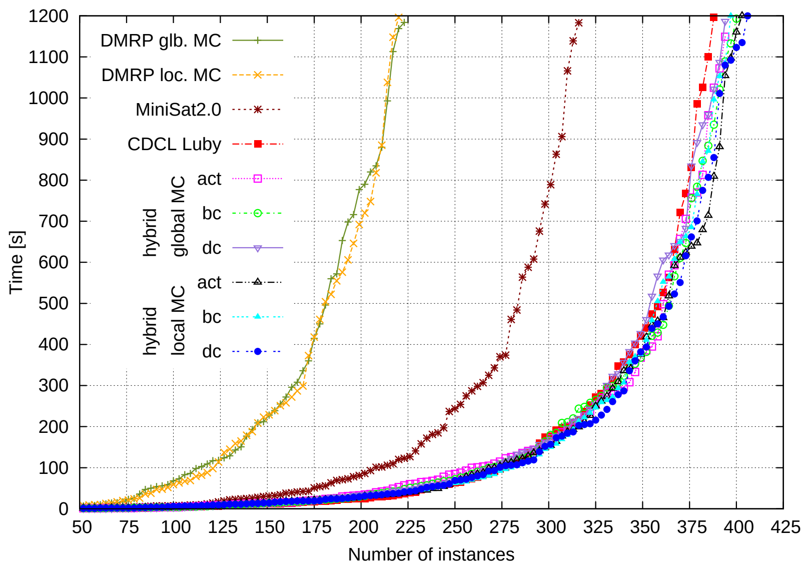


Figure 5.5: Comparison of DMRP, CDCL and their hybrid

Figure 5.5 clearly shows that pure DMRP solving cannot compete with CDCL solving. Both the global (olive green) and the local (orange) DMRP configurations solve around 224 of 564 instances within 1200 seconds. The initial reference points are always chosen at random. Timeouts for the search on one reference point (one call to `solveDMRP`) follow the Luby sequence [LSZ93]. Both plotted DMRP configurations keep well behind MiniSat2.0 (brown). Note that MiniSat2.0 neither uses the Luby restart strategy nor applies phase saving (cf. Section 2.2.5). These are the two main reasons why it exhibits a much worse performance than our implementation of the CDCL procedure (red).

The hybridisation of DMRP and CDCL, as described in Section 5.3, outperforms both the pure DMRP and the pure CDCL solvers. All hybrid configurations indicate quite similar results. However, the two configurations (black, blue) that compute the MakeCount locally seem to be ahead for most time bounds. For the time bound of 1200 seconds, the blue configuration is able to solve the greatest number of instances. It is interesting to observe that using the activity of variables to break ties does not achieve the best results. It turns out that it is better to prefer variables that were flipped least often at the current call of `solveDMRP`.

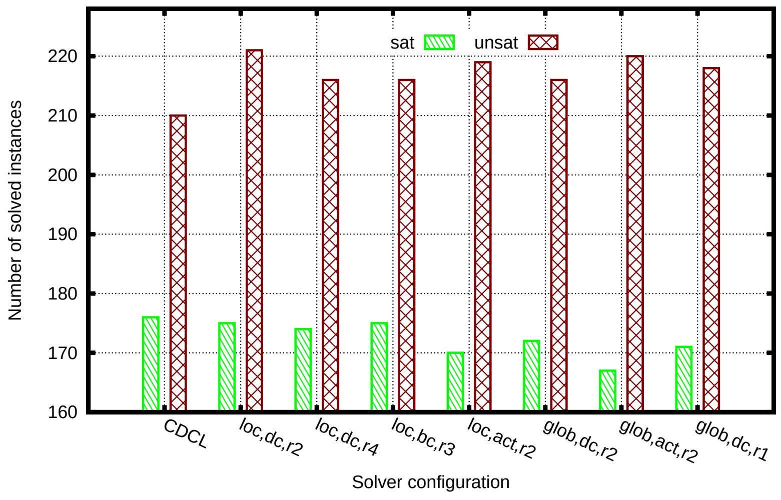


Figure 5.6: Results for satisfiable and unsatisfiable instances

The cactus plot in Figure 5.5 gives a good overview of the overall success of our hybrid approach. However, it is worth analysing the performance of the hybrid solver in more detail. Figure 5.6 distinguishes between satisfiable and unsatisfiable benchmarks. For each solver configuration, the number of solved satisfiable (unsatisfiable) instances is indicated by the green (brown) bar. The hybrid configurations are differentiated by three parameters: the first parameter states whether the MakeCount is computed locally or globally. The second parameter indicates the heuristic to break ties for variables with equal MakeCount. The last parameter rn states that in n percent of all decisions, a variable is chosen at random from a falsified clause. The presented values are based on the solver without the application of preprocessing.

Surprisingly, pure CDCL solving outperforms all hybrid configurations when only the satisfiable instances are considered. It solves one satisfiable instance more (176) than the two best hybrid configurations (175). In turn, the success of the hybrid approach is only due to the unsatisfiable instances. The leftmost hybrid configuration (*loc,dc,r2*) solves 11 more unsatisfiable instances than the CDCL solver. However, the success on the unsatisfiable benchmarks is not achieved with only that configuration. All hybrid configurations solve clearly more unsatisfiable instances than the CDCL solver.

Our conjecture about this phenomenon is that DMRP generates some important conflict clauses. When the CDCL solver reaches the (current) maximum number of conflicts, it delivers work to the DMRP solver. DMRP starts by extending the last partial assignment τ of the CDCL solver to a complete assignment that constitutes the initial reference point $\mathcal{P}$. In doing so, it focuses on a nearby part of the search space. When analysing this part, it purposely examines the clauses $\mathcal{M}$ that are falsified by $\mathcal{P}$. In CDCL solving, these clauses in $\mathcal{M}$ can become conflicting clauses if decisions similar to the values in $\mathcal{P}$ are made. Up to a certain point, phase saving will do precisely this after a normal CDCL restart. However, DMRP immediately considers clauses in $\mathcal{M}$ for search and resolution.
Even though the hybrid implementation beats the pure CDCL solver on the entire benchmark set, the CDCL part of the hybrid solver returned the result for the greatest number of instances. On average, the result was returned by the DMRP part for only 6% of all instances. However, this does not reduce the contribution of the DMRP part, as the success of the hybridisation over pure CDCL solving indicates.

A further differentiation of results is presented in Table 5.7. The benchmark set can be categorised into families of different instances, based on the submitters of the instances or the encoded application. The first column of the table lists the benchmark family and the number of instances it contains. Subsequent columns differentiate several hybrid solver configurations, whereas the naming of parameters is the same as in the figures above. An entry in the table states the number of instances of a certain benchmark family that are solved by the corresponding solver configuration. The highest number in each row is printed in bold.

Table 5.7 presents the results for around half of the benchmark families. The number of solved instances per benchmark family varies by a small number for most solver configurations. There is no configuration that is best throughout all families. However, for domain–specific solving, it may be worth finding a proper configuration. The table indicates that there are also families of instances for which pure CDCL seems to be most successful, such as the Manol–pipe and Gss instances.

Instances	#	CDCL	local MakeCount				global MakeCount		
			dc,r2	dc,r4	bc,r3	act,r2	dc,r2	act,r2	dc,r1
9dlx_vliw	11	2	**5**	4	4	4	4	4	4
AproVE	34	26	**27**	25	26	**27**	26	**27**	26
Dspam	10	8	**10**	**10**	**10**	**10**	**10**	**10**	**10**
Gss	23	**7**	5	6	5	5	4	5	5
IBM	38	**34**	31	31	32	**34**	32	30	32
Manol–pipe	59	**53**	51	51	52	52	52	52	51
Max_min	15	7	7	**8**	**8**	**8**	7	**8**	**8**
Narai	4	3	**4**	**4**	3	**4**	3	3	3
Partial	20	4	5	8	**9**	4	5	7	8
Post	15	6	7	**8**	7	6	7	**8**	6
Q_query	20	**20**	**20**	19	19	**20**	19	19	19
Simon	12	10	10	9	9	9	9	10	**11**
Total	20	13	**14**	13	13	13	13	13	13
Velev pipe	23	16	**21**	**21**	**21**	**21**	**21**	**21**	**21**
Velev vliw	14	8	**11**	10	10	10	10	**11**	10

Table 5.7: Performance for different families of instances

On the other hand, the results for the hardware verification instances of Velev are remarkable. Figure 5.8 depicts the cactus plot for the subset of 53 *Velev* instances in our benchmark set. The two plotted hybrid configurations correspond to the solvers that are listed in the third (*loc,dc,r2*) and the eighth (*glob,act,r2*) column of Table 5.7. The cactus plot illustrates the clear advance of both hybrid configurations using these families of instances. In total, eight more instances are solved by the hybrid approach than by pure CDCL.

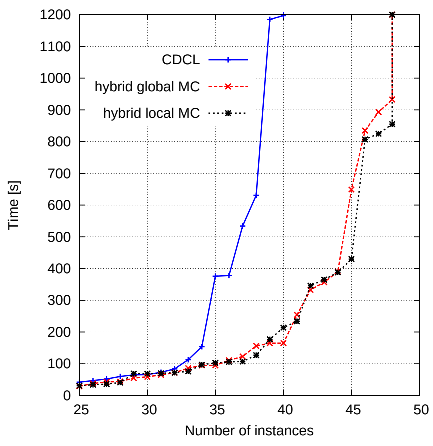


Figure 5.8: Comparison on hardware verification instances of Velev

5.5 Summary

In this chapter, we studied the application of the DMRP SAT solving approach. The idea of using a reference point for decision making in SAT solving has been proposed by Goldberg [Gol08a, Gol08b]. A reference point constitutes a complete assignment to the variables of the formula. At any time in a search, the solver distinguishes between those clauses that are satisfied by the reference point and those clauses that are falsified by the point. The solver aims to reduce the set of falsified clauses by modifications of the reference point. Unlike a greedy strategy, none of the clauses that are satisfied by a reference point must be falsified by a subsequently modified reference point. To this end, the DMRP approach requires more information during the search than the conflict–driven SAT solving algorithm.

We presented a data structure to implement the DMRP approach in an efficient way. Similar to the two watched literals scheme, one literal is selected for each clause. That literal takes responsibility to ensure that any clause which is satisfied by the reference point is also satisfied by a modification of the point. Moreover, we described how the maximal MakeCount of a variable can be computed by using this data structure. The MakeCount of a variable ν_i counts all falsified clauses that become satisfied when the value of ν_i is flipped in the reference point. We also studied the elimination of boundary points within the DMRP algorithm. Unlike the solving approach that is entirely based on the elimination of boundary points [GM10], our extension to DMRP only finds and eliminates boundary points on–the–fly.

Based on our DMRP implementation, we developed a hybrid SAT solver that combines CDCL and DMRP solving. The combination of both approaches uses the values of the variables that are stored by the phase saving heuristic in modern CDCL solvers. These values are employed to initialise the reference point for the DMRP algorithm. Experiments indicate that our hybrid approach is competitive to the highly optimised state–of–the–art CDCL solvers, and that the maintenance of complete assignments may definitely be an advantage.

Our solver SApperloT participated in the SAT–Race 2010 [Sat10]. It applied the presented hybrid approach and used an additional preprocessor implemented by Zielke [Zie10, Kot10b]. SApperloT did not win a prize in that competition. However, it outperformed both participating versions of Glucose [AS12]. Glucose had won two medals in the SAT competition of the previous year. In particular, in the competition of 2009, it had been placed first on the set of unsatisfiable application benchmarks.

We will come back to the DMRP approach in Chapter 8 where the work of this thesis is concluded. We will also give some ideas and directions for future research related to the concept of reference points and DMRP solving.

Chapter 6

SAT Solving with Multiple Cores

In the previous chapters, several concepts and approaches related to practical SAT solving are discussed. Alternative solving techniques are often successful for some families of SAT instances but are clearly inferior to CDCL on the average set of benchmarks. For the DMRP algorithm, a convenient hybrid approach is suggested in Section 5.3 to combine the predominant CDCL technique with the powerful but more expensive DMRP algorithm. Other techniques, such as the concept of asymmetric branching presented in Section 3.2, can only be applied selectively since an exhaustive application is too time–consuming. Moreover, the combination with hyper–binary resolution often generates too much redundant information. A general method to combine simplification with CDCL solving is realised by PrecoSAT [Bie09b], CryptoMiniSat [Soo12] and Lingeling [Bie11] where simplification is applied as preprocessing and inprocessing.

The engineering and analysis of different powerful solving and simplification techniques motivates a combination of the presented approaches. In the era of multi–core architectures, parallel programs may take advantage of real concurrency. In this chapter, we present the implementation of our multithreaded SAT solver SArTagnan. The architecture of the solver was designed to meet the following demands:

- Alternative solving methods and simplification techniques (as in Chapters 3 to 5) run in different threads and exchange valuable information.
- The application of more expensive techniques is worthwhile due to their use for several parallel solvers.
- The facility to share data physically on multi–core architectures is used intensely.
- The exchange of information and the sharing of data are non–blocking.

In the presented solver, all threads are allowed to share clauses logically and physically. However, the set of clauses in different threads is not required to be identical. Any solving thread can still decide whether it uses a shared clause of another thread and whether it shares an own clause with other threads. Even though data is shared among all threads, the solver does not use any locks of the operating system or OpenMP [Ope12]. Each solving thread can be configured to apply different search strategies. Due to the physical sharing of clauses, any solving thread can permanently improve the entire set of clauses. Moreover, all threads may benefit from the improvement that was made by one thread.

The chapter is organised as follows. Section 6.1 addresses the parallelisation of the solver. Section 6.1.1 gives an overview of state–of–the–art approaches in parallel SAT solving. Thereafter, the most important techniques for the parallelisation are presented, such as some issues regarding the physical sharing of clauses and the communication between threads. In Section 6.2, different search strategies of the solving threads are related to the previous chapters. The subsequent section presents experimental results and gives an insight into configuration details. The presented work is published in [KK11c, KK11b].

6.1 Parallel Solving

The engineering of practicable SAT algorithms and the intensive optimisation of SAT solvers have made the SAT problem feasible for many computational real–world problems that can be transformed into SAT formulae. The design of efficient data structures and optimised implementations [MMZ$^+$01, ES03] has been ground–breaking for the wide application of SAT solving. In recent years, parallel SAT solving has gained in importance to utilise the potential of multithreaded architectures. However, the parallelisation of SAT solving has been an interesting research area for a long time and different approaches to parallelise SAT have been studied.

