Generalised SAT Checking in a First-Order Logic Framework

Diploma Thesis

Author
Christoph Zengler

Supervisor
PD Dr. Thomas Sturm
University of Passau / Germany

Passau, 8th January 2008

Abstract

The SAT Problem is one of the core problems of theoretical computer science. Since it was proved to be NP-complete in 1971, many researchers have dealt with this problem. One of the reasons why the SAT Problem is of such great interest, is because many real world problems such as processor verification, model checking or planning problems can be encoded as SAT problems and so be solved with SAT solvers. Within the last decade algorithms for SAT solving have been improved continually and thus problems with hundreds of thousands of variables can be solved within minutes.

The concept of this thesis is to generalise SAT checking. The original SAT Problem is formulated for formulas in Propositional Logic. One can interpret these formulas as fully existentially quantified formulas in First-Order Logic over the language of Boolean Algebras. An extended version of the SAT Problem is the Q-SAT Problem, which raises the question whether a quantified formula in Propositional Logic is **true** or **false**. These kinds of formulas are equivalent to fully quantified formulas in First-Order Logic. For SAT and Q-SAT many efficient algorithms such as conflict driven clause learning exist. These were implemented and evaluated.

The last chapter is concerned with an extension of SAT and Q-SAT to handle arbitrary formulas in First-Order Logic, referred to as *parametric Q-SAT*. In contrast to all current algorithms free variables are allowed in our input formulas now. Thus the output is no longer **true** or **false** but a set of conditions to the free variables. Since with parametric Q-SAT it is feasiblie to compute an equivalent quantifier free formula for each arbitrary input formula, we derived a quantifier elimination procedure.

All of the algorithms in this thesis were implemented and evaluated in REDLOG, an extension of the Computer Algebra System REDUCE. It becomes apparent, that these new procedures are much faster than the original quantifier elimination in REDLOG. We can now handle formulas with thousands of variables and clauses, which was not possible before this implementation. This new implementation lifts the computational power of REDLOG to a higher level and can even compete with some Q-SAT solvers developed by research communities.

Contents

1 Introduction: The SAT Problem **5**
1.1 Propositional Logic . . . 5
1.1.1 Syntax . . . 5
1.1.2 Semantics . . . 5
1.2 The SAT Problem . . . 6
1.3 Classes of Algorithms for the SAT Problem . . . 7
1.3.1 The Deterministic DLL Algorithm . . . 7
1.3.2 Probabilistic Algorithms . . . 10
1.3.3 Conclusion: Algorithms in Current SAT-Solvers . . . 11

2 Implementing the SAT Algorithm **12**
2.1 Boolean Algebra in REDLOG . . . 12
2.1.1 Propositional Logic Within First-Order Logic . . . 12
2.1.2 REDLOG Commands . . . 14
2.2 The Data Structure for our SAT Solver . . . 15
2.2.1 The Variables . . . 15
2.2.2 The Clauses . . . 16
2.2.3 Implementation of the Lists . . . 16
2.2.4 Setting and Unsetting Variables . . . 17
2.3 Branching Heuristics . . . 18
2.3.1 Literal Count Heuristics . . . 18
2.3.2 MOM's Heuristic . . . 19
2.3.3 Activity Heuristics . . . 19
2.3.4 Comparison of the Heuristics . . . 20
2.4 Conflict Driven Clause Learning . . . 20
2.4.1 The Iterative Version of the DLL Algorithm . . . 22
2.4.2 The Semantics of Clause Learning . . . 23
2.4.3 Correctness and Termination of CDCL . . . 25
2.5 UP and Watched Literals . . . 26
2.6 Further Improvements . . . 27
2.6.1 Pre-Processing . . . 27
2.6.2 Simplifications . . . 27
2.6.3 Restarts . . . 28
2.6.4 Clause Deletion . . . 28
2.7 Comparison to QE and to SAT Solvers . . . 28
2.7.1 The Benchmarks . . . 28
2.7.2 Benchmark Results . . . 29
2.7.3 Comparison to Modern SAT Solvers . . . 30

3 The Next Step: Q-SAT **34**
3.1 The Q-SAT Problem . . . 34
3.1.1 Description of the Problem . . . 35

3.1.2 Illustration of the Q-SAT Problem 35
3.1.3 Complexity of Q-SAT 37
3.2 The Q-SAT Version of the CDCL Algorithm 38
3.2.1 Changes to the Data Structure 38
3.2.2 The Q-SAT Algorithm 39
3.2.3 The New Rules for UP and Empty Clauses 41
3.2.4 Correctness of Resolution 42
3.3 Comparison to QE and to Other Q-SAT Solvers 43
3.3.1 Comparison to QE 43
3.3.2 Comparison to Modern Q-SAT Solvers 45

4 Quantifier Elimination with Parametric Q-SAT 46
4.1 Parametric Q-SAT 46
4.1.1 The Parametric Q-SAT Problem 46
4.1.2 Parametric Q-SAT as Quantifier Elimination 47
4.2 The Algorithm 47
4.2.1 Operation of the Algorithm 48
4.2.2 Complexity of the Algorithm 49
4.3 Benchmarks 49
4.4 Further Ideas and Outlook 51

Appendices 52

A The DIMACS File Format 52
A.1 SAT Problems: .cnf 52
A.2 Q-SAT Problems: .qdimacs 52

B User Manual 54
B.1 Reading Input Formulas 54
B.2 Options for the SAT Solver 55

References 60

1 Introduction: The SAT Problem

When speaking about SAT checking or the SAT problem we speak about one of the core problems of computer science. It was the first problem which was proven to be NP-complete by Cook in 1971 [Cook, 1971]. An important consequence of this theorem is that SAT can be reduced to any problem in NP in polynomial time. So finding fast algorithms for SAT induces fast algorithms for many other problems (e.g. colouring of graphs or constraint solving).

At first we will define syntax and semantics of Propositional Logic before proceeding to a description of the SAT problem. The second part of this chapter will provide an outline of the development of efficient algorithms for solving the SAT problem.

1.1 Propositional Logic

1.1.1 Syntax

In Propositional Logic there is a set of variables $\mathcal{V}$, constants $\top$, $\bot$ representing **true** and **false** and boolean operations $\neg, \wedge, \vee, \Longrightarrow$ and $\Longleftrightarrow$. A literal is a variable x or its negation $\neg x$. Formulas are defined inductively:

Definition 1.1 (Syntax of the Propositional Logic) *$\top$ and $\bot$ are formulas. Every variable $v \in \mathcal{V}$ is a formula. For formulas φ and ψ also $\neg\phi$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \Longrightarrow \psi)$ and $(\varphi \Longleftrightarrow \psi)$ are formulas.*

Note *We can write $(\varphi \Longrightarrow \psi) \wedge (\psi \Longrightarrow \varphi)$ instead of $(\varphi \Longleftrightarrow \psi)$ and $(\neg\varphi \vee \psi)$ instead of $(\varphi \Longrightarrow \psi)$. So we can express every formula with Boolean operations $\neg, \wedge$ and $\vee$.*

In this thesis small Latin letters such as x, y, z denote variables and small Greek letters denote formulas.

Definition 1.2 (Set of occurring variables) *$\mathcal{V}(\varphi)$ is the set of variables occurring in the formula φ. Its inductive definition is (with $x \in \mathcal{V}$ and φ and ψ formulas):*
$\mathcal{V}(x) = \{x\}$, $\mathcal{V}(\neg\varphi) = \mathcal{V}(\varphi)$, $\mathcal{V}(\varphi \wedge \psi) = \mathcal{V}(\varphi) \cup \mathcal{V}(\psi)$ *and* $\mathcal{V}(\varphi \vee \psi) = \mathcal{V}(\varphi) \cup \mathcal{V}(\psi)$

The size of formulas can also be defined inductively.

Definition 1.3 (Size of formulas) *For a formula φ, $|\varphi|$ denotes its size. $|x| = 1$ for a variable $x \in \mathcal{V}$. $|\neg\varphi| = |\varphi| + 1$, $|\varphi \wedge \psi| = |\varphi| + |\psi| + 1$ and $|\varphi \vee \psi| = |\varphi| + |\psi| + 1$*

1.1.2 Semantics

We receive a semantics of Propositional Logic by interpreting the variables with either **true** or **false**. In the next step we utilse the common laws of Boolean Algebras to derive a truth value for a formula. A more formal view leads to the following definition:

Definition 1.4 (Interpretation of a formula) *An interpretation is a possibly partial assignment α, mapping a variable $x \in \mathcal{V}$ to a truth value: $\alpha(x) \in \{\top, \bot\}$. The codomain of α is* $\mathrm{dom}(\alpha)$.

We denote such an interpretation α as a list of assignments:

$$[x_1 \longleftarrow \alpha(x_1),\ x_2 \longleftarrow \alpha(x_2),\ \dots\ , x_k \longleftarrow \alpha(x_k)] \tag{1.1}$$

If we interpret a formula φ with a (possibly partial) interpretation α, we write: (φ, α). If α interprets all variables of $\mathcal{V}(\varphi)$ it is called a *total interpretation.*

Definition 1.5 (Extension of an interpretation) *An interpretation β extends an interpretation α if $\mathrm{dom}(\alpha) \subseteq \mathrm{dom}(\beta)$ and for all $x \in \mathrm{dom}(\alpha)$ $\alpha(x) = \beta(x)$.*

If $(\varphi, \alpha) = \top$, we say "α satisfies φ" and denote $\alpha \models \varphi$.

Definition 1.6 (Satisfiability) *If for a formula φ an interpretation α with $\alpha \models \varphi$ exists, the formula is satisfiable. A formula is not satisfiable if no interpretation $\alpha \models \varphi$ exists. If every interpretation α satisfies φ, we refer to it as a tautology.*

At last we need the concept of equivalence of two formulas because we will reduce certain formulas later and we can only do so if the equivalence of the formulas holds.

Definition 1.7 (Equivalence of formulas) *Two formulas φ and ψ are equivalent ($\varphi \equiv \psi$) if $(\varphi, \alpha) = (\psi, \alpha)$ holds for every total interpretation α (total for φ and ψ).*

1.2 The SAT Problem

The SAT problem (satisfiability problem) is a decision problem. The question is whether there is a satisfying interpretation for a given input formula in Propositional Logic. So a SAT solving algorithm must decide on this question. In 1971 Cook proved the following important theorem about the complexity of the SAT Problem:

Theorem 1.1 (NP-completeness of SAT) *The SAT Problem is NP-complete.*

It can easily be perceived why the problem is in NP: One can guess an interpretation and check if it satisfies a given formula in polynomial time. This is the definition of NP. The proof that the problem also is NP-complete (i.e. $SAT \subseteq NP$) can be found in [Cook, 1971].

Now we can restrict the problem to a certain class of formulas: conjunctive normal forms.

Definition 1.8 (Conjunctive normal form) *A formula φ in Propositional Logic is in conjunctive normal form (CNF) if it is a conjunction of clauses. A clause is a disjunction of literals x_{ij}. So φ can be written as*

$$\varphi := \bigwedge_{i=0}^{n} \bigvee_{j=0}^{m_i} x_{ij} \tag{1.2}$$

If the maximum length of a single clause is k, we refer to ϕ as formula in k-CNF. The k-SAT problem is the general SAT problem with the restriction that input formulas are in k-CNF.

Theorem 1.2 (NP-completeness of *k*-SAT) *For $k > 2$ the k-SAT problem is NP-complete.*

There are several polynomial-time algorithms to turn a general formula in Propositional Logic into a k-CNF formula, especially into a 3-SAT formula. However, one of two disadvantages always arises. Either we can expand the formula to 3-CNF with the worst-case of an exponential blow-up of the formula-length, or we can avoid this blow-up by introducing new variables (linear in the number of operators). Nevertheless most of the current algorithms are specialised for the k-SAT problem. So an arbitrary formula must be converted to k-CNF before the algorithm can function.