6.1.1 Related work

A search procedure may split the entire search space into different subareas that may or may not be disjoint. This approach seems to suggest itself in SAT solving where a subset of variables can be preassigned to different values for different parallel solving procedures. In doing so, different parts of the search space will be explored. This divide–and–conquer approach, which is often said to use a so–called *guiding path* [ZBH96], is widely used in distributed parallel SAT solving. On multi–core architectures, the guiding path approach can be realised by the application of dynamic work stealing. An inactive thread requests work from any active thread. The active thread divides its own guiding path into two paths one of which is given to the requesting thread. This idea is used by solvers as pMiniSAT [CHS09], MiraXT [SLB05] and ySAT [FDH05].

On the contrary, there is the parallel SAT solving approach that does not guide different solving processes in any way. Even when running the same algorithm in parallel, the use of different heuristics [BSK03a, HJS09b] and some random decisions will lead each process in different directions. Useful information that is globally valid for each solving process may be exchanged. Blochinger *et al.* proposed the idea to exchange learnt clauses between parallel executions of the DPLL algorithm [BSK03b]. These days, most parallel state–of–the–art SAT solvers apply this idea to a certain extent.

In the context of multithreaded SAT solving, sharing of learnt clauses is now widely applied. However, the term *sharing* may be ambiguous. Most solvers run an instance of the CDCL algorithm in each parallel thread. A copy of any learnt clause that conforms to certain criteria is sent to the other parallel threads [HJS09b]. Hence information is shared among parallel threads but a clause is not shared physically. Each thread holds its own copy of each clause. To our knowledge, the only solvers that share a unique clause database physically are MiraXT [SLB05, LSB07] and ySAT [FDH05], whereas the latter does not share learnt clauses physically. For more profound details we refer the reader to the related publications.

6.1.2 A basic concept for multithreading SAT

Working on shared memory architectures motivates for sharing clauses physically such that shared information (i.e. each clause) exists only once in memory. This is motivated by the fact that the set of literals of a clause is basically static. Moreover, sharing clauses physically allows for a better exchange of additional information as, for example, the subsumption or backward subsumption of clauses. We first present the basic concepts used to exchange and share information during SAT solving. In Sections 6.1.3 and 6.1.4 we then go into more detail.

We define the number of parallel threads to be t. And we refer to a particular thread as T_i, where $i \in [0, t-1]$. During programme execution each thread holds a unique user mask $M(T_i)$ that is defined to be 2^i. Each data object that is shared by several threads has a bit mask $usrs$ that indicates the set of users of this object. The value of $usrs$ can be formalised as $usrs = \Sigma_{T_i \in U_j} M(T_i)$, where U_j is the set of threads that link to the object O_j. In general, the $usrs$ field of an object O_j is always initialised by the creating thread before O_j is visible to the other sharing threads. After the creation of an object O_j, a reference to O_j is sent to all sharing threads

Algorithm 6.1: Release object by thread T_i

```
Require Reference of object $O$. Calling thread is $T_i$
Function releaseObject ($O, T_i$)
    $inv\_msk \leftarrow \ \sim M(T_i)$                | inverted $M(T_i)$
    repeat
        $curr \leftarrow O.usrs$                      | copy bit mask
        $rem \leftarrow curr \ \& \ inv\_msk$          | remove $M(T_i)$
    until exchangeIf ($O.usrs, curr, rem$)
    if $rem = 0$ then deallocate ($O$)
```

$\in U_j$. As soon as a thread that uses O_j wants to release the object, it has to unsubscribe as a user of O_j. The last user is responsible for the destruction of O_j. The release operation is listed in Algorithm 6.1. Note that no thread can ever add itself as a user to an already created and shared object.

The function `exchangeIf`$(addr, assum, new)$[1] is a typical atomic operation that replaces the content at the specified address *addr* by the value *new* but only if the current content is equal to *assum*. It returns *true* iff the exchange operation was successful. The application of user masks is actually very similar to the concept of semaphores that use a simple counter initialised to the number of users. However, there is a good reason for the user masks. For any shared object, the set of its users can be determined easily. This turns out to be extremely useful for heuristics on the exchange of data between several threads.

6.1.3 Physical sharing of clauses

The concept of user masks as described in the previous subsection, is used to realise the sharing of clauses that have more than two literals. In SAT solving, each clause basically represents a static set of literals. However, most state–of–the–art solvers implement the two watched literals scheme [MMZ+01] in the way it was suggested by Van Gelder [Gel02]. The two watched literals of a clause are always placed in the first two positions of the array of literals. Thus, the position of literals in a clause is permuted permanently. This idea cannot be implemented when a clause is shared, since the two watching literals may differ in different solving threads. This also disqualifies the `XOR`–watchers implementation, as it is presented in Section 3.1.

However, the ideas and results of Section 3.1 can be adapted for the

[1]In the GNU compiler the atomic function is provided by:
bool __sync_bool_compare_and_swap(addr, assume, new_val)

shared–memory architecture. With Corollary 3.1.1, the two watched literals λ_p and λ_q of a clause C can be saved by one value $\mathcal{X}(C) := ID(\lambda_p)$ `XOR` $ID(\lambda_q)$. With this, the set of literals of a clause can be shared among several parallel solving threads by keeping the value $\mathcal{X}(C)$ of a clause C locally for each thread. The set of literals of C needs only to be read but never to be written by an accessing thread. Thus, in our implementation any clause C with more than two literals consists of the value $\mathcal{X}(C)$ and a pointer to the shared set of literals L. To realise sharing and, in particular, destruction each set of shared literals has a user mask, as described in Section 6.1.2.

A motivation for sharing clauses in parallel SAT solving is to allow for sharing additional important information about the state and the modification of a clause. If the set of literals of a clause is reduced by simplification techniques, such as self–subsuming resolution (see Section 3.2.1) or on–the–fly clause improvement [HS09], it is desirable that any other thread may benefit from this information.

In SArTagnan, we realise this by sending a new version of a clause to all threads that share this clause. As soon as a new version C_n is sent, the previous version C_o of the clause is marked to be redundant by setting a particular bit flag in the clause. However, C_o is still valid and can still be used by other threads. Redundant clauses are released when a thread cleans its set of clauses. This requires every thread to be able to communicate with any other thread by sending and receiving messages. In this context, two issues are crucial:

- Regard the order. A new version of a clause has always to be sent before the previous version of this clause is marked to be redundant. Furthermore, after the release of a redundant clause, the solver has to check for messages from all other threads.
- Messages between threads must never be lost. A message of any sender must be visible to all receivers immediately, so that a new version of a clause is ensured to be visible not later than the old version is marked to be redundant.

The realisation of the message system is presented in the next subsection.

6.1.4 Communication of threads

In this subsection, we present the implementation of lossless queues used for the communication between threads. Moreover, receive and send operations are non–blocking. A common concept to exchange messages within concurrent programmes uses non–blocking circular queues: sender and receiver use a shared array of fixed size n and the next writing and reading positions (*write*/*read*) are both visible to the sender and to the receiver of the queue. A write operation changes the value $write \leftarrow write + 1 \text{ MOD } n$. The increment of the *read* value is analogous. The queue is empty if the values of *write* and *read* are equal. If the queue contains $n - 1$ elements, a push operation to the queue will not be successful since this clears the queue. However, this violates the soundness condition of our solver. Based on the described concept, we present non–blocking queues that allow for a thread–safe extension of allocated memory to ensure that write operations are always successful. The idea is related to the technique described by Hendler *et al.* [HLMS06]. Note that concurrent data structures presented for programming languages that use garbage collection [HS08] cannot always be adapted for manual memory management straightforwardly.

Linking Updates

If an object is shared among different threads, the object's data may not be modified concurrently, since there is no way to perform the modification by an atomic operation. A straight solution to this problem is to provide a link to a new version within the object itself.

Algorithm 6.2: Link new version O_n as update of object O_o

```
  Require Object $O_o$ to be updated by a new version $O_n$
  Function linkUpdate ($O_o, O_n$)
3     $O_o.udt \leftarrow O_n$          | copy reference to new object version
      repeat
          $curr \leftarrow O.usrs$                       | copy bit mask
6         $udtd \leftarrow curr \,\&\, udt\_msk$        | indication for an update
      until exchangeIf ($O.usrs, curr, udtd$)
```

A new version O_n of an object O_o is initialised before it is finally linked as a new version in Algorithm 6.2. The procedure must only be invoked by the owner of an object. In line 3, the link *O.udt* of an object is used to link a new version of this object. Note that this link is not used as an indication for an update. To indicate that a new version for an object exists, a special user mask *udt_msk* is used ($udt\_msk \neq M(T_i)$ for all threads), as shown in line 6. This ensures consistency when an update is linked while another thread releases the same object concurrently. Otherwise, a thread t_i may be set as a user for a new version O_n of an object O_o, whereas t_i releases object O_o at the time when the new version is linked. Hence t_i will not know about the object O_n but will be registered as a user of O_n. Thus the allocated memory will never be released.

Algorithm 6.3: Load update of object O_k by a thread T_i

```
Require Reference of object $O$
Return Next version of object $O$ if available
Function loadUpdate ($O, T_i$)
    $O.udt$        | May link to an update, set by Algorithm 6.2
5   if $\neg$ hasUpdate ($O$) then return $O$          | check $udt\_msk$
    $O_{new} \leftarrow O.udt$                          | copy reference
    releaseObject ($O, T_i$)                        | see Algorithm 6.1
    return $O_{new}$
```

Algorithm 6.3 shows a simple procedure to load an updated version of an object if one is available. As mentioned above, the check for an update does not use the link of an object *O.udt*. In line 5, the function `hasUpdate` rather checks if the bit *udt_msk* is set in *O.usrs* such that all operations related to memory are managed by atomic accesses to the bit mask *O.usrs*. The presented concept is used to make a queue extendable by its writing thread. Basically, both the reading and writing thread share the same data array for communication.

The operation to push data to a queue is outlined in Algorithm 6.4. The actual data stored in the queue is wrapped into an object of type `DataArray`, as listed in line 1. As in circular queues, the *read* field is incremented by the consuming process and the *write* field is incremented by the producing thread when data is put into the queue. As mentioned above, a push operation can not be applied if the *write* value is equal to the *read* value after the push, since this indicates that the queue is empty. To allow for non–blocking write operations, the producing process may link a new object of type `DataArray` if a normal push is not possible. For this reason, the queue keeps two references to the wrapped `DataArray`, one for the consuming and one for the producing process (line 8). Usually both refer to the same object in memory. However, if an update is linked by the producing thread, the consuming thread refers to a previous version until all data is consumed.

A push to the queue increments the *write* value in line 11 but the modified value is not stored in the object before the data is actually put into the queue. If a normal push is not possible (line 12), a new `DataArray` is constructed in line 13. The data is put into the new array before it is linked as an update (in line 17) and the reference to the current `DataArray` is released. Line 20 handles the normal case when data can be pushed immediately.

The procedure to read data from a queue is outlined in Algorithm 6.5. Unlike the push operation, the pop operation uses the reference C to the `DataArray`.

Algorithm 6.4: Non–blocking push to queue

```
1  Class DataArray
       udt                                   | Pointer to a new version
       read                            | next read operation in data
       write                          | next write operation in data
       size                                          | size of data
       data ...                              | actual data in the queue
8  | a queue has two references to objects of type DataArray:
   $C$ (consumer) , $P$ (producer)
   Require Thread $T_i$ pushes data $D$ to queue
   Function pushQueue $(T_i, D)$
11     $next\_w \leftarrow P.write + 1 \bmod P.size$
12     if $next\_w = P.read$ then
13         $N \leftarrow$ construct new DataArray
           $N.data[0] \leftarrow D$;
           $N.read \leftarrow 0$                   | next read for consumer
           $N.write \leftarrow 1$;                | next write for producer
17         linkUpdate $(P, N)$       | link new version (Algo. 6.2)
           releaseObject $(P, T_i)$     | unregister for old version
           $P \leftarrow N$
20     else
           $P.data[P.write] \leftarrow D$;
           $P.write \leftarrow next\_w$;
```

The crucial point to notice is the double check in lines 4 and 6 whether the referenced queue is empty. If the reading thread cannot pop any data from the queue (in line 4), it checks for an update of the DataArray C in line 5. If no update is available, it returns immediately.
However, if an update is available, it is not ensured that all data of the current version of C was actually read. Consider the following case with writing process W and reading process R. After the check in line 4 of Algorithm 6.5 by process R, the writing process performs several consecutive pushes to the queue until no more write operations are possible. It will thus create a new DataArray object and link it as new version in line 17 of Algorithm 6.4. If, for some reason, the reading process got stuck between the lines 4 and 5 of Algorithm 6.5, the check in line 5 indicates a new version of the data array. However, there is some unread data in the DataArray object that is still referenced by C. Even though this course of events seems unlikely it may and does happen in practice, especially when real concurrency comes into play.

Algorithm 6.5: Non–blocking pop from queue

```
   Require Thread $T_i$ reads next data $D$ from queue
   Return $true$ if new data was read, $false$ otherwise
   Function popQueue $(T_i, D_{out})$
4      while $C.read = C.write$ do
5          if $\neg$ hasUpdate $(C)$ then return $false$
6          if $C.read = C.write$ then
7              $C \leftarrow$ loadUpdate $(C, T_i)$
8      $D_{out} \leftarrow C.data[C.read]$
       $C.read \leftarrow C.read + 1 \bmod C.size$
       return $true$
```

If the reading process reaches line 6, it is ensured that W only operates on newer versions of the data array since it linked an update. Hence in line 7, it is ensured that no data is missed by the reading process. The actual read operation is finally performed in line 8.

The described queue can be extended to serve more than one reading process. In that case, there is one reference of the data array $C_1 \ldots C_k$ for each reading thread. Thus it is sufficient to have one queue for each solving thread where it can write messages to all other threads concurrently. This implies that all messages of one sender will be read by all other threads. This is desired and necessary if non–redundant clauses are sent but may be undesirable for optional information, such as learnt clauses.

For this reason, every message contains an additional recommendation. This is basically a user mask where the users are marked, for which the message is assumed to be interesting. Messages that contain newly learnt clauses, do not mark any user. On the other hand, for non–redundant clauses every user is marked. However, if a redundant (learnt) clause C_o is improved (e.g. its set of literals is reduced), a new clause C_n ($|C_n| < |C_o|$) is created and sent to all other threads. The recommendation is set to the user mask of C_o. C_n is marked as learnt if C_o was learnt. Any receiver considers the recommendation of a message for the heuristics to decide whether a clause is imported or whether it is released immediately. Clearly, different solver or simplification approaches may use different criteria for the import of clauses.

6.2 Utilising various Approaches

The functionality of physical clause sharing and the lossless communication between threads allow for heterogeneous SAT solving. The ability to share the entire set of clauses of all threads allows for several simplification techniques. This constitutes an advantage over parallel solving where each thread has its own copy of the clause database. Every thread may benefit from a simplification of the clause database. If, for instance, a thread reduces the set of literals of a clause by some simplification technique it can post the result immediately to all other threads. In general, progress made by one thread may be beneficial for several other threads. Besides the approaches presented in the previous chapters, our parallel solver uses some more techniques, such as blocked clause elimination or autarky detection. These techniques are mentioned but we refer the reader to the original publications for detailed descriptions.