The code for converting arbitrary formulas into CNF formulas was implemented in REDLOG. But it came apparent that it is always much slower to convert and solve non-CNF formulas with the SAT procedure instead of solving them with the standard quantifier elimination. Thus if a formula is not in CNF, the standard quantifier elimination procedure is utilised.

1.3 Classes of Algorithms for the SAT Problem

There are two different concepts for solving SAT problems. The first concept is a deterministic backtracking solution originating from a paper of Davis, Logemann and Loveland in 1962, [Davis *et al.*, 1962]. The second concept is to solve SAT with probabilistic algorithms. Important advocates for this approach are Pudlak, Zane or Schöning.

In the following we will look at both paradigms before we will examine the DLL algorithm in more detail in chapter 2.

1.3.1 The Deterministic DLL Algorithm

It is obvious that a solution for a SAT problem can be derived trivially in $O(2^n)$ where n is the number of different variables of the formula. This is realised by simply testing each possible assignment. In [Davis *et al.*, 1962] the following result is of central interest: If for a clause C of a formula φ and a partial assignment α holds $(C, \alpha) = \bot$ it also holds that $(C, \beta) = \bot$ for each extension $b \supseteq \alpha$. So we can cut the search tree at the time one clause is **false** which can be realised via backtracking. The following basic algorithm demonstrates

the concept. The algorithm is refered to as DLL (Davis, Logeman, Loveland) .

```
   Input: Formula φ in Propositional Logic, interpretation α
   Output: true or false
 1 DLL(φ, α)
 2 if (φ, α) = ⊥ then
 3 |  return false
 4 end
 5 if (φ, α) = ⊤ then
 6 |  return true
 7 end
 8 choose an unassigned x ∈ 𝒱(φ)
 9 if DLL(φ, α ∪ [x ⟵ 0]) then
10 |  return true
11 else
12 |  return DLL(φ, α ∪ [x ⟵ 1])
13 end
```

Algorithm 1: A basic version of the DLL algorithm

When looking at the algorithm there are two major questions: Is it necessary to take decisions for all literals or are there literals which must have certain values, so that we can reduce the formula? The second question is how to choose the variable in line 8 of the algorithms. We will see that these two questions are a central point for deriving a fast algorithm.

Reductions

There are many approaches for reducing clauses. It obviously does not change the satisfiability of a formula if identical clauses are deleted. Identical clauses can often be found in randomly generated problems but also in real-world problems like circuit planning.

If a variable $a \in \mathcal{V}$ has a positive and negative occurrence in a single clause such as $\varphi := a \vee \neg a \vee B$ with B an arbitrary disjunction of literals, one can delete this clause. Because no matter if a is assigned to $\top$ or to $\bot$, φ will be $\top$ in each case. These two reductions can be performed during pre-processing.

The most crucial reduction is the so called *Unit Propagation*. If for an unsatisfied clause φ all literals but one are assigned, the value the last literal must be assigned to to turn the clause to $\top$ is obvious. This must be $\top$ for a positive literal and $\bot$ for a negative literal. We refer to such a clause as *unit clause*. The resulting process is referred to as *Unit Propagation* :

```
  Input: Set of clauses C, interpretation α
1 UP(C, α)
2 while there are unit clauses in C do
3 |  if sole unassigned literal a is positive then
4 |  |  α := α ∪ [a ⟵ ⊤]
5 |  else
6 |  |  α := α ∪ [a ⟵ ⊥]
7 |  end
8 end
```

Algorithm 2: The basic Unit Propagation algorithm

In the DLL procedure the Unit Propagation (UP) is called immediately after the recursive call of DLL before any assignments are done. Modern SAT solvers spend up to 90% of their time in Unit Propagation. In chapter 2 we will demonstrate how to optimise this process, which is the key to an efficient SAT solver.

Strategies for Variable Selection

Besides reductions there is another crucial aspect: The selection of the next branching variable. The question is how to select the next variable and to which value it should be assigned. Because of the basic backtracking algorithm the choice of the variable and its assignment are responsible for the number of backtracking steps required to solve the problem, which is demonstrated in the following example.

Example 1.1

$$\varphi := (a \vee b \vee c) \wedge (a \vee \neg c) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg b) \quad (1.3)$$

The search space for the solution of the problem is illustrated in figure 1.1.

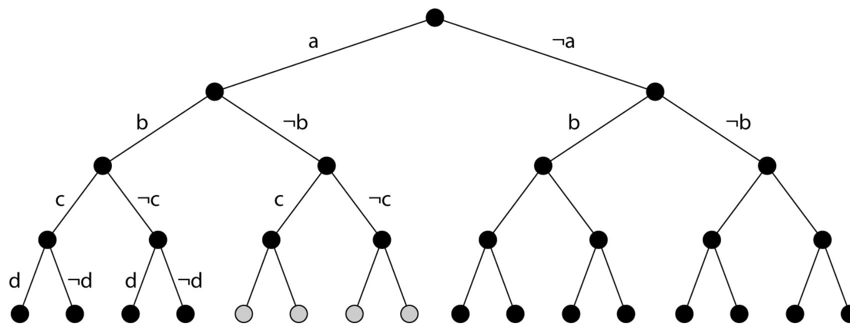

Figure 1.1: Search tree for a SAT problem

Satisfying assignments of the variables are indicated with a grey circle. It is obvious that $[a \longleftarrow \top, b \longleftarrow \bot]$ *must hold. If the heuristic selects* a *as first variable and assigns it to* $\top$*, we descend in the left sub tree. Now UP will cause that* b *is assigned to* $\bot$ *and we derive a satisfying solution in one step and with no backtracking. But if the strategy first selects variable* a *and assigns it to* $\bot$*, we descend into a sub tree where no satisfying solution is present. Therefore UP will cause assigning* b *and* c *to* $\bot$ *and as a result an empty clause emerges — an unsatisfied clause with no variable left to assign —* $a \vee b \vee c$*. For this reason a backtracking step is required.*

A bad variable selection can be critical in cases with hundreds of variables. Without a strategy for variable selection even examples with 50 variables require a large number of backtracking steps.

Because the decision for a variable is equal to a branch in the search tree, we also speak of *branching heuristics*. Variables which were selected from the heuristic are referred to as *decision variables*. Variables which are assigned because of Unit Propagation are referred to as *implied variables*. There are no provable “good” branching heuristics, but within the last twenty years many heuristics have been developed, which show good empirical results on real-world problems. A detailed overview of such heuristics is given in [Marques-Silva, 1999]. We will introduce certain branching heuristics in chapter 2.

Clause Learning

Until 1996 the DLL-based solvers had been the best SAT solvers. But then a new concept arised. One can observe that in a large SAT instance the algorithm runs in the same empty clauses again and again because pieces of information got lost.

Example 1.2 *If* $(\neg a \vee \neg b) \wedge (a \vee \neg c)$ *is part of a larger formula, one can derive that either* b *or* c *must be assigned to* $\perp$*. But this piece of information is not available for the algorithm. Therefore it could be useful to store the clause* $(\neg b \vee \neg c)$ *additionally to avoid unnecessary backtracking.*

These observations resulted in so called *conflict driven clause learning* (CDCL): Every time a conflict clause (empty clause) is derived, a new clause is learnt to avoid this conflict. This allows a non-chronological backtracking which means that the algorithm must not do backtracking step by step, level by level like in the DLL procedure, but can skip certain levels and thus prune large portions of the search tree. One of the first SAT solvers which introduced this new technique was GRASP [Silva & Sakallah, 1996].

Further Improvements

Since 1996 every state-of-the-art SAT solver utilises CDCL. But certainly many improvements have been made since then. One idea is to delete parts of the newly learnt clauses periodically in order to avoid a blow-up of the number of clauses. Another approach is to restart the SAT solver from time to time to avoid that early (and perhaps false) variable decisions have to much impact. A short overview of these current techniques can be found in [Mitchell, 2005].

1.3.2 Probabilistic Algorithms

Another completely different approach to algorithms is the probabilistic one. The main idea is to guess an initial assignment of all variables. This will result in some clauses being **true** and some clauses being **false**. Now a concept called *local search* is utilised: Variables from unsatisfied clauses are taken and flipped (from $\top$ to $\perp$ and vice versa) successively. If no satisfying assignment is found after $O(n)$ flips, the procedure is restarted with a new initial assignment.

One of the best provable algorithms following this paradigm is the one presented by Uwe Schöning in 1999 [Schoning, 1999]. The algorithm itself is very simple:

```
  Input: a formula φ in k-CNF with n variables
  Output: true or false
1 Guess an initial assignment α uniformly at random;
2 for 3n-times do
3 |  if (φ, α) = ⊤ then
4 |  |  return true
5 |  end
6 |  Let C be some clause not being satisfied by the current assignment;
7 |  Pick one of the < k literals in the clause at random and flip its value;
8 end
9 return false;
```

Algorithm 3: Local search of Schöning's probabilistic k-SAT algorithm

Schöning's main achievement was to prove that it is sufficient to repeat this local search t-times where t is within a polynomial factor of $O(2(1 - \frac{1}{k}))^n$. This means t is only dependent on the maximal clause-length k and the number of variables n. The following table shows the values of t for some values of k.

3-SAT	4-SAT	5-SAT	6-SAT
1.334	1.5	1.6	1.667

Table 1.1: Values for t in dependence of k

It is obvious that such an algorithm can only prove the satisfiability of a formula. If its output is **false** this could still be a false negative (Monte Carlo Algorithm).

A (very simple) implementation of Schöning's algorithm was tested within the REDLOG framework. But the results could not compete with those reached by the CDCL based SAT solver or even with the standard quantifier elimination. But there are three important reasons for such probabilistic SAT solvers:

1. With probabilistic algorithms the hardness of problems can be shown. If such a randomised algorithm can solve a problem within short time, it cannot be too hard.

2. For randomly generated input formulas probabilistic algorithms show a good behaviour.

3. Many professional SAT solvers are specialised in certain domains. A probabilistic solver is independent from any domain and can therefore be utilised for any sort of problem domain.

1.3.3 Conclusion: Algorithms in Current SAT-Solvers

Although probabilistic algorithms possess the best currently known upper bounds for the SAT problem, they are not favoured in current implementations of SAT solvers. The majority of successful solvers utilises conflict driven clause learning with some special branching heuristics. The three solvers, which inspired this work and the implementation of the SAT solver engine in REDLOG are GRASP [Silva & Sakallah, 1996], CHAFF [Moskewicz *et al.*, 2001] and MiniSat [En & Srensson, 2003].

The reason why probabilistic algorithms are not utilised in these efficient solvers is obvious. They have the best worst-case bounds, but this worst-case is reached rarely with the current algorithms. A probabilistic algorithm is independent of the structure of a problem. There is so much additional information which can be derived from a set of clauses (e.g. the learnt clauses) which probabilistic algorithms do not utilise within their computations.

To sum up one can state that there are three points which make an efficient SAT solver:

1. An empirical good branching heuristic

2. Clause learning

3. A fast algorithm for Unit Propagation

In the next chapter we will demonstrate how these three points are realised in REDLOG.

2 Implementing the SAT Algorithm

This chapter is concerned with the implementation of a SAT solver in REDLOG. At first there will be a short overview of REDLOG and how Boolean Algebra is realised. The necessary data structures will be presented in section 2.2. Then we will look on the already mentioned branching heuristics in more detail. Section 2.4 will describe how clause learning is implemented and how the conflict clauses are derived. After this section our SAT solver can handle formulas with about 200 variables. But to extend it to even larger formulas, a fast implementation of UP is required. A concept called *watched literals* will be introduced in section 2.5. Section 2.6 will show further improvements like restarts and clause deletion. In the last section a comparison to the standard quantifier elimination and to other SAT solvers will be realised.

2.1 Boolean Algebra in REDLOG

REDLOG is an abbreviation for "Reduce logic" system [Dolzmann & Sturm, 1997]. It is an additional package for the REDUCE computer algebra system and provides symbolic algorithms on First-Order formulas. Therefore a theory must be temporarily fixed in the first place. This is referred to as a *context*. In REDLOG there are contexts for ordered fields (OFSF), for algebraically closed fields (ACFSF), for discretely valued fields (DVFSF) or for Presburger Arithmetic (PASF). Since 2003 there is also a context for initial Boolean Algebras, refered to as IBALP [Seidl & Sturm, 2003]. Within this context the SAT solving is implemented.

2.1.1 Propositional Logic Within First-Order Logic

When we speak about SAT solving we speak about formulas in Propositional Logic. But as already mentioned, REDLOG is a system for First-Order Logic. So we have to link these two systems.

As described in Propositional Logic there are propositional variables from $\mathcal{V}$ combined with Boolean operators $\neg, \vee, \wedge$. A semantics for these formulas is obtained by interpreting all variables with $\top$ or $\bot$. In First-Order Logic there are atomic formulas over a a fixed language Σ instead of propositional variables. These atomic formulas can either be equations $t_1 = t_2$ whith t_1, t_2 being terms constructed from the function symbols in Σ or predicates $t_1 \ \varrho \ t_2$ with t_1, t_2 terms and ϱ a relation symbol of Σ.

For an interpretation of these atomic formulas we need to fix a Σ-structure $\mathbb{S}$ with a universe S. Now the extended atomic formulas can be interpreted at some point in S. From now on a truth value for the formula is derived in the same way as for Propositional Logic — by using the laws of Boolean algebras. In addition, First-Order Logic allows quantification $\forall x$, $\exists x$ for $x \in S$ which will lead us to Q-SAT in chapter 3 and parametric Q-SAT in chapter 4.

The Pure Algebraic Approach

[Seidl & Sturm, 2003] show three possibilities to implement Propositional Logic within First-Order Logic. We will only look at the one chosen for REDLOG. It is referred to as the *pure algebraic approach*. From an algebraic point of view a suitable language for Boolean algebras is required. The following language seems to be functioning:

$$\mathcal{L}_{\mathbb{B}} = \{true^{(0)}, false^{(0)}, not^{(1)}, and^{(2)}, or(2)\} \tag{2.1}$$

As demonstrated $\Longrightarrow$ and $\Longleftrightarrow$ can be expressed with function symbols and, or and not so we do not need these symbols. The set of terms $\mathcal{T}$ over this language can be defined inductively:

Definition 2.1 (Terms over the language for Boolean Algebras) *Every variable from an infinite set of variables $\mathcal{V}$ is a term, the constants* **true** *and* **false** *are terms and for terms t_1 and t_2 also $(not\ t_1)$ and $(t_1\ \varrho\ t_2)$ with $\varrho \in \{and, or\}$ are terms.*

We can now fix an $\mathcal{L}_{\mathbb{B}}$-Structure $\mathbb{B}$ for the Boolean Algebra:

$$\mathbb{B} = (\{\top, \bot\}; \top, \bot, \neg, \wedge, \vee) \tag{2.2}$$

Yet we are able to explain the concept of satisfiability in terms of our new structure. Since we have no relation symbols in our language, atomic formulas can only be equations.

Definition 2.2 (Satisfiability of terms over the language for Boolean Algebras) *An extended equation $(t_1 = t_2)(x_1, ..., x_n)$ with t_1, t_2 terms over $\mathcal{L}_{\mathbb{B}}$ is satisfiable if $b_1, ..., b_n \in \{\top, \bot\}$ with $\mathbb{B} \models (t_1 = t_2)(b_1, ..., b_n)$ exists. The equation is valid, if $\mathbb{B} \models (t_1 = t_2)$ holds for all $b_1, ..., b_n \in \{\top, \bot, \}$.*

The Common Algebraic Approach

In general one does not want to fix the structure $\mathbb{B}$ but to speak generally of *truth values* and *truth assignments* of variables which extend to truth assignments of terms. In this case we speak of *Boolean expressions* instead of terms.

Definition 2.3 (Satisfiability of Boolean expressions) *A Boolean expression $t(x_1, ..., x_n)$ is satisfiable if a truth assignment to $x_1, ..., x_n$ exists so that t is* **true**. *In terms of the pure algebraic approach this means: $t = \top$ is satisfiable. $t(x_1, ..., x_n)$ is valid if it is* **true** *for all truth assignments of $x_1, ..., x_n$. This corresponds that $t = \top$ is valid in the pure algebraic approach.*

It is important to note that the common algebraic approach and the pure algebraic approach can be straightforwardly translated into each other and that the common algebraic approach does not introduce any syntactic object or notion which is not existent in the pure algebraic approach.

In REDLOG the common algebraic approach is implemented as a *propositional wrapper*. The wrapper allows to write the equations of propositional formulas as capital propositional variables.

Example 2.1 *If we want to code the formula $\varphi := (a \vee b \vee c) \wedge (a \vee \neg b) \wedge (\neg a \vee \neg c)$ in REDLOG we can either write*

```
(a=1 or b=1 or c=1) and (a=1 or not(b=1)) and (not(a=1) or not(c=1))
```

or with the help of the propositional wrapper

```
(A or B or C) and (A or not B) and (not A or not C)
```

2.1.2 REDLOG Commands

This is just a small overview of some commands required to work with the IBALP context of REDLOG. For further details look at [Dolzmann & Sturm, 1999].

Command	Short Description
`load redlog`	load the REDLOG package
`rlset ibalp`	set the context to ibalp
`rlqsat`	the SAT solver presented in this work
`rlqe`	the standard quantifier elimination
`rlqea`	the extended quantifier elimination
`rlcnf`	conjunctive normal form
`rldnf`	disjunctive normal form
`rlall`	universal closure
`rlex`	existential closure
`rlatnum`	number of atomic formulas
`rlqnum`	number of quantifiers

Table 2.1: REDLOG commands

Example 2.2 *Finally, we want to demonstrate a small SAT solving example*

```
REDUCE Development Version, 15-May-2007 ...

1: load redlog;

2: rlset ibalp;

*** turned off switch raise

3: f := (A or B or C) and (A or not C) and (A or not B) and (not A or not B);

f := (a = 1 or b = 1 or c = 1) and (a = 1 or not(c = 1))
 and (a = 1 or not(b = 1)) and (not(a = 1) or not(b = 1))

4: rlqsat f;

true

5: f := (A or B) and (not A or B) and (A or not B) and (not A or not B);

f := (a = 1 or b = 1) and (not(a = 1) or b = 1) and (a = 1 or not(b = 1))
 and (not(a = 1) or not(b = 1))

6: rlqsat f;

false
```

2.2 The Data Structure for our SAT Solver

While solving SAT instances an updated list of all variables and of all clauses - satisfied or unsatisfied - is required at any time in the program. To derive a fast and efficient implementation we need two data structures: one for variables and one for clauses.

2.2.1 The Variables

In our implementation a variable is a list consisting of entries, shown in table 2.2.

Position	Abbreviation	Description
#1	`id`	the identifier of the variable
#2	`val`	the current value of the variable
#3	`posocc`	a list of clauses where the variable occurs positive
#4	`negocc`	a list of clauses where the variable occurs negative
#5	`numpos`	the number of currently unsatisfied clauses where the variable occurs positive
#6	`numneg`	the number of currently unsatisfied clauses where the variable occurs negative
#7	`lev`	the level at which the variable was set
#8	`reas`	the reason why the variable was set
#9	`poscc`	the number of positive occurrences in newly added conflict clauses
#10	`negcc`	the number of negative occurrences in newly added conflict clauses
#11	`wc`	a list of watched clauses (for UP)
#12	`mom`	the MOM-value (for branching heuristic)

Table 2.2: The fields of a variable

`id` is a unique identifier for a variable. The value `val` of a variable can be 0 or 1, representing $\top$ and $\bot$ or `nil` if the variable is currently not assigned. The clause lists `posocc` and `negocc` are necessary, because when a variable is assigned to a certain value all clauses in which the variable occurs must be processed. This is compulsory for a fast implementation. The numbers `numpos` and `numneg` of currently unsatisfied clauses with the variable are an important factor in branching heuristics. Each variable is set at a certain level — increasing with descending into the search tree and decreasing with backtracking. This level is stored for each variable in `lev`. If a variable is set during Unit Propagation (implied variable), the corresponding unit clause is stored as the reason for this assignment. If a variable is set because of a branch (decision variable), `nil` is the reason. For extended branching heuristics we need to know in how many new learnt clauses a variable occurs, stored in `poscc` and `negcc`. For a fast Unit Propagation we will introduce a technique called *watched literals* and therefore a list of watched clauses `wl` is required. Finally, during the implementation it became apparent to be more efficient to store the MOM-value `mom` also within the data structure of the variable rather than computing it for each branching step.

Although we risk causing a large overhead in processing the variables, benchmarks proved that the additional information has a larger benefit than overhead.

2.2.2 The Clauses

As for variables we need some information stored in the clauses. An overview is given in table 2.3.

Position	Abbreviation	Description
#1	`poslit`	a list of the positive literals of this clause
#2	`neglit`	a list of the negative literals of this clause
#3	`actpos`	the number of currently unassigned positive literals in that clause
#4	`actneg`	the number of currently unassigned negative literals in that clause
#5	`sat`	the list of variables turning this clause to $\top$
#6	`count`	a counter for newly added clauses
#7	`wl`	a list of watched literals for this clause

Table 2.3: The fields of a clause

A clause consists of positive and negative literals, so these are stored in `poslit` and `neglit`. For a fast check whether a clause is a unit clause (unsatisfied and just one unassigned literal) or empty (unsatisfied and no unassigned literal) we need to store the currently unassigned literals in `actpos` and `actneg`. A single clause can be turned to $\top$ from more than one variable, therefore a list with all variables turning a clause to $\top$ is held in `sat`. If the clause is unsatisfied, `nil` is stored. If we add newly learnt clauses, we perhaps want to delete these clauses from time to time. For this reason we introduce a counter `count` indicating how important a new clause is. Finally, as already mentioned, for fast UP we need watched literals. These are stored in `wl`.

2.2.3 Implementation of the Lists

A first implementation of the data structure for variables and clauses was realised with vectors, but it became apparent that vectors are significantly slower (about 30%) than lists and thus an implementation with lists and constructive operators was chosen. The format of the data structures for variables and clauses is hidden by constructors and access functions. This means that during programming no direct call of any list operations is necessary. For an empirical improvement of 10% the access functions were implemented as `smacros` which are replaced within the code at compile time.

Example 2.3 *The constructor for a new clause looks like this:*

```
smacro procedure ibalp_clause!-new();
   {nil,nil,0,0,nil,nil,nil};
```

Also the access functions for `actpos` *are given as an example:*

```
smacro procedure ibalp_clause!-setactpos(clause,actpos);
   caddr clause := actpos;

smacro procedure ibalp_clause!-getactpos(clause);
   caddr clause;
```

While reading an input formula the list of clauses and the list of variables are constructed.

Remark *These two data structures — variables and clauses — are cyclic. Variables store the clauses where they occur, clauses store their positive and negative variables. A first implementation used A-Lists (associative lists) instead of this cyclic data structure in order to only store the identifiers of variables within the clauses. But this caused a large number of pointer comparisons. So introducing the cyclic data structure leads to a speedup of factor 5.*

2.2.4 Setting and Unsetting Variables

When a variable is set because of a branch or a Unit Propagation several values need to be set and updated. Let us first look at the pseudo-code for setting a variable to $\top$.