6.2.1 The simplification thread

One thread of SArTagnan is mainly dedicated to simplifying the entire clause database. It imports most of the clauses that it receives from all other threads. It performs basic simplification techniques as subsumption, backward subsumption and self–subsuming resolution and aims to eliminate variables, as it is applied by common preprocessors (see Section 3.2.1 and [SP04, EB05]). Moreover, it eliminates blocked clauses, as suggested by Järvisalo *et al.* [JBH10]. A blocked clause constitutes a specific redundant constraint [Kul99a, Kul99b] that can be detected by inspecting all clauses where a particular literal occurs in. This process can be incorporated into variable elimination. Equal variables are detected by searching for strongly connected components in the binary implication graph [APT79, Bra01].

Only the simplification thread is allowed to decide on variable elimination, the replacement of equal variables and the deletion of blocked clauses. All three techniques are critical in terms of concurrent application. Granting these simplification techniques only to one thread is a safe way to ensure the soundness of the parallel solver. If, for instance, two threads were allowed to perform variable elimination, one had to ensure that the concurrent elimination of different variables is independent of each other. In particular, it has to be avoided that the elimination of a variable in one thread generates a new clause that contains a variable that is eliminated by another thread at the same time. Clauses that are removed as blocked clauses or within variable elimination are kept in an extra list η which can be read by any thread. If a thread finds an assignment that satisfies all clauses it can reimport the clauses of η to compute a model for the formula.

If a variable is eliminated or detected to be equal to another variable, new clauses are constructed and sent to all other threads. In case of variable equality, one variable r is chosen to be representative for the set of variables E_r that are equal to r. For each clause that contains a variable of E_r a new clause is created and sent to all threads and the original clause is marked to be redundant. As soon as all replacements are performed a particular message is sent to the other threads.

An important task of the simplification thread is the detection of equal or subsuming clauses. The fact that any thread is allowed to create and send improved versions of a clause may introduce duplicate clauses. These duplicates are detected and removed by common subsumption checks within the simplification thread. However, to avoid an unnecessarily large number of duplicates, every solving thread obeys to the following rule: if a clause C_o can be improved (i.e. some literals may be removed), a new version of the clause C_n is sent to the other threads only if C_o was not already marked to be redundant.

Guided search for autarkies in the SArTagnan version 2010

An autarky is a partial variable assignment that can safely be applied to a formula $\mathcal{F}$. The resulting formula $\mathcal{F}'$ is equisatisfiable to $\mathcal{F}$, i.e. $\mathcal{F}'$ is satisfiable iff $\mathcal{F}$ is. However, if $\mathcal{F}$ is satisfiable, the set of models for $\mathcal{F}$ and $\mathcal{F}'$ may be different [Kul00].
The simplification thread searches for autarkies, whereas hints are given by other threads: whenever conflict analysis within a CDCL search determines to jump back over several decision levels, then none of these decisions contributes to the conflict. At this point, a CDCL thread sends a particular message to the simplification thread, indicating the variable assignments it jumped over. The simplification thread may check whether a subset of these assignments is autarkic to the entire formula. However, these checks are performed with little priority by the simplification thread.

Extensive asymmetric branching in the SArTagnan version 2011

The concept of asymmetric branching is an effective but rather costly simplification technique that can be used in different solvers, such as MiniSat [ES12], PrecoSAT [Bie09b], SApperloT 2009 [Kot09] and CryptoMiniSat [Soo12]. As described in Section 3.2, the common technique has the drawback that it depends on the order of propagation. This drawback is overcome by using the idea proposed in Section 3.2 in a slightly modified version. Moreover, the concept of hyper–binary resolution is incorporated into the asymmetric branching procedure, as described in Section 3.3.

6.2.2 DMRP threads

Decision making with a reference point is an alternative SAT solving method proposed by Goldberg [Gol06, Gol08a]. It spends more effort on decision making than usual CDCL solving. The DMRP algorithm holds a complete assignment (a so–called reference point) to the variables and considers those clauses for decision making that are falsified by the reference point. A competitive implementation of the approach is presented in the previous chapter where also some modifications of the original heuristic are analysed.

We already pointed out that clauses which are learnt during the DMRP algorithm seem to be more valuable than clauses that are learnt during CDCL. This motivates the sharing of clauses that are generated within the DMRP algorithm. In our configuration, one thread solely applies DMRP solving as described in Chapter 5. However, in the parallel context it implements a crucial modification for the initialisation of the reference point:

As in the hybrid approach presented in Section 5.3, a restricted kind of search is applied that considers the binary clauses of the formula $\mathcal{F}_2 \subseteq \mathcal{F}$. A model for $\mathcal{F}_2$ constitutes the new reference point. Consequently, the resulting reference point satisfies all binary clauses of the formula $\mathcal{F}$. To compute a model for $\mathcal{F}_2$, unassigned variables are chosen iteratively. Each variable is assigned to that polarity, which is predominant for this variable among all other solving threads. Subsequently, unit propagation is applied by considering the clauses in $\mathcal{F}_2$. Note that within this process new unit clauses may be deduced or unsatisfiability may be proven.

6.2.3 CDCL threads

Most threads of SArTagnan apply conflict–driven SAT Solving with clause learning. The use of class and function templates allows for diverse configurations in each thread. If eight or more threads are available, all but one CDCL thread use activities for variables for the decision heuristic. One CDCL thread uses activity values for literals, as it was applied in the original version of Chaff [MMZ$^+$01].

All CDCL threads can be configured to apply hyper–binary resolution [Bac02a]. Binary dominators [Bie09a] are used to detect clauses for hyper–binary resolution during BCP. Most threads apply on–the–fly clause improvement [HS09]. If a clause C can be improved (i.e. its set of literals can be reduced), a new clause is created and is sent to the other threads. The message will be recommended to all users of C. Every CDCL thread applies a different restart setting. Most threads use the Luby restart strategy [LSZ93] with different initial sizes.

More details on the exact configuration of different solving threads are presented in Section 6.3. The technique described in Chapter 4 to extend Boolean constraint propagation is applied in most parallel CDCL threads. In the following, the idea is briefly recapitulated.

Extended propagation in CDCL threads

In modern SAT solvers Boolean constraint propagation is mostly equal to unit propagation. Basically, clauses that are unit (considering the current partial assignment) imply the corresponding value for the remaining variable. In Chapter 4, we analyse two approaches on how to extend unit propagation by considering clauses with more than one unassigned literal. The described approach uses the binary implication graph to detect common implications of some literals.

This idea clearly benefits from the fact that several binary clauses are exchanged among the different solving threads. Particularly, the application of hyper–binary resolution within asymmetric branching in the simplification thread produces plenty of binary clauses. The heuristic approach that either uses the pessimistic or optimistic sink–tags technique (see Section 4.2.3) is applied in almost all CDCL threads.

6.2.4 Handling incoming messages

In Section 6.1 and Section 6.2.1, different types of messages are described to send newly created clauses and notifications about simplifications. Overall, there are the following types of messages:

- Unit and binary clauses
- Shared clauses with more than two literals
- Elimination of variables
- Replacement of equal variables

Any thread checks for new messages whenever its search process is at decision level zero. However, the receive procedure may be also called at higher decision levels when more than k conflicts happened without an application of the receive procedure. For the most threads, k is equal to 256. Moreover, whenever the set of clauses has been cleaned, i.e. clauses that are marked redundant are released, incoming messages are handled subsequently. Unit and binary clauses are always imported and binary clauses are put in their own data structure. Messages with shared clauses contain a recommendation to indicate the receivers for which the clause may be interesting. If a (learnt) clause C is not recommended to a receiving thread, the receiver may still decide to import C by considering the following criteria:

- The LBD of C is smaller than $f \cdot \Lambda$, where Λ is the maximum LBD value of any learnt clause that survived the previous garbage collection of learnt clauses.
- Not more than p percent of literals (or variables) of C have an activity value that is smaller than $\frac{\Psi}{2}$, where Ψ is the current maximum activity value of all variables.

Audemard and Simon show that the LBD value of a clause may be a successful criterion to predict the quality of a learnt clause [AS09]. However, since the LBD value is related to a particular CDCL search, it does not have to be meaningful for a different search procedure. Our experiments have shown that using the second criterion, as similarly used in ManySAT [HJS09a, HJS09b], causes a more stable behaviour of the solver. Unlike ManySAT, we do not yet apply control–based adaption for the input criteria. The values f and p are static but different in each thread.

6.3 Evaluation

In this section, we evaluate our parallel solver SArTagnan with different configurations. The solver is implemented in C++ and uses the OpenMP library [Ope12] for parallelisation. The first part analyses the modification of the data structure to allow for physical clause sharing. The latter part studies different configurations and settings for the solver framework.

Sharing clauses physically

The data structure for clauses presented in Section 6.1.3 uses an additional indirection for clauses with more than two literals: A clause C contains the `XOR` value $\mathcal{X}(C)$ and a link to the shared set of literals. The cactus plot in Figure 6.1 shows the effect of the indirection to the shared set of literals on the solver's speed. To disregard the effects of parallelisation and simplification techniques, only one thread is used for this analysis. Each configuration (i.e. each curve) applies a CDCL search with equal heuristic settings. Configurations differ in the implementation of watched literals. Figure 6.1 extends the evaluation that is depicted in Figure 3.3. The tests have been run on all 614 industrial instances of the SAT competitions of 2007 and 2009 [Sat11] and the SAT–Races of 2008 and 2010 [Sat10]. As before, a point (x, y) in the cactus plot indicates that x instances could be solved when the time per instance is limited to y seconds.

The first configuration (red) implements the common watching scheme keeping the two watched literals at the front of a clause [Gel02, ES03]. The

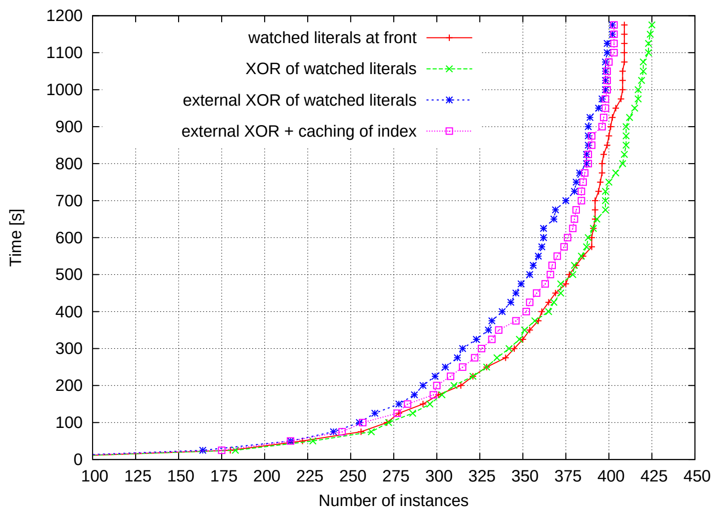


Figure 6.1: Effect of different clause implementations

second implementation (green) uses the `XOR` idea to omit both watched literals of a clause C completely and replace them by the value $\mathcal{X}(C)$, as presented in Section 3.1. Both configurations cannot be used to share clauses physically and are plotted for the sake of comparison. The third configuration (blue) wraps a clause C into a data structure that contains the value $\mathcal{X}(C)$ and a link to the set of literals, as described in Section 6.1.3. The fourth implementation (magenta) also uses this idea but extends the wrapping data structure to cache the index idx of a literal in the clause. Whenever a new watched literal has to be determined during BCP, the literals in clause C are processed in the order $[idx, \ldots, |C| - 1, 0, \ldots, idx - 1]$.

When comparing the third configuration to the first and second, the drawback of using another indirection to access clauses is clearly observable. However, the fourth configuration shows that the caching of one literal's index considerably improves the performance. Nevertheless, the drawback of using another indirection to access clauses cannot be compensated completely, since the magenta curve is consistently left of the red curve, which represents the common implementation of watched literals. For this reason, physical clause sharing in parallel SAT solving has to compensate for this drawback by taking advantage of global simplification.

Different solver configurations

We evaluated various solver configurations using eight parallel threads. To allow for a fair comparison using real concurrency, each run of a solver has to request eight cores per cluster node [BWG12]. Since this often increases the queue time of submitted jobs considerably, we reduced the set of benchmarks to the 100 instances of the SAT–Race 2008 [Sat10]. Each solver was allowed to run for a maximum of 1200 seconds per instance.

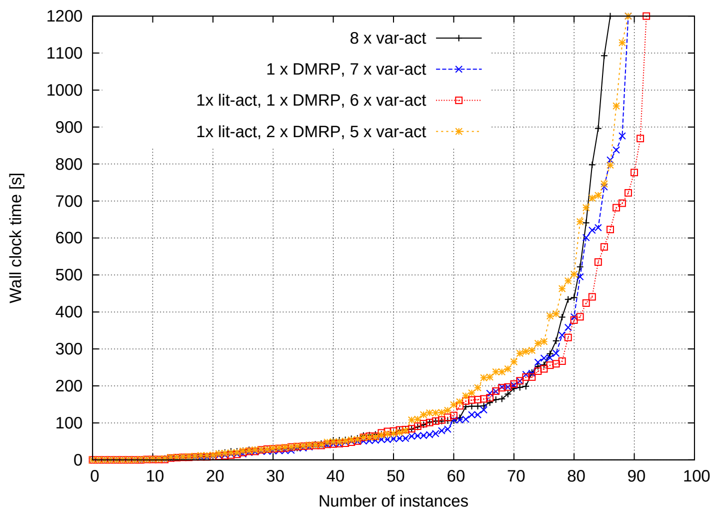


Figure 6.2: Effect of heterogeneous solver threads

Figure 6.2 compares four different solver configurations in a cactus plot and depicts an interesting issue regarding the heterogeneity of different threads. The first solver configuration (black) applies the VSIDS decision heuristic for variables in each parallel thread (cf. Section 2.2.5). However, the settings for each parallel CDCL thread still differ. The second configuration (blue) replaces one of the eight CDCL threads by one DMRP thread, using the heuristic described in Section 6.2.2.