Input: *var*, the variable to set *var*; *level*, the level of the assign;*reas*, the reason for the assign
1 Set value of *var* to $\top$;
2 Set level of *var* to *level*;
3 Set reason of *var* to *reas*;
4 **foreach** *clause C in* ***posocc(****var****)*** **do**
5 | **if** *C is not satisfied yet* **then**
6 | | decrement counter **numpos**(v) or **numneg**(v) for every variable v in C;
7 | **end**
8 | add *var* to **sat**(C);
9 **end**
10 **foreach** *clause C in* ***negocc(****var****)*** **do**
11 | decrease counter **actneg**(C)
12 **end**

Algorithm 4: Setting a variable to $\top$

In lines 1–3 the given parameters `val`, `level` and `reas` can be set directly without any further restrictions. When the variable is assigned to $\top$, every clause C in which the variable occurs positive is satisfied from this variable *var* now. So we add this *var* to the list of satisfying variables `sat` in line 8. In line 5–7 we check if the clause C was satisfied before this assignment. If so, no further action is required, if not then we must decrement the counter `numpos` or `numneg` for every variable in C. This is because this counter indicates in how many *unsatisfied* clauses the variable occurs. So when changing a clause to *satisfied* we must decrement it. Finally, in lines 10–12 we look at all clauses in which *var* occurs negative. These clauses will not be satisfied if *var* is assigned to $\top$. Thus we will decrease the counter `actneg`, meaning that the number of unassigned variables, having the possibility to turn this clause to **true**, is decreased by one.

If a variable is assigned to $\bot$, is is required to call the dual operation where `posocc` and `negocc` are changed and instead of `actneg` we decrease `actpos`. Certainly the final procedure must perform more updates, e.g. updating the values for the MOM heuristic or searching new watched literals, but this will be described in the corresponding sections.

If a variable *var* is unset, is is required to know to which value the variable was assigned

before. The following code unassignings a variable from $\top$ to `nil`.

```
    Input: var, the variable to unassign var
 1  Set value of var to nil;
 2  Set level of var to -1;
 3  Set reason of var to nil;
 4  foreach clause C in negocc(var) do
 5  |   increase counter actneg(C)
 6  end
 7  foreach clause C in posocc(var) do
 8  |   delete var from sat(C);
 9  |   if C now is not satisfied then
10  |   |   increment counter numpos(v) and numneg(v) for every variable v in C;
11  |   end
12  end
```

Algorithm 5: Unsetting a variable from $\top$

As one can easily see, all the assigns which were done while setting the variable must be reversed. Lines 1–3 reset the values `val`, `lev` and `reas`. Then the counter `actneg` for each clause C where the variable occurs negative is increased. As a result the variable is unassigned and is a potential candidate to turn the clause C to true (lines 4–6). Then we go into the list of clauses C where the variable has positive occurence. These are the clauses which were turned to **true** from this variable. At first the variable is deleted from the list `sat` of variables turning the clause C **true** (line 8). If C is unsatisfied now the counters `numpos` or `numneg` for each literal of C are increased (lines 9–11) because they have one occurrence more in an unsatisfied clause now.

We can see that setting and unsetting of variables is a procedure which is time-consuming in the SAT solving process. And it is obvious that a large number of backtracking steps is equal to a large number of variable unsets and sets. For this reason we try to minimise the number of backtracking steps with the help of good branching heuristics.

2.3 Branching Heuristics

We will now look at some different branching heuristics : DLIS and DLCS as literal counting heuristics, MOM's heuristic and activity heuristics. In section 2.3.4 we will compare the heuristics using examples from the DIMACS suite. Besides the heuristics given here there are many other heuristics. [Marques-Silva, 1999] gives a detailed overview.

2.3.1 Literal Count Heuristics

There is a section of heuristics only based on counting the occurrences of the variables in unsatisfied clauses. Therefore we stored the values `numpos` and `numneg`. One approach is to select variables with a large number of occurrences at first. This can have a largerer impact to the backtracking process than selecting variables with only one or two occurrences. Let us look at two different heuristics: DLIS and DLCS.

DLCS (Dynamic Largest Combined Sum)

For each variable we stored the counter `numpos` for positive occurrences of the variable in unsatisfied clauses and `numneg` for negative occurrences. In DLCS heuristic the variable with the largest sum of $numpos{+}numneg$ is selected. The value is $\top$ if $numpos > numneg$

and $\bot$ else. Since for each variable just the sum of positive and negative occurrences are considered and these values are computed dynamically during the search we speak of the *dynamic largest combined sum*.

DLIS (Dynamic Largest Individual Sum)

As for DLCS, DLIS only depends on `numpos` and `numneg`. But here not the sum but the individual count is relevant. So the variable with the largest `numpos` or `numneg` is chosen. If the largest value is from a `numpos`, we assign the value to $\top$. If it is from a `numneg`, we assign it to $\bot$. Because we now look at the individual count of positive and negative occurrences we speak of the *dynamic largest individual sum*.

Randomised Versions

For both strategies exist randomised versions, referred to as RDLCS and RDLIS. The only difference is that the value the variable is assigned to is computed randomly.

2.3.2 MOM's Heuristic

MOM is an abbreviation for **M**aximal **O**ccurrence in clauses of **M**inimal size. So the chosen variable has a maximum number of occurrences in minimal size clauses. Minimal size means the minimal number of unassigned literals. By selecting these variables we achieve a maximum number of Unit Propagations in the next step. How is MOM computed? In the first place we have to find out the minimal size of an unsatisfied clause l. This value must be ≥ 2 because otherwise it would have been a unit clause in the last step.
Let $numpos_l$ and $numneg_l$ denote the number of positive/negative occurrences in clauses of minimal size l. In the next step MOM chooses the variable with the maximum value for

$$(numpos_l + numneg_l) \cdot 2^k + (numpos_l \cdot numneg_l) \tag{2.3}$$

with k a value such that the first term dominates the second. This should simulate a lexicographic order on the pair $(numpos_l + numneg_l, numpos_l \cdot numneg_l)$. At first variables with the largest number of occurrences in clauses of minimal size are chosen, so maximal $(numpos_l + numneg_l)$ is preferred. From all these variables with the same maximal $(numpos_l + numneg_l)$ the one with the smallest difference between $numneg_l$ and $numpos_l$ is selected, meaning maximal $numpos_l \cdot numneg_l$ (The square is the rectangle with the largest area).

In the REDLOG SAT solver a variation of MOM's heuristic is utilised as default heuristic, called ZMOM. The ZMOM value is computed from every clause not only from clauses of minimal size. The selected variable must occur in a clause of minimal size. So it is guaranteed that (in most cases) a Unit Propagation will follow. And the computation of this new ZMOM value can be done more efficiently than the original one. The current ZMOM value is stored separately for each variable — as seen in section 2.2.1 — and updated every time the variable's settings are changed. In empirical studies this has appeared to be a quite good heuristic.

2.3.3 Activity Heuristics

When we utilise conflict driven clause learning we have additional information to use for a branching heuristic. How active is a variable? The activity of a variable is measured by its occurrences in newly generated and learnt clauses (see section 2.4). The number

of positive and negative occurrences of a variable (poscc and negcc) is stored for each variable. Every time a variable is used in a new conflict clause, this value is increased by 1. Like in MiniSat poscc and negcc are decreased after every learnt clause by 0.05 [En & Srensson, 2003]. This leads to the effect that the heuristic selects variables with a high activity (occurrences in many conflicts) and prefers those with occurrences in recent conflicts rather than old conflicts.

2.3.4 Comparison of the Heuristics

The first benchmark was realised with a simple DLL implementation, without features such as clause learning or watched literals. The only aim was to test the quality of the heuristics. Certainly, the activity heuristics could not be tested because of a lack of learnt clauses. The test benchmarks were 4 classes of the DIMACS suite, namely aim-50-1_6-no, aim-50-1_6-yes1, aim-50-2_0-no and aim-50-2_0-yes. Each of this classes has 4 instances. All these instances have 50 variables. The results can be seen in table 2.4.

Class	# vars	# cl	SAT?	# ZMOM	# MOM	# DLIS	# DLCS
aim-50-1_6-no	50	80	no	**4399**	58041	13481	4482
aim-50-2_0-no	50	100	no	**8218**	11957	28021	12382
aim-50-1_6-yes1	50	80	yes	**508**	6167	5826	1100
aim-50-2_0-yes1	50	100	yes	1229	**1142**	3057	2408

Table 2.4: Benchmark 1 for testing the branching heuristics

vars and # *cl* are the numbers of variables and clauses. *sat* indicates whether the formula is satisfiable or not. In the following columns the numbers of backtracking steps are noticed. As one can easily see the newly developed ZMOM appears to be the best heuristic for the first three classes. In the fourth class the difference to the best heuristic is only approximately 10%.

In the next step the heuristics were tested with an improved SAT solving engine and larger examples. A SAT solver with clause learning, non chronological backtracking and restarts evolved. We will examine this later later in this chapter. For now we can evaluate the activity heuristic, too.

The benchmark includes four classes from the DIMACS suite. The complete aim-200 class with 24 instances, the ii8 class with 14 instances and the ii16 class with 10 instances. The ii8 and aim200 class appeared to be very easy to solve but in ii16 there are some hard instances. An overview of the used CPU time in seconds is given in table 2.5. It is not easy to decide which heuristic performs best, but because of the better performance of ZMOM with larger formulas, we decided to set ZMOM as default heuristic. But the activity heuristic is often worth a try.

There is one interesting remark: The standard quantifier elimination procedure (QE) for the ibalp context in REDLOG was not able to solve the first instance of the ii16 class, ii16a1 with 1650 variables and 19368 clauses within six hours [Seidl & Sturm, 2003]. The newly implemented SAT solver yields the result after 10 seconds.

2.4 Conflict Driven Clause Learning

The biggest impact on the implemented SAT solver was derived by introducing a concept called clause learning. This was first realised by Silva and Sakallah in 1996 [Silva &

Class	# vars	# cl	SAT?	ZMOM	Act	DLCS
aim-200-1_6-no	200	320	no	**0.11**	0.27	**0.11**
aim-200-2_0-no	200	400	no	**0.11**	0.5	0.14
aim-200-1_6-yes1	200	320	yes	16.26	**0.83**	14.78
aim-200-2_0-yes1	200	400	yes	**1.49**	6.6	2.04
aim-200-3_4-yes1	200	680	yes	19.08	37.46	**14.03**
aim-200-6_0-yes1	200	1200	yes	6.81	**5.89**	18.73
ii8	66–1068	186–8214	yes	**8.54**	20.25	12.16
ii16	532–1728	7825–24792	yes	**187.65**	657.56	264.54

Table 2.5: CPU time in seconds for large examples

Sakallah, 1996]. This is referred to as *conflict driven clause learning* (CDCL*)* because new clauses are derived from conflict cases. To show how this concept works we will look at the following example.

Example 2.4 *Consider the CNF-Formula* $\varphi := \bigwedge_{i=0}^{n} \psi_i \wedge (x \vee y \vee z) \wedge (\neg x \vee \neg y) \wedge (\neg x \vee z) \wedge (y \vee \neg z)$ *with further clauses* ψ_i*. We assume that the branching heuristic chose* $a_1, ..., a_m$ *as branching variables before it selects* x*. If* x *is now set to* $\top$*,* y *must be assigned to* $\bot$ *(because of* $(\neg x \vee \neg y)$*) and* z *must be assigned to* $\bot$ *(because of* $(\neg x \vee z)$*). But as a consequence we derive an empty clause* $\epsilon = (y \vee \neg z)$ *which cannot be satisfied anymore. So we perform a backtrack step and set* x *to* $\bot$ *and for example* y *to* $\top$*. As a result the four given clauses are satisfied.*
From now on we know that in every case x *must be assigned to* $\bot$ *but we "forget" this piece of information. If we backtrack to one of the variables* $a_1, ..., a_m$ *because of another empty clause in further processing of the formula, we will run into the same empty clause* ϵ *again. Figure 2.1 illustrates the example.*

Figure 2.1: Situation without clause learning

The idea of clause learning is to store information like "x must be set to True" by adding new clauses to the set of clauses. We will first rewrite the DLL algorithm from chapter 1 in an iterative version. Then we will introduce the semantics of resolution, finding the conflict clause and non-chronological backtracking. At last we prove the correctness and the termination of the algorithm, which is — in contrast to the DLL algorithm — not obvious.

2.4.1 The Iterative Version of the DLL Algorithm

Because in CDCL we will not backtrack from level to level like in DLL but non-chronological, meaning we can skip levels while backtracking, we will use an iterative version of the DLL algorithm.

```
   Input: Formula f in Propositional Logic
   Output: true or false
 1 level := 0;
 2 α := ∅;
 3 while true do
 4   UP(f, α);
 5   if a conflict is reached then
 6     ec := empty clause;
 7     level := analyseConflict(ec, C (the set of clauses));
 8     if level = 0 then
 9       return false
10     end
11     backtrack(level);
12   else
13     if formula is satisfied then
14       return true
15     end
16     level := level + 1;
17     choose an unassigned x ∈ V(F);
18     α := α ∪ [x ⟵ 0];
19   end
20 end
```

Algorithm 6: The CDCL algorithm

Because of non-chronological backtracking we store the decision level for each variable. We begin at level 0 (line 1) and increment the level before a new variable is set (line 16). All variables assigned through Unit Propagation (line 4) are at the same level as the antecedent decision variable. If a conflict is reached, the corresponding empty clause (line 6) is stored. The main procedure of this section is called in line 7: The conflict analysis. Within this analysis a new clause is learnt and the backtrack level is computed. If the backtrack level is 0, we know that f must be unsatisfiable because at level 0 no variable was set and only Unit Propagation was performed. If a conflict arises at this level, the formula is unsatisfiable and the return value is **false** (line 9). If *level* > 0, we backtrack to this level, meaning we unassign all variables with decision level larger than the backtrack level. If no conflict is reached, we check if the formula is already satisfied. If that is the case, we return **true**, if not, the level is incremented and a new variable is selected and assigned to a certain value.

Remark *If we use a naive implementation of the* analyseConflict *procedure, we achieve an iterative version of the standard* DLL *procedure. In order to do so we set a flag "flip" to* **false** *if a variable is set the first time. If the value of a variables flips because of backtracking, we set the flag to* **true**. *So by choosing the level at which the last unflipped variable is found, we get the original chronological backtracking such as in* DLL.

A proof of correctness and termination of the algorithm will follow in section 2.4.3 after the necessary concepts for conflict learning will have been introduced.

2.4.2 The Semantics of Clause Learning

In this section we will look at the procedure analyseConflict, which computes a new clause out of an empty clause and the new level to which we must backtrack. It is obvious that the new clause must not change the satisfiability of the original formula, which means it must be redundant with respect to the original clauses. To reach this aim we must iteratively resolve two clauses, referred to as *resolution.*

Definition 2.4 (Resolution) *The resolution of two clauses $\varphi := A \vee p$ and $\psi := B \vee \neg p$ in a CNF with A, B disjunctions of other literals x_i is defined as $\varphi \oplus \psi = A \vee B$ and is referred to as resolving on x.*

It is obvious that resolution does not change the satisfiability of the formula.

As already mentioned we store a reason for each variable. If a variable was set because of Unit Propagation, the reason is the corresponding unit clause. For a decision variable the reason is nil. We can now state the code for the procedure:

```
   Input: an empty clause ec, the set of clauses C
   Output: The backtrack level
 1 lv := the last assigned variable before the conflict clause;
 2 reas := the reason of lv;
 3 newclause := resolve(ec,reas);
 4 while the stop criterion for newclause is not met do
 5 |   cv := chooseLiteral(newclause);
 6 |   reas := the reason of cv;
 7 |   newclause := resolve(newclause,reas);
 8 end
 9 C := C ∪ {newclause};
10 level := computeBTLevel(newclause);
11 return level;
```

Algorithm 7: The analyseConflict procedure

We require the variable lv, which was assigned last before the empty case has arised (line 1). In our implementation this variable is passed from the Unit Propagation procedure. In the next step the reason $reas$ for the assignment of this variable is required (line 2). The empty clause must be of the form $ec := p \vee q \vee A$ with A a disjunction of further literals. p and q must both have been assigned at the conflict level, since otherwise ec would have been a unit clause at the previous decision level. Let p be the variable assigned last. Since ec is an empty clause, p must be assigned to $\bot$, thus the reason for p must be of the form $reas := \neg p \vee r \vee B$ with B a disjunction of further literals (r and q can be equal). So the first derived new clause after resolution is $newclause_1 := q \vee r \vee A \vee B$ (line 3). We see that p does not occur in the new clause, so we can state that we resolved p out.

In line 4–8 we repeat resolution until a certain stop criterion is met. This stop criterion is met when in the new clause only one literal is at the current decision level. To understand this we must examine the whole CDCL algorithm. After the new learnt clause was added to the set of clauses we perform a backtrack to a certain level. The idea is now, that the new learnt clause is unit after this backtrack and so the sole unassigned variable will be assigned. We also know which level we must backtrack to: it is the highest level of a variable in the new learnt clause which is really lower than the conflict level. So all variables with a higher decision level than this backtrack level are unassigned — and that is the only variable at conflict level in the new clause. For this reason the new clause has exact one unassigned variable after the backtrack and therefore gets unit.

If this stop criterion is not met, a new variable of the current new clause is chosen. The best strategy to do so is to choose a variable at the highest level. Certainly only implied variables (the ones set through UP) are chosen, since decision variables have no reason. This process must terminate but in the worst case all variables but the only decision variable at the current level are resolved out. After we will finally have derived the new learnt clause, we will add it to the set of clauses (line 9) and compute the backtrack level as shown above and return it (line 10).

This strategy for deriving the new clause is referred to as *first unique intersection point (FUIP)*. This proved to be a very efficient strategy. [Beame *et al.*, 2004] give an overview of other strategies.

Example 2.5 *Consider the formula*

$$\varphi := (u \vee \neg w) \wedge (\neg u \vee \neg z) \wedge (w \vee \neg y) \wedge (w \vee \neg x) \wedge (x \vee y \vee z)$$

Assume that in the first step we assign variable u to $\top$ then we have a UP because of $(\neg u \vee \neg z)$ and assign z to $\bot$. In the second step w is assigned to $\bot$ so y and x are assigned to $\bot$ because of Unit Propagation and we derive an empty clause with $(x \vee y \vee z)$. The situation is illustrated in figure 2.2. In the following we compute a new clause with the

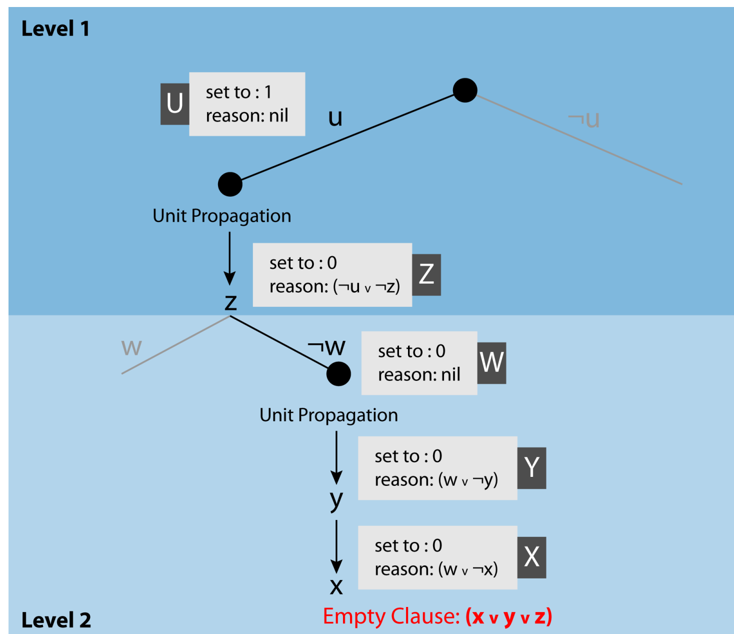

Figure 2.2: Example: Deriving the empty clause

analyseConflict *procedure. The variable last assigned is x so we resolve the empty clause and the reason for x. The numbers with each variable denote the level at which it was assigned.*

$$(x_{(2)} \vee y_{(2)} \vee z_{(1)}) \oplus (w_{(2)} \vee \neg x_{(2)}) = (y_{(2)} \vee z_{(1)} \vee w_{(2)}) \tag{2.4}$$

The stop criterion is not yet met because there are still two variables w and y at the current decision level. So we do another resolution with variable y and therefore the reason for y:

$$(y_{(2)} \vee z_{(1)} \vee w_{(2)}) \oplus (w_{(2)} \vee \neg y_{(2)}) = (w_{(2)} \vee z_{(1)}) \tag{2.5}$$

Now the stop criterion is met. We add the new clause $(w \vee z)$ to our set of clauses and we can backtrack to the highest level which is smaller than the current decision level which is level 1 (variable z).

After the backtrack, the variables w, x and y are unassigned and we are back at level 1. But because of the new learnt clause $(w \vee z)$ this results in another UP now: w is assigned to $\top$. Now x or y can be assigned to $\top$ and we derive a satisfying assignment for φ. The new situation is shown in figure 2.3.

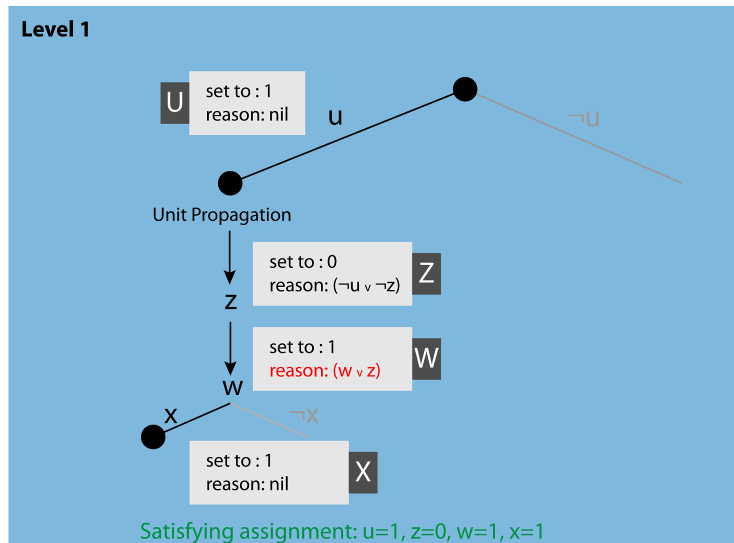

Figure 2.3: Example: The new situation after backtracking

2.4.3 Correctness and Termination of CDCL

In this section we will prove the correctness and the termination of the CDCL algorithm. This proof can be found in [Zhang & Malik, 2003b].