The improvement is clearly observable in the plot. The next solver configuration (red) changes the VSIDS heuristic in one CDCL thread to use activities for literals instead of variables. This modification causes another clearly observable improvement. Note that the use of activity values for literals was originally proposed for the VSIDS heuristic [MMZ^{+}01]. Since the improvements made by MiniSat, activity values are now used for variables instead [ES03].

	threads	1	2	3	4	5	6	7	8
	main task	SIMP	CDCL	CDCL	DMRP	CDCL	CDCL	CDCL	CDCL
decisions	act. for:	vars	vars	vars	/	vars	lits	vars	vars
	decay	1.05	1.05	1.05		1.09	1.07	1.09	1.04
	pol.	phase	phase	phase		phase	/	false	phase
	rand.	0.2	0.15	0.2	0.15	0.15	0.15	0.15	0.2
	BUP	opt.	pes.	opt.	⊘	opt.	opt.	⊘	pes.
restart	type	static	Luby	Luby	Luby	Luby	geo.	geo.	geo.
	init	100	32	64	100	16	100	100	100
	inc.	0	Luby	Luby	Luby	Luby	1.5	1.5	1.3
import	LBD	3/2	3/2	3/2	3/2	5/4	3/2	3/2	7/6
	act	90	70	80	70	70	80	70	80
	hyp. bin.	√	√	√	⊘	√	√	√	√
	cls improve	√	√	√	⊘	√	√	√	√

Table 6.3: Configuration for the SAT–Race 2010 using eight threads

Overall, the third configuration performed best on the selected benchmark set. On average, it solved 92 of 100 instances for repeated runs within a time limit (wall clock time) of 1200 seconds. Interestingly, the result of the parallel solver is rarely returned by the DMRP thread or by the CDCL thread, which uses the activity of literals (see also Figure 6.5). However, the benefit of using these thread configurations is clearly visible in Figure 6.2. The fourth configuration (orange) replaces one CDCL thread of the third configuration by another DMRP solving thread. The solver performance with this setting clearly declines.

Table 6.3 lists the details for the best performing solver configuration, as chosen for the SAT–Race 2010 [Kot10b, KK11c]. The first four lines indicate the settings for the VSIDS heuristic (where it makes sense) and the percentage of random decisions. The decaying factor (second line) is slightly varied in some threads. Most threads apply phase saving ([PD07a, PD07b]) to choose the polarity (third row) for decision variables. One CDCL thread always assigns the value *false* to decision variables, as was done in the first version of MiniSat [ES03].

The fifth row indicates the application of extended propagation, as explained in Chapter 4. Six of all eight solving threads apply either the pessimistic or the optimistic heuristic. In particular, the concept is not applied within DMRP to avoid linking the generated clauses into the data structure (cf. Section 5.2.4). The next three rows list the restart configurations. As the main task of the simplification thread is not solving, a restart is performed every 100th conflict. For this thread, each restart also triggers the extensive application of simplification techniques. Four threads apply the Luby restart strategy [LSZ93] with a different initial frequency (see also Section 5.3). The three remaining threads use geometric restarts, where the number of maximal conflicts for one search process is multiplied by the given value (increment).

As described above, a learnt clause is imported by a solving thread if the corresponding message is recommended to the thread. Moreover, a clause may still be imported based on the estimation of its quality. The subsequent two rows state the threshold for the two criteria explained in Section 6.2.4. As pointed out in Section 6.2.1, the simplification thread imports most clauses to detect duplicates and subsumptions, and to apply several other simplification techniques. On the other hand, the DMRP solving thread imports fewer clauses due to its increased effort on the maintenance of clauses. Besides the DMRP thread all other solving threads apply lazy hyper–binary resolution [Bie09a] (cf. Section 3.3.1) and on–the–fly clause improvement [HS09]. By using the latter technique, the possibility to remove some literals from a shared clause may be detected during conflict analysis.

↓ average ↓		1	2	3	4	5	6	7	8
		SIMP	CDCL	CDCL	DMRP	CDCL	CDCL	CDCL	CDCL
	restarts	44	1501	630	214	2416	14	15	21
	props [10^6]	24.1	143.5	100.3	74.3	115.8	96	145.3	156.3
	jumps [10^6]	0.4	1.9	1.2	1.4	1.7	1.1	1.3	1.5
	decisions [10^3]	48	1893	1203	3115	1747	1055	1322	1505
	conflicts [10^3]	4.3	254.5	184.8	100.6	220.6	229.8	292.4	298.5
	max level	373	1852	1723	551	1604	726	1823	1787
	prop/conf [10^3]	48.1	3.6	3.2	3.6	3.1	3.2	2.3	2.9
	jump/conf	225	41	35	10	23	9	18	16
BUP	short [10^3]	23.2	0.4	4.9		6.7	13.5		0.5
	long [10^3]	0.2	29.8	83.9	/	93.3	69.6	/	25.3
	selfsub	6	473	534		565	695		670
	hyper–bins [10^3]	221.8	21.5	16.7	42.2	22.1	32.6	36.2	20.7

Table 6.4: Statistics for the best solver configuration

In Table 6.4, some statistics of the previously described solver configuration are listed. As in Table 6.3, each thread is represented by one column. A value in the table represents the average over the results of all solved instances. If stated in the first column, the values in one row may have to be multiplied by 10^3 or 10^6. When comparing the values of different threads, one has to bear in mind that the first thread spends most of its time on simplification, not on searching.

The first six rows give some information on common search properties. The average number of restarts, shown in the first row, follows directly from the restart policy. The highest number of propagations is reached by the two CDCL threads that use variable activities and geometric restarts. The third row indicates the average value of the total number of decision levels that are undone within search (backjumps). The application of frequent restarts (threads 2 and 5) causes the highest number of jumps. The comparatively high number of decisions for the DMRP solving thread is due to the complete assignments that are computed at each restart to initialise a new reference point (cf. Section 6.2.2). It is remarkable that the CDCL solving thread that uses activity values for literals (thread 6) produces a small number of decisions.

A big difference is observable for the number of generated conflict clauses. The rightmost thread produces almost three times more conflicts than the DMRP solving thread. On the other hand, the fourth and sixth threads detect conflicts at clearly lower decision levels. On average, their maximal decision level is significantly smaller than the maximal decision levels that are reached by the other solving threads (disregarding the simplification thread).

In the next two rows, the number of propagations and jumps is related to the number of conflicts. Again, it is remarkable that the two threads using DMRP and activities for literals exhibit a comparatively small number of decision levels that are undone by each conflict.

The application of extended propagation (BUP) generates additional clauses, as explained in Section 4.2.4. The table distinguishes between generated short clauses with two literals at most and longer clauses. The average number of generated clauses is particularly high when the optimistic heuristic is applied (cf. Section 4.2.3). On the other hand, the number of generated clauses that subsume the triggering clause is rather negligible. Extended propagation clearly benefits from the application of lazy hyper–binary resolution. As listed in the last row, each thread produces several binary clauses that are, in turn, shared among all threads.

A high degree of sharing

Sharing a lot of information among several solving threads may be beneficial in terms of the simplification of an instance. It can be still more useful to consider more information for heuristics, as is done by the DMRP solving thread. On the other hand, the interchange of information exhibits a major drawback when it comes to deterministic solving. As we already pointed out in the first two chapters of this thesis, small modifications of a solver may have unexpectedly big side–effects [AS08]. This issue makes it difficult to evaluate a modification of a solver or an entirely new approach. For the parallel case, things become even more complicated, the higher the interchange of information is. Hamadi *et al.* presented the first deterministic parallel SAT solver [HJPS11]. However, this solver does not share information physically. Copies of clauses may be interchanged in designated synchronised blocks.

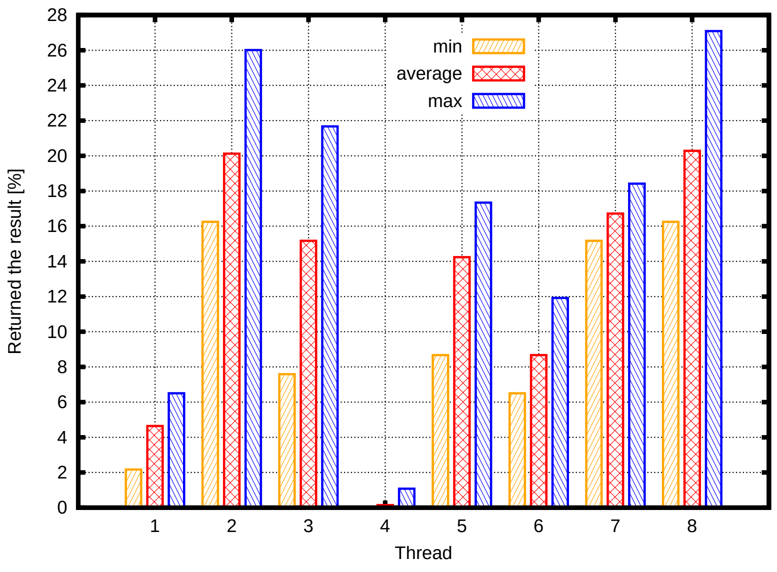


Figure 6.5: Threads returning the result

Figure 6.5 summarises the results of seven solver runs with nearly identical settings on all 100 instances of the SAT–Race 2008. For each run on the 100 instances, we stored the percentage of instances for which a particular thread was fastest. The red bar illustrates the average value over all seven runs. The minimal (maximal) percentage is indicated by the yellow (blue) bar.

The plot shows that neither the simplification thread nor the DMRP thread returns the result very often. Furthermore, the sixth thread, which uses activity of literals, exhibits smaller values than the remaining threads. However, this does not necessarily reflect their contribution to the solving of an instance (cf. Figure 6.2). An interesting issue regarding Figure 6.5 is the difference between the minimal and the maximal values. For the third thread in particular, there was one run where it solved 21.6% of all instances. In another run, it only returned the result for 7.6% of the same instances. This underlines the nondeterministic behaviour of the solver.

# solved	plingeling	ManySAT 1.5	ManySAT 1.1	SArTagnan	antom
in 1st run	78	75	72	70	67
SAT/UNSAT	23/55	19/56	18/54	18/52	19/48
⌀ seconds	97.7	143.9	124.0	86.5	83.1
in 2nd run	79	74	73	70	65
in 3rd run	79	74	71	72	68
in at least one run	80	78	76	76	69

Table 6.6: Results of parallel solvers that qualified for the second round in the SAT–Race 2010 [Sat10]

Table 6.6 shows the results of the parallel track in the SAT–Race 2010. The table is available at [Sat10] and lists all parallel solvers that qualified for the second round. Due to the nondeterministic behaviour of parallel SAT solvers, all tests in the parallel track are performed three times. The table indicates the results for all three runs. However, only the first run was evaluated in the 2010 race.

In particular, the last row of the table is interesting. It states the number of instances that are solved in at least one of the three runs. For ManySAT and SArTagnan, the difference of this value to its best run is bigger than for the other solvers. Besides the difference between the best and the worst run, this can be considered as another indication for a high degree of nondeterminism.

Regarding the parallelisation of SAT solving, it is remarkable that the most successful solvers share the least information among different threads. In Table 6.6, plingeling is clearly the best solver, whereas it only interchanges unit clauses among different threads. This may be a direct consequence of information sharing, being it physical or logical. Another reason may be the possibility of optimising such a parallel solver in a better way. The solving strategy in each thread can be optimised independently and deterministically to broaden the total set of solved instances. The success of this strategy was impressively shown by the parallel solver ppfolio in the SAT competition 2011 [Sat11]. Five different SAT solvers are compiled to one parallel solver without any exchange of information between the solvers [Rou11]. In the competition, ppfolio beat several other parallel solvers. This result may set a trend for further parallelisation of SAT.

6.4 Summary

In Chapters 3 to 5, different SAT solving approaches and techniques have been presented and evaluated. In this chapter, we incorporated many of these techniques into a parallel SAT solver. We distinguished two aspects of parallel SAT solving. The first aspect is related to the physical sharing of clauses in the solver. The second aspect is concerned with heterogeneous SAT solving by using different solving techniques.

We first presented a design and implementation that allows for physical clause sharing in parallel SAT solving. This is contrary to most other state–of–the–art parallel solvers that only exchange copies of some clauses among different solving threads. Physical clause sharing and the communication between threads is used to let all threads benefit from the application of simplification and clause minimisation techniques in any other thread. Thereby, all communication and sharing of data is realised without the use of operating system locks. One drawback of physical clause sharing is the additional indirection to access the set of literals of a clause. This issue was studied in Section 6.3.

In the second part, we discussed the incorporation of different solving techniques in more detail and referred to the relevant work within the preceding chapters. We explained and evaluated different configurations of our parallel solver SArTagnan. Moreover, we examined the final configuration of the solver that participated in the SAT–Race 2010. SArTagnan could compete with state–of–the–art parallel SAT solvers and was awarded the best student solver in the parallel track.

Chapter 7

Related Projects

Apart from the work presented in the previous chapters, there are some projects that are slightly beside the common thread of the thesis. Therefore, this chapter briefly describes some projects[1] that are related to SAT in different ways.

Understanding and analysing the success and behaviour of SAT solving techniques is still an open challenge in SAT research. In Section 7.1, we present the tool CoPAn that allows for a deep analysis of several aspects related to learning within CDCL SAT solvers. On the Computer Science Day in Tübingen, the *Algorithmik* group[2] demonstrated an interactive tool to embed planar graphs onto grids. For some graphs even the presenters could not agree on the embeddability of the graph. Section 7.2 presents the work that was subsequently launched to prove facts.

SAT solving has established itself in many different areas often related to verification or configuration problems. The *Symbolisches Rechnen* group[2] has a long lasting cooperation with the automotive industry to verify product configuration. Section 7.3 summarises some results of our collaboration. The introduction of the concept of backdoors for SAT instances opened new and interesting perspectives for theoretical and practical research related to SAT. Section 7.4 sketches two approaches in the context of backdoors.

In 2011 the SAT competition provided two new tracks on the minimisation of unsatisfiable subsets (MUS). Previous achievements [Kot09, Kot10b] motivated the submission of a MUS solver that finished third in the clause based MUS computation. Section 7.5 gives an overview of the main aspects of the implementation.

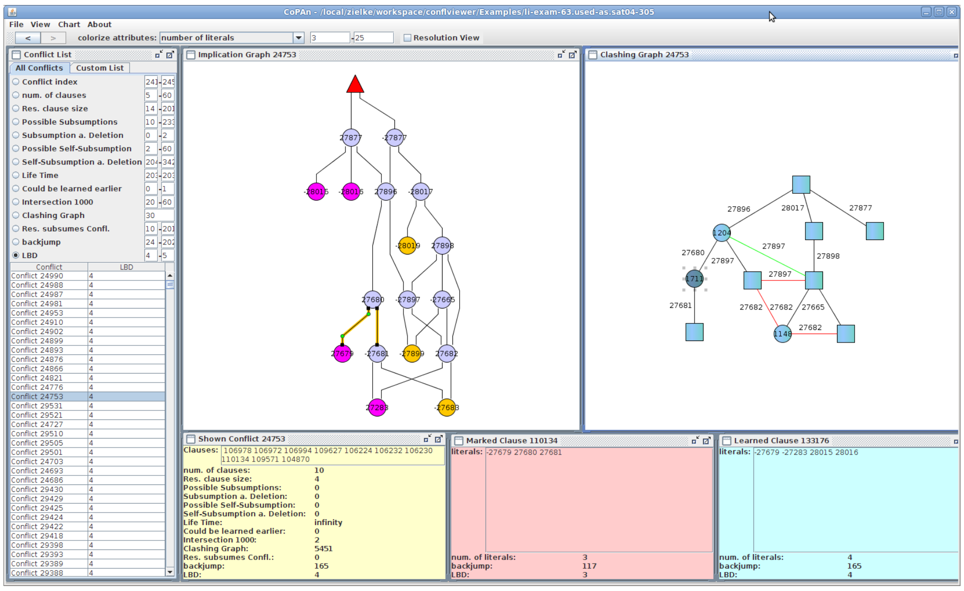
[1] The projects in Sect. 7.1,7.3 and 7.4 were part of the StrAlEnSATs proposal [KK13]

[2] Wilhelm–Schickard–Institut, Universität Tübingen

7.1 Conflict Pattern Analysis

Though the vast success of the CDCL approach to SAT solving [MSS99, ZMMM01, ES03] is well documented, it is incomprehensible why small changes in the choice of parameters may cause significantly different behaviour of the solver. With the tool CoPAn (**Co**nflict **P**attern **An**alysis) we provide a perspective to find an answer for the question about subtle differences between successful and rather poor solver runs [KZSK12, Sei10]. CoPAn allows for an in–depth analysis of conflicts and the associated process of producing learnt clauses. The analysis requires common proof logging output of the systems and uses efficient external data structures to cope with a big amount of logged data.

Taking a closer look at the inside of learning within CDCL is motivated by the high contribution of learning to a solver's efficiency [KSMS11]. On one hand, new measures are proposed to estimate the quality of learnt clauses based on the observation of CDCL solvers on industrial instances [AS09]. On the other hand, it is very promising to turn away from these static measures that cause a definitive elimination of clauses and focus on a dynamic handling of learnt clauses [ALMS11]. Therefore, changes within conflict analysis can lead to a considerable speedup of the solving process (see [SB09, AS09, HJS10]).



colorize attributes: number of literals | 3 - 25

○ Conflict index	241	245
○ num. of clauses	5	60
○ Res. clause size	14	201
○ Possible Subsumptions	10	233
○ Subsumption a. Deletion	0	2
○ Possible Self-Subsumption	2	60
○ Self-Subsumption a. Deletion	204	342
○ Life Time	203	203
○ Could be learned earlier	0	1
○ Intersection 1000	20	60
○ Clashing Graph	30	
○ Res. subsumes Confl.	10	201
○ backjump	24	202
● LBD	4	5

Conflict	LBD
Conflict 24990	4
Conflict 24988	4
Conflict 24987	4
Conflict 24981	4
Conflict 24953	4
Conflict 24910	4
Conflict 24902	4
Conflict 24899	4
Conflict 24893	4
Conflict 24876	4
Conflict 24866	4
Conflict 24821	4
Conflict 24776	4
Conflict 24753	4
Conflict 29531	4
Conflict 29521	4
Conflict 24727	4
Conflict 29510	4
Conflict 29505	4
Conflict 29501	4
Conflict 24703	4
Conflict 24693	4
Conflict 24686	4
Conflict 29430	4
Conflict 29429	4
Conflict 29425	4
Conflict 29424	4
Conflict 29422	4
Conflict 29418	4
Conflict 29398	4
Conflict 29393	4
Conflict 29389	4
Conflict 29388	4

Shown Conflict 24753

Clauses: 106978 106972 106994 109627 106224 106232 106230 110134 109571 104870

num. of clauses:	10
Res. clause size:	4
Possible Subsumptions:	0
Subsumption a. Deletion:	0
Possible Self-Subsumption:	0
Self-Subsumption a. Deletion:	0
Life Time:	infinity
Could be learned earlier:	0
Intersection 1000:	2
Clashing Graph:	5451
Res. subsumes Confl.:	0
backjump:	165
LBD:	4

Marked Clause 110134

literals: -27679 27680 27681

num. of literals:	3
backjump:	117
LBD:	3

Learned Clause 133176

literals: -27679 -27283 28015 28016

num. of literals:	4
backjump:	165
LBD:	4

Figure 7.1: Main view of the CoPAn GUI

Patterns in clause learning

In contrast to any generic learning scheme we focus on different resolution trees that are applied to learn new clauses. At first CoPAn allows for visualising the resolution trees that were logged by a CDCL solver (Figure 7.1). The main focus, however, is put on the patterns that can be observed in the resolution trees of different conflicts rather than pure visualisation. Unlike the implication graph, as it is described by Marques-Silva *et al.* [MSS99, SB09], we consider a resolution graph (tree) that contains one node for each clause contributing to the conflict. For each resolution operation an edge is drawn. In CoPAn, we consider the resolution graph with additional edges, a so–called clashing graph. An edge between two nodes (clauses) is drawn iff they share exactly one clashing literal (see [Kul04]). The decision to focus on clashing graphs rather than on common implication graphs is based on its direct relation to resolution. Isomorphic subpatterns in different clashing graphs are likely to allow for similar resolution operations.

Features of CoPAn

One of the most advanced features in CoPAn is the search and filter functionality that goes along with resolution patterns. Any resolution pattern that is shown in the main window can be used to filter all conflicts of a solver's run. This allows a user to search for conflicts that exhibit similar resolution patterns even if the pattern under consideration only constitutes a subgraph of another learning operation.

A typical task could be to search for significantly common properties among conflicts with isomorphic resolution pattern. Properties of clauses may be attributes, such as backjumping distance [MSS99], activity [MMZ$^+$01, ES03] and LBD value [AS09] but also user–defined properties. The most important features of CoPAn can be summarised as follows:

- Search for isomorphic resolution patterns - within one instance or within a selection of several instances
- Subsumption checks of (subsets of) clauses
- Specification of user–defined properties that can be linked to clauses
- GUI to visualise conflicts, to explore the effect of various filters and to trace learning and proofs interactively
- Preprocessing of logged data to build efficient data structures and indices for further examinations
- Batch processing to analyse sets of instances in unattended mode

7.2 Using SAT for a Graph Problem – GridFit

There are many problems in graph theory that allow for a translation into SAT [Hoo98, Vel07, GSM10]. It is also a common approach to generate so–called crafted instances for benchmarking. In [KK10], we present a SAT encoding to tackle the following graph problem for small graphs. Especially for research purposes it may be useful to certainly answer this question for a particular setting.

For a given planar graph $G = (V, E)$ (as in Figure 7.2) and a rectangle of size $h \times w$ we ask, whether there exists a planar straight–line embedding of G onto the grid–points of the rectangle. For this NP–hard problem [KW07] some powerful heuristics have been developed to minimise the area of an embedding of a given graph [KW07, Kru07]. Moreover, for particular families of graphs upper and lower bounds on the area have been proven [FP08]. However, in the general case it is not possible to ensure whether there exists an embedding onto a rectangle that preserves a particular area restriction.

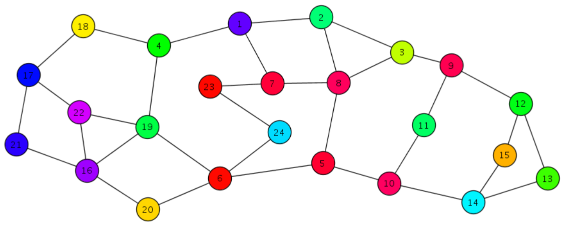


Figure 7.2: The depicted graph contains 24 vertices and is planar, i.e. it can be embedded without edge crossings. As on the Computer Science Day in Tübingen the question may arise whether it is possible to find an embedding for this graph onto a rectangle with 6×4 grid points.

The planar embeddability of a graph onto a given rectangle can be formulated as a SAT problem. The output of a solver for such a SAT encoding can be translated back straightforwardly. If the solver proves unsatisfiability, the existence of an area preserving embedding is disproved. If, on the other hand, the solver proves satisfiability, the computed model can be translated back into a valid embedding. The tool *GridFit* takes a graph and a grid specification as input and invokes a particularly tuned SAT solver for the transformed problem. If existing, it returns a valid embedding.

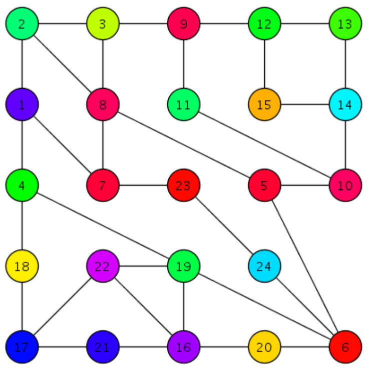


Figure 7.3: Embedding the graph onto a 4 x 6 rectangle is impossible and can be disproved by the solver. However, for a 5 x 5 rectangle the depicted solution is computed.

SAT encoding and solver modification

Choosing a proper encoding may often be crucial for the feasibility to solve the underlying problem. In [KK10], we present one encoding that is basically composed of the following two parts:

(i) Vertices have to be matched to grid positions.

(ii) A computed embedding has to be planar.

For a proper matching of vertices to grid positions it has to be ensured that any vertex of the graph is placed on at least one grid position and secondly any grid position holds at most one vertex. To make part (i) more efficient, we modified our SAT solver to additionally support cardinality constraints. To enforce a planar embedding, additional variables are used that represent the existence of straight–line connections between any two grid positions. Constraints are added to prohibit the coexistence of any two crossing straight–lines. This also reduces the number of symmetric constraints.

Experiments have shown, that the SAT approach has difficulties to deal with larger graphs in general. However, significant speedups can often be achieved for special classes of graphs. Particularly interesting in the context of minimal grid embeddings are the results for nested triangle graphs (see [FP08]).

7.3 Application for Automotive Industry

The application of SAT technology has proven its usefulness for industrial applications. Many verification and configuration applications use a SAT solver as back–end technology [Vel02, Vel04, BP08, MS08, ABL^{+}10, VG11]. The cooperation with the car manufacturer Daimler [SKK03] is one of the pillars of the project StrAlEnSATs [KK13]. Theoretical approaches and implementations for the formal verification of the constructability of cars can be evaluated in real practice.

Boolean logic meets practice

Comprehensive work has been done [SKK03] to allow for the direct utilisation of formal methods to the applications in the field of automotive product configuration. To this end, all constraints regarding the configuration of an automobile are expressed as a single Boolean formula, a so–called POF[3]. Every solution of the POF corresponds to one possible car configuration.
By using the formal approach, conclusions can be drawn about cars that are not required to have ever been constructed before. In general, all cars that can be configured and constructed according to the POF may be considered and analysed. For example, the following questions about the bill of materials can be modelled and solved as satisfiability problems:

- Are the constraints for a particular part of the bill of materials compatible with the POF?
- Are there any automobiles for which several alternative parts are ordered at the same time?
- Are there any automobiles where parts may be lacking?

The solver that was used in practice until 2008 implemented all satisfiability checks without a transformation into CNF [Kai03]. However, the increase of complexity of the resulting SAT problems caused the solver to exceed the timeout of 30 seconds for some instances. Based on our SAT solver [Kot09] we implemented a more efficient solver engine tailored to the particular needs of the application at Daimler. The main features of the plain CNF solver can be summarised as follows:

- In difference to the non–CNF solver, the input formula is transformed into CNF. The Tseitin transformation [Tse68] is modified to limit the total number of additional variables. As long as the resulting CNF encoding does not exceed a given size, expansion is applied for the translation into CNF.

[3]Product Overview Formula

- Due to the kind of queries of the practical application two resulting formulae F_0 and F_1 may often have the following properties: $F_1 = F_0 \wedge F_2$, whereas F_2 contains only a few additional constraints. The transformation to CNF can incorporate this fact and generate formulae F_0^* and F_1^* that only differ in a small set of clauses. As a consequence, after solving F_0^*, the set of learnt clauses and heuristic information (e.g. activity values of variables) are kept by the solver. This knowledge can be reused to solve F_1^*. Consequently, learning is realised beyond the scope of one single instance.

Evaluation

The modified solver engine was evaluated in a joint work with Matthias Sauter. Figure 7.4 compares the runtime of the two solving approaches (CNF and non–CNF) for different series of cars. The higher the complexity of a formula the less significant are the costs for the transformation into CNF.

The queries for the Daimler series C906 induce 162,000 SAT instances that have to be tested. While the non–CNF solver required 39,678 seconds for the overall computation ($\varnothing$: 0.25 seconds per instance), our modified solver reduces the time remarkably by more than 90 percent. All computations are completed after 3,078 seconds ($\varnothing$: 0.02 seconds per instance).

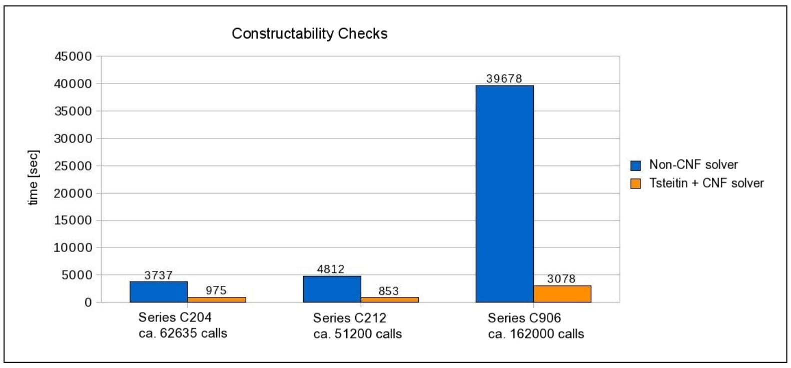


Figure 7.4: Runtime comparison of the two solvers for complete constructability checks of different series of cars.

7.4 Structure in SAT Instances – Backdoors

There is a strong belief in different SAT communities that real–world SAT instances are structured in some way. Several properties can be specified in which randomly generated instances differ from industrial instances [NLBD$^+$04, XHLB07]. This fact encourages the development of domain specific solvers, as the results of the SAT competitions clearly show [Sat11]. Based on structural information portfolio SAT solvers may then choose a proper solver for a particular domain [XHHLB08]. Despite the recent success of portfolio solvers it is still an open and challenging task to predict the hardness of an instance in advance. The related question of estimating the progress during solving is still open, although many advances have been published recently [HW08].

The concept of backdoors was introduced in 2003 to measure and categorise the hardness of SAT instances [WGS03a, WGS03b]. Basically, a subset of variables $B \subseteq \mathcal{V}$ of a formula F is a backdoor to solve F if the satisfiability of F can be decided within a time–bound that is only exponential in B. In the seminal paper [WGS03a], the authors give examples of extremely small backdoors for industrial SAT instances. Hence, the idea was taken up by different SAT communities and enhanced in many directions [Int03, Sze05, SS07, DGS07, DGS09]. In [Kot07], we explore and analyse different aspects related to backdoors. Further improvements and advances are published in [KKS08a, KKS08b].