Correctness

Proposition 2.1 (Correctness of output true) *If the* CDCL *procedure returns* **true**, *the formula is satisfiable.*

Proof The CDCL procedure returns only **true** if a satisfying assignment was found. Since the new learnt clauses do not change the satisfiability of the original formula this must be a satisfying assignment for the original formula, which is then satisfiable. ■

Proposition 2.2 (Correctness of output false) *If the* CDCL *procedure returns* **false**, *the formula is unsatisfiable.*

PROOF The CDCL procedure returns **false** if it had to backtrack to level 0 after a conflict. This means one of the variables assigned at level 0 has to be flipped. But there was no branch at level 0, only Unit Propagation with clauses of the original formula and so only assignments that are implied by the original formula. So there must be a conflict in the original formula and it is not satisfiable. ∎

We have shown that the output of the CDCL is always correct, which means we have partially correctness.

Termination

For the total correctness we have to prove the termination of the algorithm.

Proposition 2.3 (Termination of CDCL) *Given enough runtime, the* CDCL *procedure will always terminate.*

PROOF Let n be the number of variables of a CNF input formula. At each level l there are $k(l)$ variables assigned. For all $l,\ 0 < l \leq n$ holds that $0 < k(l) \leq n$ because at each level (except level 0) at least one variable is assigned. It also holds that $\sum_l k(l) \leq n$. Consider function f:

$$f = \sum_{l=1}^{n} \frac{k(l)}{(n+1)^l} \tag{2.6}$$

Function f has a useful property: for two different assignments α and β $f_\alpha > f_\beta$ can hold if and only if a level l' with $0 \leq l' < n$ exists where $k_\alpha(l') > k_\alpha(l')$ and at all levels $l'' < l'$ it holds that $k_\alpha(l'') = k_\beta(l'')$. One can state that the function biases the sum towards the number of variables assigned at lower levels. During the search process the value of f monotonically increases. This is obvious if there is no conflict because in this case no variables are unassigned. But it also increases in conflict — and so backtrack — cases. As a consequence all assignments back to a certain level are undone. But at this level we have at least one more assignment because of Unit Propagation of the new learnt clause. This is level l' and compared to its value before the conflict analysis the value of f has also increased. Function f can only take a finite number of different values for a given SAT instance, which is why the algorithm must terminate. ∎

2.5 UP and Watched Literals

A state-of-the-art SAT Solver spends more then 90% of the time in the UP process. So the key to an efficient SAT Solver is a fast Unit Propagation procedure. A naive way would be to search in the set of clauses if there are clauses with only one literal unassigned. But it is obvious that this is not very efficient. One has to search the whole set of clauses because a Unit Propagation can cause many other UPs in every step. In the SAT Solver Chaff there is another way to handle UP, called *watched literals* [Moskewicz *et al.*, 2001].

The main idea is, that we do not have not look at every clause in every step of UP. A clause with three or more unassigned literals is not as “interesting” as one with only two unassigned literals which is very likely to get unit soon. So we introduce the concept of two watched literals. This means within each clause we store two unassigned literals. As long as there are two of them, the clause cannot get unit. If we set a variable, we need

to find a new watched literal in every clause this variable was a watched literal - because now it is not further unassigned. If we do not find a new unassigned literal, the clause is unit now and the other watched literal is the unit variable.

We must change our implementation of setting and unsetting a variable and therefore get the new unit clauses at the end of each assignment and do not need to search all the clauses. This causes a big speedup. We will illustrate this procedure with an example.

Example 2.6 *Consider the formula*

$$\varphi := (a \vee b \vee \neg c \vee d) \wedge (a \vee \neg b \vee \neg c)$$

Let the watched literals for both clauses be a and b. If the assignment $a \longleftarrow \bot$ is performed, neither the first nor the second clause is satisfied. But for both clauses a new watched literal has to be searched. In both cases we choose c, so that the new watched literals are now b and c for both clauses. In the next step c is assigned to $\top$. The first clause is still unsatisfied and we choose the new watched literal d, so now b and d are the watched literals. But in the second clause there are no more unassigned variables. So we know that the clause is now unit and b is the unit variable. Since b occurs negative in the second clause it must be set to $\bot$. With the assignment of b a new watched literal for the first clause has to be searched. But there is no further unassigned variable. This means that also the first clause got unit now and the unit variable is d which must be assigned to $\top$.

2.6 Further Improvements

The current SAT Solvers like Satz, Chaff or MiniSat use many other improvements. A few of them are implemented in the REDLOG SAT Solver.

2.6.1 Pre-Processing

When the original formula is read, two little simplifications are made:

1. Identical clauses are deleted. It makes no sense to have identical clauses in the clause set.

2. Tautology clauses are deleted. If a clause is of the form $(a \vee \neg a \vee A)$ with A a disjunction of further literals, this clause is **true** for every choice of a. Thus this clause is a tautology and it does not change the satisfiability of the formula when it is deleted.

   A second step in pre-processing is, to eliminate *pure literals*. Pure literals are variables, which occur either positive or negative. If a variable occurs only positive, it can be assigned to $\top$ and all clauses where the variable has a positive occurrence can be deleted. Then the variable can be deleted. If the variable occurs only negative, the same can be done with the dual strategy.

2.6.2 Simplifications

During the CDCL procedure an important simplification step is performed: If there are clauses with only one literal, this literal can be assigned to $\top$ or $\bot$ depending on whether the literal is positive or negative in this clause. Because there is no other choice for this

literal, we can delete it at once and save its assignment. For this reason every clause which is satisfied by this variable assignment can be deleted too. In every clause in which this literal occurs and which is not satisfied, this literal is deleted because its assignment is firm and can not be changed to satisfy this clause. This step is performed for single literal clauses in the original formula and those which are computed during the learning process.

2.6.3 Restarts

Another important concept is that of restarts. Empirical studies showed that often an early decision can cause a large number of backtracks and so waste a lot of time. To reduce the impact of such decisions, many of the current SAT solvers restart after a certain number of learnt clauses. *Restart* means that all variables are unassigned but learnt clauses and heuristic information such as the activity of variables are kept. However one must consider the following problem: With restarts the solver can lose the property of termination since it can not be guaranteed that it solves the problem within one restart-phase now. To avoid this problem, the number of needed learnt clauses for a restart increases with time. So the property of termination does not get lost.

As a further improvement the value a variable is assigned to is changed after every restart. So in the first run the variables are set to $\top$ per default, in the second to $\bot$, and so on.

2.6.4 Clause Deletion

In large examples the number of learnt clauses is often much higher than the original number of clauses. To avoid an overproportional blow up of clauses they are deleted periodically. Since it is not useful to delete arbitrary clauses we, look at the activity of new learnt clauses. Just as the activity heuristic for the branching variables, the activity of a clause is increased when it is used during a resolution step in the conflict analysis. From time to time the activity values of all clauses are decreased so that clauses with older conflicts are preferred to be deleted. After a restart we delete a certain number of old and not useful clauses.

2.7 Comparison to QE and to SAT Solvers

2.7.1 The Benchmarks

Since the SAT solver in REDLOG is not as powerful as current solvers like Satz, MiniSat or Chaff which are devoloped for years and are implemented very efficiently in C, we used smaller benchmarks up to 20000 clauses and 4000 variables. But this is far beyond the limits of the standard QE in REDLOG. But it is important to mention that the worst-case runtime of the SAT solver is still in $O(2^n)$. But the following benchmarks will demonstrate that these worst-cases are not reached with real-world problems.

AIM Benchmarks

The AIM instances are all generated with a particular Random-3-SAT instance generator [Y. Asahiro & Miyano, 1996]. So all the examples are in 3-CNF with a variable number from 50 to 200 and 80 to 1200 clauses (table 2.6). There are unsatisfiable formulas and satisfiable formulas with only one satisfying assignment. The small examples are the only benchmarks the standard QE of REDLOG could solve within acceptable time.

Benchmark	# instances	SAT?	# vars	# cl
aim-50-1_6-no	4	No	50	80
aim-50-1_6-yes1	4	Yes	50	80
aim-100-1_6-no	4	No	100	160
aim-100-2_0-no	4	No	100	200
aim-100-1_6-yes1	4	Yes	100	160
aim-100-2_0-yes1	4	Yes	100	200
aim-100-3_4-yes1	4	Yes	100	340
aim-100-6_0-yes1	4	Yes	100	600
aim-200-1_6-no	4	No	200	320
aim-200-2_0-no	4	No	200	400
aim-200-1_6-yes1	4	Yes	200	320
aim-200-2_0-yes1	4	Yes	200	400
aim-200-3_4-yes1	4	Yes	200	680
aim-200-6_0-yes1	4	Yes	200	1200

Table 2.6: The AIM benchmark suite

II Benchmarks

The II benchmark suite stems from the boolean formulation of a problem of inductive interference [A.P. Kamath & Resende, 1992]. The suite consists of 41 benchmarks, which are all satisfiable. The smallest formula has 66 variables and 186 clauses. The biggest formula consists of 1728 variables and 24792 clauses. A detailed overview is given in table 2.7

Benchmark	# instances	SAT?	# vars	# cl
ii8	14	Yes	66–1068	186–8214
ii16	10	Yes	532–1728	7825–24792
ii32	17	Yes	222–824	1186–20862

Table 2.7: The II benchmark suite

DLX Processor Verification

This benchmark suite is from Miroslav N. Velev of the Department of Electrical and Computer Engineering at the Carnegie Mellon University in Pittsburgh. It is a real world example and is used for model verification of 1- and 2-issue DLX processors. There are 8 examples for correct model verification (table 2.8) and 40 examples for buggy processors and therefore incorrect model verification.
The 40 incorrect model verifications are variations of the `dlx2_cc` processor.

2.7.2 Benchmark Results

All benchmarks were performed on an Apple MacBook Pro with an Intel Core2 Duo 2,2 GHz Processor. Operating System was OS X 10.5 and the REDUCE Developement Version, 15-Mai-2007 was used.

At first we will look at the results for some small examples, namely the `aim-50` and the `ii8` benchmarks (table 2.10). The configuration for the SAT solver is shown in table 2.9

The benchmarks were performed with the standard QE of REDLOG, an implemention of an algorithm of Kapur to solve SAT problems, which was newly implemented in REDLOG, and the new SAT procedure. All the procedures were tested in the PSL and the CSL variant of REDUCE. The results are shown in table 2.10. It is very obvious that the new procedure yields an extrem speedup for solving SAT Problems in REDLOG. The quantifier elimination procedure works well on the small examples with 50 variables (`aim50`) but performs rather bad on the examples with 100 variables or more. For the complete `ii8` suite the SAT solver engine required about 8 seconds versus 53 minutes with the standard QE. None of the unsatisfiable `aim-100` benchmarks could be solved within one hour of time. The SAT procedure yields the result after a few milliseconds. The implementaion of the Kapur algorithm could only solve the `aim-50-1_6-yes1` in PSL but with over one hour of runtime versus 20 milliseconds of the SAT engine. All other instances could not be solved with Kapur because the memory size was exceeded. Thus one can state that this new procedure really levels up the computational power of REDLOG.

The complete overview of all the benchmarks with the SAT engine is given in table 2.11. We conclude that all problems of the `aim` suite can be solved within a few seconds of cpu time. Also the `ii` suite can be solved in moderately time. The processor verification examples from the `dlx` suite took about 2 minutes each with some exceptions. The reason for this is that we do not have a special procedure for binary clauses, which appear very numerous in such verifications. But again: These examples are not solvable with the standard QE of REDLOG in acceptable time.

It should also be mentioned that tuning the parameters can have a big impact on the speed. For instance the whole `aim200` suite could be solved within 10 seconds when chosing other parameters for restart values and initial restart clause number. But the default parameters were chosen to perform well on any sort of problem: random generated such as `aim` and industrial examples such as `dlx` and `ii`. But for special computations it can be useful to change some parameters (See Appendix A).