Backdoors in theory

In [KKS08a], we improve a theoretical bound for an NP–hard subclass of 3–SAT, first presented in [Kot07]. The considered class $2^\star$-CNF is defined as a subclass of 3–SAT with the restriction that any clause C with $|C| = 3$ must only contain negative literals.

To decide whether an instance of class $2^\star$-CNF is satisfiable is NP–complete [Kot07]. Algorithm 7.1 shows a procedure to solve instances of this class by using two different kinds of backdoors. Unlike the common application of backdoors, this algorithm also uses the absence of backdoors of a particular size. The complexity of the algorithm is bounded by $O(1.427^n \cdot p(n))$ where $p(n)$ is polynomial. This bound is slightly better than the bound $O(1.4423^n)$ to solve the more general class of mixed Horn formulae (MHF) [PS07]. At the time of publication, the fastest deterministic algorithm to solve 3–SAT was bounded by $O(1.473^n)$ [BK04], meanwhile it has been improved to $O(1.439^n)$ [KS10].

Algorithm 7.1: A backdoor–driven 2*-CNF solver

```
Function bdSolve ($F$)
    $c \leftarrow \log_{4.151}(2.0755) \approx 0.513$
    $C^+ \leftarrow \{(x_i \vee x_j) \in F : \ x_i, x_j$ positive$\}$
    Choose minimum $\mathcal{B}^+ \subseteq \mathcal{V}$, such that $\forall\ C \in C^+\ \exists\ x_i \in \mathcal{B}^+ : x_i \in C$
    if $|\mathcal{B}^+| \leq c \cdot |\mathcal{V}|$ then
        return Solve $F$ by using the Horn–backdoor $\mathcal{B}^+$
    $C^- \leftarrow \{(\overline{x_h} \vee \overline{x_i} \vee \overline{x_j}) \in F : \ \overline{x_h}, \overline{x_i}, \overline{x_j}$ negative$\}$
    Choose minimum $\mathcal{B}^- \subseteq \mathcal{V}$, such that $\forall\ C \in C^-\ \exists\ x_i \in \mathcal{B}^- : \overline{x_i} \in C$
    if $|\mathcal{B}^-| \leq (1-c) \cdot |\mathcal{V}|$ then
        return Solve $F$ by using the binary–backdoor $\mathcal{B}^-$
    return $F$ Unsatisfiable
```

Given a formula F of class $2^\star$-CNF, the algorithm focuses on the set of binary clauses C^+ that contain positive literals only. If a Vertex Cover $\mathcal{B}^+$ of at most $c \cdot |\mathcal{V}|$ variables can be found such that all clauses in C^+ are covered, then $\mathcal{B}^+$ constitutes a Horn–backdoor. In that case, F can be solved by checking all $2^{|\mathcal{B}^+|}$ assignments for the variables in $\mathcal{B}^+$. For any such assignment the remaining formula will be a polynomially decidable Horn formula. If no such backdoor exists, the algorithm checks analogously if there is a 3–Hitting Set $\mathcal{B}^-$ with at most $(1-c) \cdot |\mathcal{V}|$ variables to cover all ternary clauses with only negative literals C^-. If existent, $\mathcal{B}^-$ constitutes a binary–backdoor. If none of these two size restricted backdoors exist, then F has to be unsatisfiable since C^+ and C^- cannot be satisfied simultaneously. With a proper choice of constant c and the application of FPT algorithms [Nie02, Wah07] for the Vertex Cover and the 3–Hitting Set computation, the worst case upper bound can be guaranteed.

Backdoors in practice

The work presented in [KKS08b], deals with backdoors with regard to Renameable Horn subformulae. The requirements to rename the variables of a given formula F to make it a Horn formula are modelled as a directed graph. This graph constitutes the binary implication graph of Aspvall *et al.* [APT79] of the constraints to decide whether a formula is Renameable Horn [Lew78]. Based on the so–called dependency graph we present two algorithms that remove a possibly small number of variables B to make the graph acyclic. A result of these algorithms gives a renaming of the variables $\mathcal{V} \setminus B$. Furthermore, B constitutes a Horn–backdoor for the renamed formula. The algorithms allow for the computation of small backdoors for some industrial SAT instances within reasonable time. Unlike the first results by Williams *et al.* [WGS03a], the presented algorithms do not require a complete solution of the given formula.

7.5 Minimal Unsatisfiable Subsets

The transformation of a real–world problem into SAT can often be a convenient way to solve the problem. However, due to the nature of SAT the result of a standard solver will at first be Boolean. For a satisfiable instance, a found model may be reasonably retransformed into the original problem. On the other hand, if a solver detects unsatisfiability, this may only indicate the existence of a failure in the original application. If so, it may be indispensable to narrow down the set of clauses to an unsatisfiable core to possibly eliminate a failure in the input formula.

Given an unsatisfiable instance, a MUS solver computes a minimal unsatisfiable subset of clauses (MUS, or MUC for Minimal Unsatisfiable Core) that is still unsatisfiable. Hence the removal of any clause of a MUS results in a satisfiable instance. Minimal sets can also be defined on a higher level where each clause belongs to a group or category of clauses that represent a logical unit (e.g. a hardware module) of the original problem. Minimality is then related to groups of clauses.

A basic algorithm

Surprisingly, a very basic approach to minimise unsatisfiable cores turns out to exhibit the currently best performance. The idea is formally described by Nadel [Nad10]. We only sketch the procedure for clause–based MUSes here. An unsatisfiable set of clauses $\mathcal{U}$ is minimised by iteratively testing each clause C_i of $\mathcal{U}$. The initially empty set $\mathcal{M}$ holds the clauses that are definitely known to be in the minimal set. If $\mathcal{T}_i = \mathcal{M} \cup \mathcal{U} \setminus C_i$ is satisfiable, then C_i is required for unsatisfiability and will thus be put into $\mathcal{M}$. If, on the other hand, $\mathcal{T}_i$ is proven to be unsatisfiable, then the empty clause was eventually learnt by the solver. Analysing the proof trace of the solver allows for removing all clauses from $\mathcal{U}$ that did not contribute to learning the empty clause. Obviously, at least C_i can be removed from $\mathcal{U}$. As soon as $\mathcal{U}$ gets empty, $\mathcal{M}$ constitutes a MUS.

The minimisation is based on the modification of clauses and the use of so–called selector variables. Each non–unit clause $C_i = \{l_x \vee \ldots \vee l_y\}$ of the original formula is extended by a fresh variable such that $C_i' = \{s_i \vee l_x \vee \ldots \vee l_y\}$. By default, all selector variables are assumed (i.e. decided at level 0) to be $false$ so that we have $C_i = C_i'$ for all clauses. When analysing the proof trace after an unsatisfiable result, the selector variables allow for the identification of assumptions that contributed to the proven result. Secondly, the clause set $\mathcal{T}_i$ can be built easily by temporarily setting the selector variable s_i to $true$.

Tuning the MUS computation

In order to prove the minimality of an unsatisfiable core the approach described above requires a SAT solver to solve at least $|\mathcal{M}|$ different formulae. Many of these formulae are easy and can be solved quickly. The success of MoUsSaka [Kot11] in the MUS competition 2011 [Sat11] is rather based on its data structure than on extraordinary MUS heuristics.

Clause extension

In addition to the `XOR`–watchers scheme described in Section 3.1, the implementation of clauses is adapted to the demands for proof tracing and MUS computations. Each non–unit clause contains at least one variable that is $false$ at level zero — most of the time. Learnt clauses may contain several such literals. This motivates the partition of clauses into fixed literals that are only used for conflict analysis and other literals.

The implementation shown in Figure 7.5 extends the one of Figure 3.1 by using an extra bit to mark the end of unfixed literals within a clause. Unit propagation only needs to check literals up to this partition mark. Frequently, after k calls of the SAT solver the next k selector variables are scheduled. This requires to abolish all partition marks within clauses where a scheduled selector would be invisible for BCP.

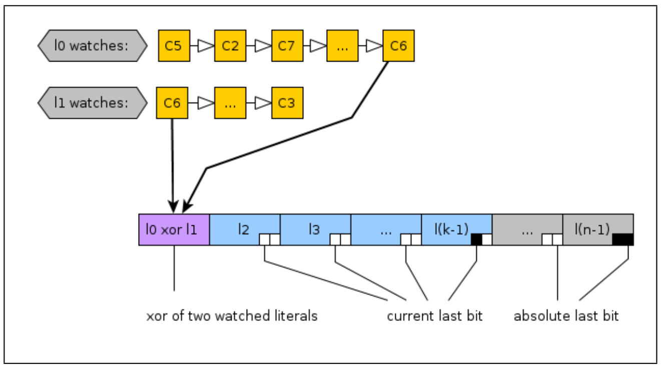


Figure 7.5: Partitioning of literals (extension to Figure 3.1)

Cleaning and simplification

After some tests of selector variables there are many variables (at least the tested selectors) whose state is finally fixed and will not be changed in later minimisation iterations. Using this fact simplification can make the formula much smaller and often easier for later calls of the solver. Furthermore, experiments have shown that it is eminently valuable to apply pure literal elimination, i.e. the value of variables that only occur in at most one polarity can be fixed. Since a CDCL–based SAT solver has to assign all variables to detect satisfiability for a formula, even trivially satisfiable inputs $\mathcal{T}_i$ require the assignment and unassignment of all unfixed variables. Thus fixing as many variables as possible is worthwhile. Frequent simplifications of the formula can be used to rearrange the partition marks of clauses.

Variable activity

The commonly applied VSIDS heuristic [MMZ$^+$01] uses activity scores for unfixed variables [ES03]. During solving variables with high scores are chosen for decisions. Analogously, we increment the activity value of a fixed variable whenever it is involved in a conflict. In contrast to VSIDS, a low activity score is preferred when minimisation chooses a selector variable from $\mathcal{U}$ for the next clause set $\mathcal{T}_i$. This is motivated by the fact that a selector variable s_i with low activity shows a minor contribution to recent conflicts. Thus, the actual set of learnt clauses may be useful to prove unsatisfiability for the set $\mathcal{T}_i$ (where s_i is irrelevant) with little effort.

Quick tests

For many temporary clause sets $\mathcal{T}_i$, the solver requires only a few conflicts to return a result. If a selector variable s_i is tested for the first time and too many conflicts arise during solving, the solver quits and the test of s_i is postponed until quick tests have been conducted for all selector variables. It may also happen that the complete (i.e. the second) test is never performed. This applies if s_i is determined to be irrelevant by the proof analysis after an unsatisfiability result for another set $\mathcal{T}_j$.

Chapter 8

Conclusion

In this thesis, we studied different methods and techniques to solve instances of the Satisfiability problem. We particularly focused on SAT instances that encode real–world applications. For this kind of instances, conflict–driven solving constitutes the method of choice and its implementations have been highly optimised within the last 15 years. The motivation of this work was to elaborate on rather uncommon methods that go beyond predominant solving techniques.

Results of the thesis

In Chapter 3, we suggested an improvement of the implementation of the two watched literals scheme that is applied in most state–of–the–art SAT solvers. The idea is based on the observation that a clause is solely accessed by one of its two watched literals during the search. This allows for an `XOR` compression of the two watched literals, which, in turn reduces the resource requirements of the solver. In the remainder of this chapter, we analysed the simplification of SAT instances by the application of asymmetric branching. Asymmetric branching allows for the tightening of clauses (removal of literals of a clause) and constitutes a powerful technique in terms of quality. We proposed an algorithm to enhance the technique by making its application independent of the order of literals in a clause. In this context, we proved that it is NP–hard to compute an optimal tightening of a clause. We further described how hyper–binary resolution can be incorporated in the enhanced algorithm, as this is done for common asymmetric branching. We evaluated the practical application of the presented approach by considering several different aspects. Although the quality of tightening can clearly be improved, the technique has to be applied selectively to keep the computational effort acceptable.

A fast execution of Boolean constraint propagation (BCP) is utterly essential for state–of–the–art SAT solvers. Many improvements have been proposed and engineered to improve BCP in terms of runtime. In contrast to this, we studied the improvement of BCP in terms of quality in Chapter 4. More precisely, we proposed an idea on how to enhance BCP by considering clauses for propagation that are not required to be unit under the partial assignment. We presented two main approaches to put the theoretical idea into practice. Thereupon, we suggested different heuristics to improve the enhancement of propagation. We analysed the quality as well as the computational effort for both approaches and the proposed heuristics. Ultimately, the most superficial heuristic proved itself to be the most efficient in practice. It clearly outperforms the pure application of unit propagation within BCP.

In Chapter 5, we moved further away from CDCL SAT solving. We examined an approach proposed by Goldberg [Gol08a] to use a so–called reference point for decision making (DMRP). A reference point is basically a complete assignment to the variables of the formula. Any decision is based on the set of clauses that are falsified by the reference point. As a consequence, the solver has to maintain more information during the search. We presented a data structure and an efficient implementation of the DMRP algorithm. Moreover, we proposed different heuristics for the practical application of DMRP. Particularly, we suggested the hybridisation of DMRP and CDCL solving. While the pure application of our DMRP implementation cannot compete with CDCL, the hybrid approach exhibits a considerable improvement over pure CDCL solving. Furthermore, the evaluation of different solver configurations indicated a significant improvement for particular families of SAT instances.

We incorporated most of the presented techniques into a multithreaded SAT solver. In the spirit of this thesis, we used a rather uncommon implementation of the solver. Our parallel solver SArTagnan shares clauses logically and physically. Most parallel SAT solvers enable the logical sharing of clauses but keep different copies of the same clause in memory. In Chapter 6, we described the implementation of SArTagnan with a focus on avoiding mutex–locks by simultaneously sharing data among several threads. The extensive sharing of data offers the advantage of easily exchanging valuable information, such as the tightening of a clause. On the other hand, the performance of the solver for one instance is likely to vary in different runs. This nondeterminism makes it difficult to optimise the solver in practice.

In the area of practical SAT solving, international competitions are organised to evaluate solvers and new solving approaches reasonably and transparently [Sat11, Sat10]. While working on this thesis, we implemented several solving methods and different kinds of solvers. The first version of our sequential solver SApperloT has won a silver medal in the SAT competition of 2009. Our parallel solver SArTagnan, which we described in Chapter 6, was awarded the best student solver in the parallel track within the SAT–Race of 2010. In the SAT competition of 2011, a MUS (minimal unsatisfiable subsets) track was offered for the first time. We briefly described the main ideas of our MUS solver MoUsSaka in Section 7.5. The solver is based on SApperloT and finished third in the plain MUS track. Its success is also based on the improved data structure which uses the `XOR` compression, as described in Section 3.1.