2.7.3 Comparison to Modern SAT Solvers

Naturally the implementation of the SAT solver in REDLOG cannot compete with current SAT solvers like Satz, Chaff or MiniSat. The reason is obvious: These solvers were partially developed for more then ten years by many researchers. The SAT solver in REDLOG was not designed to compete with these solvers but as a proof of concept to show, how SAT solving can be used in a framework of First-Order Logic. Here are some reasons why current SAT Solvers perform much better than our implementation:

1. There are many special cases that can be handled separately. For example hardware verification problems often have a large amount of binary clauses. Here the watched literal concept is not necessary because one can store these binary clauses separately. If one of the two literals of such a binary clause is assigned and the clause is not **true**, it must be unit. Therefore no computation is necessary.

2. The data structures used in modern SAT solvers are often more complex than our implementation with two lists of lists. Since most of the solvers are written in C they use arrays for the clause database. But it became apparent that the performance of vectors in RLISP is very poor, which is why we used our list implementation. For variables a priority queue with the variable to be chosen next as top element can be utilised.

3. A cache-optimised implementation has a very big impact on the runtime of a solver.

[Zhang & Malik, 2003a] deals with this problem. In RLISP there are not as many possibilities to deal with these cache effects as in languages such as C and C++ since most of the memory management is done by the REDUCE system itself.

4. Many SAT solvers do much more pre-processing of the original formula than the two points mentioned in section 2.6.2. Special reduction strategies such as HypBinRes [Bacchus & Winter, 2003] can be used to reduce the number of clauses and variables for instance.

As already mentioned: The goal was not to implement a SAT solver which can compete with modern solvers, but to show how current techniques work also within a framework of First-Order Logic. Based on this proof of concept we will expand our current SAT solving engine to a Q-SAT solver. Since Q-SAT solvers are developed first for a few years, the current implementations are not as improved as the SAT solvers. We will demonstrate that our implementation of the Q-SAT engine can even compete with some of the modern solvers.

Benchmark	# vars	# cl	processor type
`dlx1_c`	295	1592	1-issue, 5-stage DLX processor, 6 instruction types: register-register, register-immediate,load, store, branch, jump;
`dlx2_aa`	490	2804	two arithmetic pipelines, so that either 1 or 2 new instructions are fetched every clock cycle, conditional on the second instruction in the decode stage having a data dependency on the first instruction in that stage
`dlx2_sa`	490	2804	can execute arithmetic and store instructions by the first pipeline and arithmetic instructions by the second pipeline, so that in addition to the case of the above data dependency, 1 instruction will be fetched also when the second instruction in the decode stage is a store
`dlx2_la`	598	3503	can execute arithmetic, store, and load instructions by the first pipeline and arithmetic instructions by the second pipeline, so that 2 load interlocks come into play now (between the instruction in Execute in the first pipeline and the two instructions in decode) and 0, 1, or 2 new instructions can be fetched each cycle
`dlx2_ca`	1225	9579	has a complete first pipeline, capable of executing the 6 instruction types, and an arithmetic second pipeline, so that 0, 1, or 2 new instructions can be fetched each cycle
`dlx2_cs`	1313	10491	has a complete first pipeline, and a second pipeline that can execute arithmetic and store instructions, so that 0, 1, or 2 new instructions can be fetched each cycle
`dlx2_cl`	1453	12531	has a complete first pipeline, and a second pipeline that can execute arithmetic, store, and load instructions, such that 0, 1, or 2 new instructions can be fetched each cycle, conditional on 4 possible load interlocks and the resolution of the structural hazard of branches and jumps in decode of pipeline two, which need to wait for pipeline one
`dlx2_cc`	1516	12812	has two complete pipelines, 4 possible load interlocks, but no structural hazards, such that 0, 1, or 2 new instructions can be fetched each cycle

Table 2.8: Correct model verification of DLX processors

Branching heuristic:	ZMOM
First restart after:	5 learnt clauses
First variable set to:	$\top$
Increase factor for restarts:	1.2
Clause deletion after:	200 learnt clauses

Table 2.9: The configuration of the SAT Solver for the benchmarks

Benchmark	QE (PSL/CSL)	Kapur (PSL/CSL)	SAT (PSL/CSL)
aim-50-1_6-no	13.17 / 108.9	— / —	0.01 / 0.1
aim-50-1_6-yes1	3.14 / 24.42	3925.91 / —	0.02 / 0.04
ii8	3230.04 / 6487.26	— / —	8.53 / 55.32

Table 2.10: Benchmark results for small examples (time in sec)

Benchmark	Q-SAT (PSL)
aim-100-1_6-no	0.09
aim-100-2_0-no	0.17
aim-100-1_6-yes1	0.57
aim-100-2_0-yes1	1.79
aim-100-3_4-yes1	0.36
aim-100-6_0-yes1	0.13
aim-200-1_6-no	0.11
aim-200-2_0-no	0.11
aim-200-1_6-yes1	16.26
aim-200-2_0-yes1	1.49
aim-200-3_4-yes1	19.08
aim-200-6_0-yes1	6.81
ii8	8.53
ii16	457.47
ii32	52.330
dlx2_cc_bug (40 instances)	14,711.93
dlc1_c	4.87
dlx2_aa	40.47
dlx2_sa	439.54
dlx2_la	—
dlx2_ca	—
dlx2_cl	—
dlx2_cc	—

Table 2.11: Benchmark results for large examples (time in sec)

3 The Next Step: Q-SAT

In the last section we examined an algorithm for solving SAT problems, which means answering the question of the satisfiability of formulas in Propositional Logic. As described in section 2.1.1 formulas in Propositional Logic can be interpreted as formulas in First-Order Logic in which every propositional variable is an existentially quantified variable and instead of literals we use equations as atomic formulas.

Example 3.1 *The SAT Problem*

$$(a \vee b \vee c) \wedge (a \vee \neg b) \wedge (\neg b \vee \neg c)$$

can also be written as:

$$\exists a\ \exists b\ \exists c\,((a = \top \vee b = \top \vee c = \top) \wedge (a = \top \vee \neg b = \top) \wedge (\neg b = \top \vee \neg c = \top))$$

Thus the SAT solving algorithm is an algorithm to solve fully existentially quantified formulas in First-Order Logic. In our next step we will also allow universal quantifiers but still under the condition that all variables are quantified — in such a way that there are no free variables. This problem is referred to as *Q-SAT (Quantified Boolean Satisfiability Problem)*.

At first we will give a formal description of the problem and its complexity in section 3.1. In 3.2 we will state a version of the CDCL algorithm which is also operating on Q-SAT problems and its implementation details will be shown. We will also look at the correctness of the algorithm there. In the last section we will examine several benchmarks made with the new procedure and compare them to the standard quantifier elimination and to modern Q-SAT solvers.

3.1 The Q-SAT Problem

As already mentioned the Quantified Boolean Satisfiability Problem is the question whether a fully quantified formula in Propositional Logic is **true** or **false**. Since we now have quantifiers we do not speak of satisfiable and unsatisfiable, because the answer to a fully quantified formula must always be yes or no.
Again we can interpret a fully quantified formula in Propositional Logic also as fully quantified formula in First-Order Logic.

Example 3.2 *The Q-SAT Problem*

$$\forall a\ \exists b\ \forall c\ \ ((a \vee b \vee c) \wedge (a \vee \neg b) \wedge (\neg b \vee \neg c))$$

can also be written as:

$$\forall a\ \exists b\ \forall c\,((a = \top \vee b = \top \vee c = \top) \wedge (a = \top \vee \neg b = \top) \wedge (\neg b = \top \vee \neg c = \top))$$

3.1.1 Description of the Problem

Definition 3.1 (Q-SAT Problem) *The Q-SAT Problem is the question whether a given fully quantified input formula in Propositional Logic or its counterpart in First-Order Logic is* **true** *or* **false***.*

To provide an efficient algorithm we need to make two assumptions to the input formula. At first the formula must be in Prenex Normal Form (PNF) and at second its formula matrix (the formula without quantifiers) must be in CNF. An arbitrary formula can be converted to be in PNF and CNF in polynomial time. Now we will define PNF.

Definition 3.2 (Prenex Normal Form (PNF)) *A formula φ is in PNF if all quantifiers are written at the beginning of the formula. So if ψ is the formula matrix of φ, it can be written as*

$$Q_1 x_1 \; Q_2 x_2 \; ... \; Q_n x_n \; \psi \tag{3.1}$$

where $Q_i \in \{\exists, \forall\}$ and $x_i \in \mathcal{V}(\psi)$

As we will demonstrate we can use the variable selection heuristics from the SAT solver but they need to follow one additional condition: the quantification level of a variable.

Definition 3.3 (Quantification Level) *In a formula in PNF such as 3.1, successive quantifiers of the same type can be grouped into sets such as*

$$Q_1 x_{1,1} ... x_{1,m_1} \; Q_2 x_{2,1} ... x_{2,m_2} \; ... \; Q_n x_{n,1} ... x_{n,m_n} \; \psi \tag{3.2}$$

where $Q_i \in \{\exists, \forall\}$ and $x_{j,1} ... x_{j,m_j} \in \mathcal{V}(\phi)$ are quantified at the same level, beginning at the outermost level with 1 and increasing the level with every quantifier shift.

Example 3.3 *The corresponding quantifier levels for the following formula:*

$$\underbrace{\exists x \; \exists y}_{level1} \; \underbrace{\forall z \; \forall w}_{level2} \; \underbrace{\exists u}_{level3} (x \vee y \vee z \vee w \vee u)$$

A heuristic for selecting variables must obey the quantification level in the following sense:

Proposition 3.1 *A variable with quantification level n may be first assigned if all variables with quantification level $< n$ are assigned. Within one level the selection of the variable is arbitrary.*

Now we defined all necessary preliminaries to illustrate the Q-SAT problem.

3.1.2 Illustration of the Q-SAT Problem

As for the SAT problem one can imagine the Q-SAT problem as a binary tree. But in the SAT case it was sufficient to derive one satisfying leaf so that the formula was satisfiable. If there was no satisfying leaf, the formula was not satisfiable.

In the Q-SAT case however the problem is more complex. One single satisfying leaf is often not enough because of the occurrence of universal variables in the input formula. For this reason there are two different types of nodes in a Q-SAT tree:

1. Existential nodes, representing existential variables
   Since the question is whether a variable exists, it is sufficient that one of the two branches of an existential node has a satisfying leaf to turn the node to **true**.

2. Universal nodes, representing universal variables
   A formula must hold for both assignments of a universal variable, $\top$ and $\bot$. So a universal node needs a satisfying leaf in both branches to be **true**.

A given formula is **true** if the root node is **true**, otherwise **false**. As will be demonstrated in the example, there are two sorts of backtracking: A conflict driven backtracking such as in the SAT case and a backtracking in case a satisfying assignment was found. The conflict driven backtracking works as described in the last chapter. But in case a satisfying leaf is found, we maybe need to backtrack to universal variables and flip their value.