Directions for future work

In Section 3.4, we suggested a heuristic to apply our asymmetric branching variant AB^* for a subset of clauses in between CDCL searches. Values of variables that are stored by the phase saving heuristic [PD07a] are interpreted as a complete assignment. Clauses that are falsified by this implicit assignment are likely to be hard constraints for the solver. The restriction of costly simplification to the set of these hard clauses improved the overall performance for the application of AB^*. Further heuristics may choose other reasonable subsets of clauses for which asymmetric branching or AB^* are applied. As the techniques are very powerful in terms of quality (removal of literals), the higher computational costs may be worthwhile, when the considered constraints are known to be important for the solver. Particularly, the application of simplification techniques in between CDCL searches (inprocessing) may use different properties to estimate the importance of a clause.

The suggested extension of BCP (Chapter 4) considers the set of binary clauses of a formula to detect implied assignments. In particular, the binary implications of all unassigned literals of a clause are inspected. If a variable assignment is implied by all unassigned literals of a clause, the assignment can be applied immediately. The evaluation indicates that there are many of these inevitable implications. The presented methods only consider the binary clauses of a formula. However, during a search there are plenty of clauses that are binary under the current partial assignment. These clauses extend the binary implications at the current decision level in the search. It would be interesting to study the impact of these (temporary) binary implications. Especially for hard instances, when search goes deep and reaches high decision levels, a further improvement of BCP could turn to account, as it might help the solver to escape from these states of the search more quickly.

The presented implementation of DMRP has proven to be useful when applied reasonably. The combination of DMRP and CDCL to a hybrid solver as well as the application of DMRP within parallel solving improved the performance of the solvers. We see a promising application of DMRP in the context of MUS solving. Marques-Silva *et el.* presented the (recursive) model rotation technique to speed up MUS computation [MSL11, BMS11]. Basically, if an assignment τ was found that satisfies all but one clauses of the formula, model rotation aims to find further assignments (close to τ) that also satisfy all but one clauses. For each assignment τ' that is found to satisfy $\mathcal{F} \setminus C'$, the solver knows that C' has to be contained in the minimal unsatisfiable formula. However, this kind of search, which is based on a complete assignment, is thoroughly applied by the DMRP approach. A careful adaption of a DMRP implementation for the application within a MUS solver has therefore a good chance to outperform (recursive) model rotation.

We are convinced that studying and engineering advanced SAT solving methods constitutes an important issue for practical SAT research. The number of applications where SAT is used as backend technology still keeps on increasing. Solving techniques that are successful on instances of particular applications may thus gain even more importance in the future.

Bibliography

References

[ABH$^+$08] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. A Generalized Framework for Conflict Analysis. In *SAT 2008: 11th International Conference on Theory and Applications of Satisfiability Testing*, pages 21–27, 2008.

[ABL$^+$10] Josep Argelich, Daniel Le Berre, Inês Lynce, João P. Marques-Silva, and Pascal Rapicault. Solving linux upgradeability problems using boolean optimization. In *LoCoCo*, pages 11–22, 2010.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[AKS10] Gilles Audemard, George Katsirelos, and Laurent Simon. A restriction of extended resolution for clause learning sat solvers. In *AAAI*, 2010.

[ALMS11] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezeing and reactivating learnt clauses. In *SAT 2011: 14th International Conference on Theory and Applications of Satisfiability Testing*, pages 188–200, 2011.

[APT79] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas. *Information Processing Letters*, 8:121–123, 1979.

[AS08] Gilles Audemard and Laurent Simon. Experimenting with small changes in conflict-driven clause learning algorithms. In *CP*, pages 630–634, 2008.

[AS09] Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *IJCAI 2009: International Joint Conference on Aritifical Intelligence*, pages 399–404, 2009.

[Bac02a] Fahiem Bacchus. Enhancing Davis Putnam with Extended Binary Clause Reasoning. In *18th AAAI Conference on Artificial Intelligence*, pages 613–619, 2002.

[Bac02b] Fahiem Bacchus. Exploring the Computational Tradeoff of more Reasoning and Less Searching. In *SAT 2002: Fifth International Symposium on Theory and Applications of Satisfiability Testing*, pages 7–16, 2002.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

[Ber01] Daniel Le Berre. Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9:59–80, 2001.

[BFMP09] Ramón Béjar, Cèsar Fernández, Carles Mateu, and Nuria Pascual. Bounding the phase transition on edge matching puzzles. In *ISMVL*, pages 80–85, 2009.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability.* Frontiers in Artificial Intelligence and Applications. IOS Press, 2009.

[Bie08a] Armin Biere. Adaptive Restart Strategies for Conflict Driven SAT Solvers. In *SAT 2008: 11th International Conference on Theory and Applications of Satisfiability Testing*, pages 28–33, 2008.

[Bie08b] Armin Biere. PicoSAT Essentials. *JSAT Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[Bie09a] Armin Biere. Lazy Hyper Binary Resolution. Algorithms and Applications for Next Generation SAT Solvers, Dagstuhl Seminar 09461, Dagstuhl, Germany, 2009.

[BK04] Tobias Brüggemann and Walter Kern. An improved deterministic local search algorithm for 3-SAT. *Theoretical Computer Science*, 329(1-3):303–313, 2004.

[BMS11] Anton Belov and João Marques-Silva. Accelerating mus extraction with recursive model rotation. In *FMCAD 2011: 11th International Conference on Formal Methods in Computer-Aided Design*, pages 37–40, 2011.

[BP08] Daniel Le Berre and Anne Parrain. On sat technologies for dependency management and beyond. In *SPLC (2)*, pages 197–200, 2008.

[Bra01] Ronen I. Brafman. A simplifier for propositional formulas with many binary clauses. In *IJCAI 2001: International Joint Conference on Artificial Intelligence*, pages 515–522, 2001.

[Bra04] Ronen I. Brafman. A simplifier for propositional formulas with many binary clauses. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 34(1):52–59, 2004.

[BSK03a] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. A Universal Parallel SAT Checking Kernel. In *PDPTA 2003: International Conference on Parallel and Distributed Processing Techniques and Applications*, 2003.

[BSK03b] Wolfgang Blochinger, Carsten Sinz, and Wolfgang Küchlin. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing*, 29(7):969–994, 2003.

[BW03] Fahiem Bacchus and Jonathan Winter. Effective Preprocessing with Hyper-Resolution and Equality Reduction. In *SAT 2003: Sixth International Conference on Theory and Applications of Satisfiability Testing*, pages 341–355, 2003.

[CHS09] Geoffrey Chu, Aaron Harwood, and Peter J. Stuckey. Cache Conscious Data Structures for Boolean Satisfiability Solvers. *JSAT Journal on Satisfiability, Boolean Modeling and Computation*, 6:99–120, 2009.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176, 2004.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[Coo71] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *STOC 1971: Third Annual ACM Symposium on Theory of Computing*, 1971.

[Coo76] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32, October 1976.

[DDD$^+$05] Sylvain Darras, Gilles Dequen, Laure Devendeville, Bertrand Mazure, Richard Ostrowski, and Lakhdar Sais. Using boolean constraint propagation for sub-clauses deduction. In *CP 2005: 11th International Conference on Principles and Practice of Constraint Programming*, pages 757–761, 2005.

[DGS07] Bistra Dilkina, Carla P. Gomes, and Ashish Sabharwal. Tradeoffs in the complexity of backdoor detection. In *CP 2007: 13th International Conference on Principles and Practice of Constraint Programming*, 2007.

[DGS09] Bistra Dilkina, Carla P. Gomes, and Ashish Sabharwal. Backdoors in the context of learning. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, SAT '09, pages 73–79, Berlin, Heidelberg, 2009. Springer-Verlag.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[EB05] Niklas Eén and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *SAT 2005: Eighth International Conference on Theory and Applications of Satisfiability Testing*, pages 61–75, 2005.

[ES03] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT 2003: Sixth International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[FDH05] Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel Multithreaded Satisfiability Solver: Design and Implementation. *Electr. Notes Theor. Comput. Sci.*, 128(3):75–90, 2005.

[FG03] John Franco and Allen Van Gelder. A perspective on certain polynomial-time solvable classes of satisfiability. *Discrete Applied Mathematics*, 125(2-3):177–214, 2003.

[FGMS07] Olivier Fourdrinoy, Éric Grégoire, Bertrand Mazure, and Lakhdar Sais. Eliminating Redundant Clauses in SAT Instances. In *CPAIOR 2007: Fourth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 71–83, 2007.

[FP08] Fabrizio Frati and Maurizio Patrignani. A Note on Minimum Area Straight-line Drawings of Planar Graphs. In *GD 2007: 15th International Symposium on Graph Drawing*, 2008.

[Fuk04] Alex S. Fukunaga. Efficient Implementations of SAT Local Search. In *SAT 2004: Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[Gel02] Allen Van Gelder. Generalizations of watched literals for backtracking search. In *Senventh International Symposiums on Artificial Intelligence and Mathematics*, 2002.

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[GM10] Eugene Goldberg and Panagiotis Manolios. Sat-solving based on boundary point elimination. In *Haifa Verification Conference*, pages 93–111, 2010.

[GN07] Eugene Goldberg and Yakov Novikov. BerkMin: A fast and robust Sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.

[Gol02] Eugene Goldberg. Testing satisfiability of cnf formulas by computing a stable set of points. In *CADE*, pages 161–180, 2002.

[Gol05] Eugene Goldberg. Testing satisfiability of cnf formulas by computing a stable set of points. *Ann. Math. Artif. Intell.*, 43(1):65–89, 2005.

[Gol06] Eugene Goldberg. Determinization of resolution by an algorithm operating on complete assignments. In *SAT 2006: Ninth International Conference on Theory and Applications of Satisfiability Testing*, 2006.

[Gol08a] Eugene Goldberg. A decision-making procedure for resolution-based SAT-solvers. In *SAT 2008: 11th International Conference on Theory and Applications of Satisfiability Testing*, 2008.

[Gol08b] Eugene Goldberg. A resolution based sat-solver operating on complete assignments. *JSAT*, 5(1-4):217–242, 2008.

[Gol09] Eugene Goldberg. Boundary points and resolution. In *SAT*, pages 147–160, 2009.

[GS05] Roman Gershman and Ofer Strichman. Cost-effective hyper-resolution for preprocessing cnf formulas. In *SAT 2005: Eighth International Conference on Theory and Applications of Satisfiability Testing*, pages 423–429, 2005.

[GSM10] Graeme Gange, Peter J. Stuckey, and Kim Marriott. Optimal k-level planarization and crossing minimization. In *Graph Drawing*, pages 238–249, 2010.

[GT93] Allen Van Gelder and Yumi K. Tsuji. Satisfiability Testing with More Reasoning and Less Guessing. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge.* American Mathematical Society, 1993.

[Heu08a] Marijn Heule. *SmArT Solving.* PhD thesis, Technische Universiteit Delft, 2008.

[Heu08b] Marijn J.H. Heule. Solving edge-matching problems with satisfiability solvers. In *Second International Workshop on Logic and Search (LaSh 2008)*, pages 88–102. University of Leuven, 2008.

[HJB11] Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. Efficient cnf simplification based on binary implication graphs. In *SAT 2011: 14th International Conference on Theory and Applications of Satisfiability Testing*, pages 201–215, 2011.

[HJPS11] Youssef Hamadi, Saïd Jabbour, Cédric Piette, and Lakhdar Sais. Deterministic parallel DPLL. *JSAT*, 7(4):127–132, 2011.

[HJS09a] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel sat solving. In *Proceedings of the 21st international joint conference on Artifical Intelligence*, pages 499–504, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[HJS09b] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *JSAT Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2009.

[HJS10] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Learning for Dynamic Subsumption. *International Journal on Artificial Intelligence Tools*, 19(4):511–529, 2010.

[HLMS06] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic–sized nonblocking work stealing deque. *Distributed Computing*, 18(3): 189–207, 2006.

[HM04] Marijn Heule and Hans Van Maaren. Aligning cnf- and equivalence-reasoning. In *SAT 2004: Seventh International Conference on Theory and Applications of Satisfiability Testing*, pages 174–181. Springer, 2004.

[Hoo98] Holger H. Hoos. *Stochastic local search - methods, models, applications.* PhD thesis, Technische Universität Darmstadt, 1998.

[HS04] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications.* Elsevier / Morgan Kaufmann, 2004.

[HS07] HyoJung Han and Fabio Somenzi. Alembic: An efficient algorithm for cnf preprocessing. In *DAC*, pages 582–587, 2007.

[HS08] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[HS09] HyoJung Han and Fabio Somenzi. On-the-fly clause improvement. In *SAT 2009: 12th International Conference on Theory and Applications of Satisfiability Testing*, 2009.

[Hua07a] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI 2007: International Joint Conference on Artificial Intelligence*, pages 2318–2323, 2007.

[Hua10] Jinbo Huang. Extended clause learning. *Artificial Intelligence*, 174(15):1277–1284, 2010.

[HW08] Shai Haim and Toby Walsh. Online estimation of sat solving runtime. In *SAT*, pages 133–138, 2008.

[Int03] Yannet Interian. Backdoor sets for random 3-sat. In *SAT 2003: Sixth International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[IYG$^+$08] F. Ivancic, Z. Yang, M. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2008.

[JBH10] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked Clause Elimination. In *TACAS 2010: 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 129–144, 2010.

[Kai03] Andreas Kaiser. *Beweisertechnologien für die Produktdatenverwaltung.* PhD thesis, Universität Tübingen, 2003.

[KK97] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *DAC 1997: 34th Design Automation Conference*, pages 263–268, 1997.

[KK13] Michael Kaufmann and Wolfgang Küchlin. StrAlEnSATs – Structure Based Algorithm Engineering for SAT Solving, 2007-2013. Deutsche Forschungsgemeinschaft, SPP 1307 – Algorithm Engineering, `http://www.algorithm-engineering.de`.

[KPKG02] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.

[Kru07] Markus Krug. Minimizing the Area for Planar Straight-Line Grid Drawings. Master's thesis, University of Karlsruhe, 2007.

[KS92] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI 1992: Tenth European Conference on Artificial Intelligence*, pages 359–363, 1992.

[KS00] Wolfgang Küchlin and Carsten Sinz. Proving consistency assertions for automotive product data management. *Journal of Automated Reasoning*, 24(1–2):145–163, 2000.

[KS10] Konstantin Kutzkov and Dominik Scheder. Using csp to improve deterministic 3-sat. *CoRR*, abs/1007.1166, 2010.

[KSMS11] Hadi Katebi, Karem A. Sakallah, and João P. Marques-Silva. Empirical Study of the Anatomy of Modern Sat Solvers. In *SAT 2011: 14th International Conference on Theory and Applications of Satisfiability Testing*, pages 343–356, 2011.

[Kul99a] O. Kullmann. On a generalization of extended resolution. *Discrete Applied Mathematics*, 96-97(1):149–176, 1999.

[Kul99b] Oliver Kullmann. New methods for 3-sat decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999.

[Kul00] Oliver Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, 107(1-3):99–137, 2000.

[Kul04] Oliver Kullmann. The combinatorics of conflicts between clauses. In *SAT 2004: Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[KW07] Marcus Krug and Dorothea Wagner. Minimizing the Area for Planar Straight-Line Grid Drawings. In *GD 2007: 15th International Symposium on Graph Drawing*, 2007.

[LA97] Chu Min Li and Anbulagan. Look-Ahead Versus Look-Back for Satisfiability Problems. In *CP 1997: Third International Conference on Principles and Practice of Constraint Programming*, 1997.

[Lew78] Harry R. Lewis. Renaming a Set of Clauses as a Horn Set. *Journal of the ACM*, 25:134–135, 1978.

[LMS06] Ines Lynce and Joao Marques-Silva. SAT in bioinformatics: Making the case with haplotype inference. In *SAT 2006: Ninth International Conference on Theory and Applications of Satisfiability Testing*, pages 136–141, 2006.

[LSB07] Matthew D. T. Lewis, Tobias Schubert, and Bernd Becker. Multi-threaded SAT solving. In *12th Asia and South Pacific Design Automation Conference*, 2007.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. In *ISTCS 1993: Second Israel Symposium on Theory of Computing Systems*, pages 128–133, 1993.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *DAC 2001: 38th Design Automation Conference*, 2001.

[MS99] J. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *EPIA 1999: Ninth Portuguese Conference on Artificial Intelligence*, pages 62–74, London, UK, 1999. Springer-Verlag.

[MS08] João P. Marques-Silva. Practical Applications of Boolean Satisfiability. In *WODES 2008: Workshop on Discrete Event Systems*, pages 74–80, 2008.

[MSL11] João P. Marques-Silva and Inês Lynce. On improving mus extraction algorithms. In *SAT 2011: 14th International Conference on Theory and Applications of Satisfiability Testing*, pages 159–173, 2011.

[MSS96] Joao P. Marques-Silva and Karem A. Sakallah. Grasp a new search algorithm for satisfiability. In *ICCAD 1996: IEEE/ACM International Conference on Computer-aided design*, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[MSS99] Joao P. Marques-Silva and Karem A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[Nad10] Alexander Nadel. Boosting Minimal Unsatisfiable Core Extraction. In *FMCAD 2010: 10th International Conference on Formal Methods in Computer-Aided Design*, 2010.

[Nie02] Rolf Niedermeier. Invitation to Fixed-Parameter Algorithms. Habilitation, Universität Tübingen, 2002.

[NLBD+04] Eugene Nudelman, Kevin Leyton-Brown, Alex Devkar, Yoav Shoham, and Holger Hoos. Understanding random SAT: Beyond the clauses-to-variables ratio. In *CP 2004: Tenth International Conference on Principles and Practice of Constraint Programming*, pages 438–452, 2004.

[NZ02] Stefan Näher and Oliver Zlotowski. Design and Implementation of Efficient Data Types for Static Graphs. In *ESA 2002 Tenth Annual European Symposium on Algorithms*, pages 157–164. Springer, 2002.

[PD07a] Knot Pipatsrisawat and Adnan Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *SAT 2007: Tenth International Conference on Theory and Applications of Satisfiability Testing*, pages 294–299, 2007.

[PD10] Knot Pipatsrisawat and Adnan Darwiche. On modern clause-learning satisfiability solvers. *Journal of Automated Reasoning*, 44(3):277–301, 2010.

[PHS08] Cedric Piette, Youssef Hamadi, and Lakhdar Sais. Vivifying propositional clausal formulae. In *ECAI 2008: 18th European Conference on Artificial Intelligence*, pages 525–529, Patras (Greece), jul 2008.

[PS04] Stefan Porschen and Ewald Speckenmeyer. Worst case bounds for some NP-complete modified Horn-SAT problems. In *SAT 2004: Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[PS07] Stefan Porschen and Ewald Speckenmeyer. Satisfiability of mixed Horn formulas. *Discrete Applied Mathematics*, 155(11):1408–1419, 2007.

[RHN06] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.

[Rob79] John Alan Robinson. *Logic, form and function: the mechanization of deductive reasoning.* Artificial Intelligence Series. North-Holland, 1979.

[Rob83] John Alan Robinson. Automatic deduction with hyper-resolution. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 1: Classical Papers on Computational Logic 1957-1966*, pages 416–423. Springer, 1983.

[Rya04] Lawrence Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, School of Computing Science, Simon Fraser University, 2004.

[SB09] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *SAT 2009: 12th International Conference on Theory and Applications of Satisfiability Testing*, pages 237–243, 2009.

[Sch99] Uwe Schöning. A probabilistic algorithm for k-sat and constraint satisfaction problems. In *Symposium on Foundations of Computer Science*, 1999.

[Sht01] Ofer Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. *Lecture Notes in Computer Science*, 2144:58–70, 2001.

[SI09] Carsten Sinz and Markus Iser. Problem-sensitive restart heuristics for the DPLL procedure. In *SAT 2009: 12th International Conference on Theory and Applications of Satisfiability Testing*, pages 356–362, 2009.

[SKC93] Bart Selman, Henry Kautz, and Bram Cohen. Local Search Strategies for Satisfiability Testing. In D. S. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, pages 521–532. American Mathematical Society, 1993.

[SKK03] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design,Analysis and Manufacturing*, 17:75–97, 2003.

[SLB05] Tobias Schubert, Matthew D. T. Lewis, and Bernd Becker. PaMira - A Parallel SAT Solver with Knowledge Sharing. In *MTV 2005: Sixth International Workshop on Microprocessor Test and Verification, Common Challenges and Solutions*, pages 29–36, 2005.

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *AAAI 1992: Tenth National Conference on Artificial Intelligence*, 1992.

[SP04] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. NiVER: Non Increasing Variable Elimination Resolution for Preprocessing SAT instances. In *SAT 2004: Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[SS07] Marko Samer and Stefan Szeider. Backdoor Sets of Quantified Boolean Formulas. In *SAT 2007: Tenth International Conference on Theory and Applications of Satisfiability Testing*, 2007.

[Sze05] Stefan Szeider. Backdoor sets for dll subsolvers. *Journal of Automated Reasoning*, 35:73–88, 2005.

[Tse68] G.S. Tseitin. On the complexity of derivation in propositional calculus. *Structures in Constructive Mathematics and Mathematical Logic*, Part II:115–125, 1968.

[Vel02] Miroslav N. Velev. Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer. In *DATE 2002: Design Automation and Test in Europe*, 2002.

[Vel04] Miroslav N. Velev. Comparative study of strategies for formal verification of high-level processors. In *ICCD 2004: IEEE International Conference on Computer Design*, pages 119–124, 2004.

[Vel07] Miroslav N. Velev. Exploiting hierarchy and structure to efficiently solve graph coloring as sat. In *ICCAD 2007: International Conference on Computer-Aided Design*, pages 135–142, Piscataway, NJ, USA, 2007. IEEE Press.

[VG11] Miroslav N. Velev and Ping Gao. Automatic formal verification of multithreaded pipelined microprocessors. In *ICCAD 2011: IEEE/ACM International Conference on Computer-aided design*, pages 679–686, 2011.

[Wah07] Magnus Wahlström. *Algorithms, measures, and upper bounds for satisfiability and related problems.* PhD thesis, Linköping University, 2007. Dissertation no 1079.

[WGS03a] R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In *IJCAI 2003: International Joint Conference on Artificial Intelligence*, 2003.

[WGS03b] R. Williams, C. Gomes, and B. Selman. On the connections between backdoors, restarts, and heavy-tailedness in combinatorial search. In *SAT 2003: Sixth International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[XHHLB08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[XHLB07] Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Hierarchical Hardness Models for SAT. In *CP 2007: 13th International Conference on Principles and Practice of Constraint Programming*, pages 696–711, 2007.

[ZBH96] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *ICCAD 2001: International Conference on Computer-Aided Design*, 2001.

[ZS02] Lei Zheng and Peter J. Stuckey. Improving SAT using 2SAT. In *ACSC 2002: 25th Australasian Computer Science Conference*, pages 331–340. E, 2002.

— **Solver Presentations** —

[AS12] Gilles Audemard and Laurent Simon. Glucose. Solver descriptions at `http://www.lri.fr/~simon/?page=glucose`, 2012.

[Bie09b] Armin Biere. PrecoSAT Solver Description. `http://fmv.jku.at/precosat/preicosat-sc09.pdf` 2009, 2009.

[Bie11] Armin Biere. Lingeling and Friends at the SAT Competition 2011. FMV Reports Series 11/1, Johannes Kepler University, Institute for Formal Models and Verification, Altenbergerstr. 69, 4040 Linz, Austria, March 2011.

[ES12] Niklas Eén and Niklas Sörensson. MiniSat. Solver descriptions at `http://minisat.se`, 2012.

[Hua07b] Jinbo Huang. TiniSat. Solver description at `http://satcompetition.org`, 2007.

[PD07b] Knot Pipatsrisawat and Adnan Darwiche. Rsat 2.0. Solver description at `http://satcompetition.org`, 2007.

[Rou11] Oliver Roussel. ppfolio. Solver description at `http://www.cril.univ-artois.fr/~roussel/ppfolio`, 2011.

[Soo12] Mate Soos. CryptoMiniSat. Solver descriptions at `http://msoos.org`, 2012.

— Other Resources —

[BWG12] bwGRID – a distributed aggregation of computer nodes, operated by eight Baden–Württemberg state universities. `http://www.bw-grid.de`, 2010-2012.

[GW07] Dirk Gieselmann and Wolfgang Weber. Mit zehneinviertel Mann. *11 Freunde*, 66, 05 2007. `http://www.11freunde.de/artikel/wolfgang-weber-und-das-europacup-drama`.

[Ope12] The OpenMP API. `http://openmp.org`, 1997-2012.

[Sat10] SAT-Race. `http://baldur.iti.uka.de`, 2006-2010.

[Sat11] The international SAT competition. `http://satcompetition.org`, 2002-2011.

Author's Publications

[BKK12] Sebastian Burg, Michael Kaufmann, and Stephan Kottler. Creating industrial-like SAT instances by clustering and reconstruction (Poster Presentation). In *SAT 2012: 15th International Conference on Theory and Applications of Satisfiability Testing*, pages 471–472, 2012.

[KK10] Michael Kaufmann and Stephan Kottler. Proving or Disproving Planar Straight-Line Embeddability onto given Rectangles. In *GD 2009: 17th International Symposium on Graph Drawing*, pages 419–420, 2010.

[KK11a] Michael Kaufmann and Stephan Kottler. Beyond Unit Propagation in SAT Solving. In *SEA 2011: Tenth International Symposium on Experimental Algorithms*, pages 267–279, 2011.

[KK11b] Stephan Kottler and Michael Kaufmann. A parallel portfolio SAT solver with lockless physical clause sharing. Technical report, Universitätsbibliothek Tübingen, Wilhelmstr. 32, 72074 Tübingen, 2011.

[KK11c] Stephan Kottler and Michael Kaufmann. SArTagnan - A parallel portfolio SAT solver with lockless physical clause sharing. In *PoS 2011: Pragmatics of SAT*, 2011.

[KKS08a] Stephan Kottler, Michael Kaufmann, and Carsten Sinz. A New Bound for an NP-Hard Subclass of 3-SAT Using Backdoors. In *SAT 2008: 11th International Conference on Theory and Applications of Satisfiability Testing*, pages 161–167, 2008.

[KKS08b] Stephan Kottler, Michael Kaufmann, and Carsten Sinz. Computation of Renameable Horn Backdoors. In *SAT 2008: 11th International Conference on Theory and Applications of Satisfiability Testing*, pages 154–160, 2008.

[Kot07] Stephan Kottler. Backdoors in SAT-Instanzen. Diplomarbeit, Universität Tübingen, 2007.

[Kot10a] Stephan Kottler. SAT Solving with Reference Points. In *SAT 2010: 13th International Conference on Theory and Applications of Satisfiability Testing*, pages 143–157, 2010.

[KZSK12] Stephan Kottler, Christian Zielke, Paul Seitz, and Michael Kaufmann. Exploring recurring patterns in conflict analysis of CDCL SAT solvers (Tool Presentation). In *SAT 2012: 15th International Conference on Theory and Applications of Satisfiability Testing*, pages 449–455, 2012.

— Unrelated to Satisfiability —

[AEH+10] Benjamin Albrecht, Philip Effinger, Markus Held, Michael Kaufmann, and Stephan Kottler. Visualization of Complex BPEL Models. In *GD 2009: Proceedings of the 17th International Symposium on Graph Drawing*, pages 421–423, 2010.

[LK07] Katharina A. Lehmann and Stephan Kottler. Visualizing Large and Clustered Networks. In *GD 2006: 14th International Symposium on Graph Drawing*, pages 240–251, 2007.

— Solver Descriptions —

[Kot09] Stephan Kottler. Solver descriptions for the SAT competition 2009. `http://satcompetition.org`, 2009.

[Kot10b] Stephan Kottler. Solver descriptions for the SAT race 2010. `http://baldur.iti.uka.de/sat-race-2010`, 2010.

[Kot11] Stephan Kottler. Solver descriptions for the SAT competition 2011. `http://satcompetition.org`, 2011.

Co-supervised Student Work

[Alb09] Benjamin Albrecht. Ein pfadorientierter hierarchischer Layouter. Studienarbeit, Universität Tübingen, 2009.

[Bur10] Sebastian Burg. Generation von SAT Instanzen mit Ähnlichkeit zu industriellen Instanzen. Diplomarbeit, Universität Tübingen, 2010.

[Gro08] Christian Groth. Analyse verschiedener Variablenverteilungen für k-SAT. Studienarbeit, Universität Tübingen, 2008.

[Hef09] Walentin Heft. Visualisierung von Backdoors in SAT Instanzen. Studienarbeit, Universität Tübingen, 2009.

[Lah11] Martin Lahl. Reduktion von Unerfüllbarkeitsbeweisen. Studienarbeit, Universität Tübingen, 2011.

[Lah12] Martin Lahl. Beweisbasierte Berechnung von minimal unerfüllbaren Kernen. Diplomarbeit, Universität Tübingen, 2012.

[Sch10] Christoph Schmidt. Engineering an efficient Implementation for parameterized 3-Hitting Set. Studienarbeit, Universität Tübingen, 2010.

[Sei10] Paul Seitz. Ein Tool zur visuellen Analyse von Konfliktgraphen beim SAT Solving. Bachelor's thesis, Universität Tübingen, 2010.

[Zie10] Christian Zielke. Effizientes Preprocessing von Erfüllbarkeitsinstanzen mit Anwendung auf phylogenetische Probleme. Master's thesis, Universität Tübingen, 2010.

List of Figures and Tables

List of Algorithms