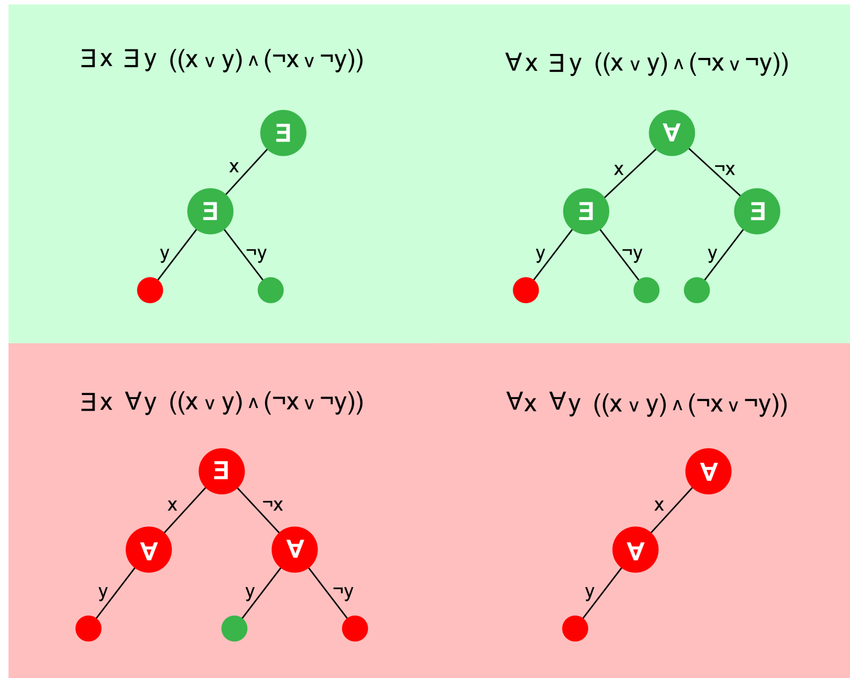

Figure 3.1: Search tree for a Q-SAT problem

Example 3.4 *Consider the formula $(x \vee y) \wedge (\neg x \vee \neg y)$ with different quantifications of x and y to understand the impact of the quantification to the search tree.*

1. $\exists x\ \exists y\ ((x \vee y) \wedge (\neg x \vee \neg y))$
   Since all variables are existential, this is a SAT problem. Every node is an existential node so one single satisfying leaf (green) is sufficient. By assigning x to $\top$ and y to $\bot$ we derive a satisfying assignment and can output **true**.

2. $\forall x\ \exists y\ ((x \vee y) \wedge (\neg x \vee \neg y))$
   Now x is universally quantified and has quantification level 1. So we must first branch after this variable. Since it is universal we need a satisfying leaf in each of the branches $x \longleftarrow \top$ and $x \longleftarrow \bot$. In both of the branches there is an existential node, so we need just one satisfying leaf in one of its branches. For $x \longleftarrow \top$ we

choose $y \longleftarrow \bot$. *Thus we get a satisfying leaf, but now we have to backtrack to the last universal variable and flip it, since the formula must hold for every choice of* x. *For* $x \longleftarrow \bot$ *we choose* $y \longleftarrow \top$ *so that we have satisfying leaves for each of the two branches of the universal node and the formula is* **true**.

3. $\exists x\ \forall y\ ((x \lor y) \land (\neg x \lor \neg y))$
   x *is existentially quantified so one satisfying leaf in one branch is sufficient. But since* y *is universally quantified each of this nodes needs two satisfying leaves. If* $x \longleftarrow \top$ *we get a non satisfying leaf (red) with* $y \longleftarrow \top$ *and this branch cannot hold. For this reason we flip* x *to* $\bot$. *Now we have one satisfying leaf with* $y \longleftarrow \top$, *but again with* $y \longleftarrow \bot$ *the formula does not hold. So none of the two branches of the existential variable is satisfied and the formula is* **false**.

4. $\forall x\ \forall y\ ((x \lor y) \land (\neg x \lor \neg y))$
   In the last case both variables are universally quantified. For this reason every leaf must be a satisfying leaf. Since the first combination of $x \longleftarrow \top$ *and* $y \longleftarrow$ *top is already non satisfying, the formula is* **false**.

3.1.3 Complexity of Q-SAT

We demonstrated that SAT is a classical NP-complete problem. Q-SAT is harder than SAT because backtracking is required also for satisfying leaves in the search tree. The Q-SAT problem is in another complexity class.

Theorem 3.2 (Complexity of Q-SAT) *The Q-SAT problem is PSPACE-complete, meaning the decision problem can be solved on a deterministic Turing machine in polynomial space. Every decision problem that can be solved in polynomial space can be reduced to the Q-SAT problem in polynomial time.*

The proof for this theorem is given e.g. in [Fujiwara & Toida, 1982].

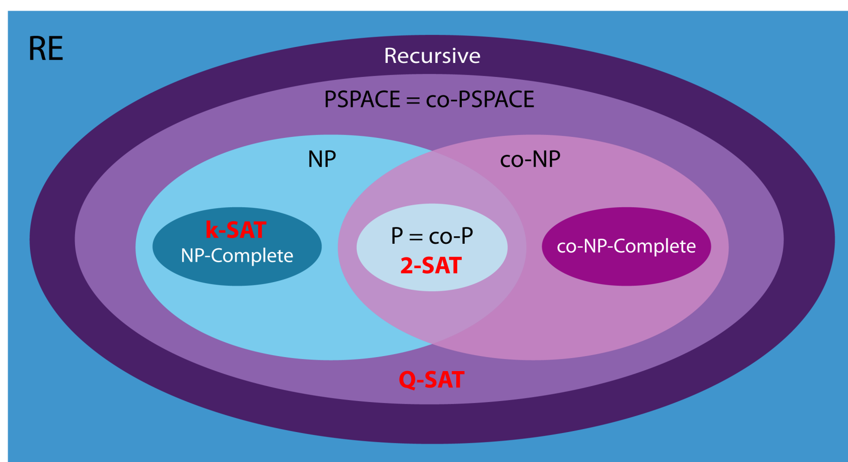

Figure 3.2: The complexity hierarchy

Figure 3.2 shows an illustration of the relevant fragment of the complexity hierarchy.

The innermost ellipse represents the complexity class **P** within which all problems which can be solved on a deterministic Turing machine in polynomial time are found. These are the problems one considers as "tractable" or "efficiently computable". The 2-SAT problem is in **P**. The next two complexity classes are **NP** and **co-NP**. In **NP** lie all problems which can be solved in polynomial time on a non-deterministic Turing machine. A problem P is in **co-NP** if its complement $\overline{P}$ is in **NP**. In both classes we have the subset of the **NP-complete** and **co-NP-complete** Problems. The property of these problems is that each can be reduced to each other in polynomial time. In section 1 we demonstrated, that the k-SAT problem is **NP-complete**.

Q-SAT is a **P-SPACE-complete** problem. Every problem in **P, NP** or **co-NP** can be solved in **P-SPACE**. Complexity classes beyond **P-SPACE** are **R** for all problems which are decidable on a Turing machine or **RE (Recursive enumerable)** for all problems which are semi-decidable on a Turing machine.

3.2 The Q-SAT Version of the CDCL Algorithm

As we will see, the Q-SAT version of the CDCL algorithm is principally the same as the SAT version but also with backtracking in the satisfaction case. At first we will consider how the data structure must be changed to store also information with respect to the quantification. In the second part we will examine how the adapted algorithm works. Because of universal variables we need new rules for empty clauses and unit clauses, which we will look at in part 3 before we will glance at the correctness of the described algorithm.

3.2.1 Changes to the Data Structure

We add three additional fields to the data structure of the variable, which are shown in table 3.1

Position	Abbreviation	Description
#13	`quant`	the quantifier of the variable
#14	`qlevel`	the quantification level of the variable's quantifier
#15	`flip`	a flag if the variable is flipped or not

Table 3.1: The additional fields of a variable for Q-SAT

`quant` stores `ex` for an existential variable and `uni` for a universal variable. If the variable is not quantified at all (which we will required in the next chapter) `nil` is stored. In `qlevel` the quantification level of the variable's quantifier is stored. This is necessary because the branching heuristics have to respect this level. The third field `flip` is a flag which indicates if the variable has already flipped its assignment or not. This is important for backtracking in the case of satisfaction. There we must backtrack to the last unflipped universal variable. Which is why we must store this value. The data structure for clauses is not to be changed because clauses are furthermore disjunctions of literals with no respect to their quantification.

3.2.2 The Q-SAT Algorithm

The Q-SAT algorithm is an alteration of the standard CDCL procedure demonstrated in chapter 2. This version of the algorithm was proposed 2002 in [Zhang & Malik, 2002a].

```
1  level := 0;
2  UP(f, α);
3  level := 1;
4  while true do
5  |  choose an unassigned x ∈ V(F) (wrt. the q-level);
6  |  α := α ∪ [x ⟵ ⊥];
7  |  while true do
8  |  |  UP(f, α);
9  |  |  if a conflict is reached then
10 |  |  |  level := analyseConflict;
11 |  |  |  if level = 0 then
12 |  |  |  |  return false
13 |  |  |  end
14 |  |  |  backtrack(level);
15 |  |  else
16 |  |  |  if formula is satisfied then
17 |  |  |  |  level := analyseSAT;
18 |  |  |  |  if level = 0 then
19 |  |  |  |  |  return true
20 |  |  |  |  end
21 |  |  |  |  backtrack(level)
22 |  |  |  else
23 |  |  |  |  level := level + 1;
24 |  |  |  |  break;
25 |  |  |  end
26 |  |  end
27 |  end
28 end
```

Algorithm 8: The CDCL procedure for Q-SAT

In lines 1 and 2 the initial UP is performed. A combinations of two while loops follows. These loops are left if the final result is computed and the return value is **true** or **false**. In the beginning of the outer while loop an unassigned variable is selected by the heuristic and set to a certain value (lines 5,6). As already mentioned, this selection must obey the quantification level so all variables of one level must have been assigned before a variable of the next level can be selected.

After this assignment the Unit Propagation is performed (line 8). If this UP leads to a conflict, this conflict is analysed, a new clause is learnt and the backtrack level is computed (line 10). Again the new learnt clause is constructed in such a way that it gets unit after backtracking. If we have to backtrack to level 0, the original formula had a conflict and so we can return **false** (line 12) else we perform the backtrack (line 14) and jump to the begin of the inner while loop (line 8) where we perform UP again (where we have at least the one UP for the new learnt — and now unit — clause).

If the formula was satisfied after the UP in line 8, we analyse this satisfaction case, which means we calculate a backtrack level and could also compute a (satisfaction driven)

new clause. If we must backtrack to level 0, we know that we tested all universal variables with all assignments and can finish. In this case we return **true** (line 19), otherwise we backtrack to the computed level.

If neither a conflict nor a satisfaction case was derived after UP, we increase the level, leave the inner while loop and jump to the beginning of the outer loop where we choose a new literal. It is obvious that there must be an unassigned literal if there is neither a conflict nor a satisfaction case.

The analyseConflict Procedure

The analyseConflict procedure is literally the same as given in chapter 2:

```
   Input: an empty clause ec, the set of clauses C
 1 lv := the last assigned variable before the conflict clause;
 2 reas := the reason of lv;
 3 newclause := resolve(ec,reas);
 4 while the QSAT stop criterion for newclause is not met do
 5 |  cv := chooseLiteral(newclause);
 6 |  reas := the reason of cv;
 7 |  newclause := resolve(newclause,reas);
 8 end
 9 C := C ∪ {newclause};
10 level := computeBTLevel(newclause);
11 return level;
```

Algorithm 9: The analyseConflict procedure for Q-SAT

The only thing changed is the stop criterion. We have to achieve that the new learnt clause is unit after backtracking which is why we need an altered criterion here. We will look at this criterion in the next chapter when we consider the new rules for UP and empty clauses.

The analyseSAT Procedure

In the case of satisfaction we need to backtrack to the last unflipped universal variable. What does this mean? When assigning a variable we set its flag `flip` to **false** because its value is set but still not flipped. If we change the assignment of this variable because of backtracking, the flip value is set to **true**. For backtracking in the satisfaction case we must search the unflipped universal variable with the highest quantification level, flip its value and return its level:

```
1 lv := search the unflipped universal variable at the highest q-level;
2 flip the value of lv;
3 set flip(lv) to true;
4 level := getLevel(lv);
5 return level;
```

Algorithm 10: The analyseSAT procedure for Q-SAT

This can be imagined as descending into the second branch of the variable in the search tree. If we found a satisfying assignment for the first branch, we must backtrack to the last universal node and descend into the second branch to find a satisfying leaf there.

There are ideas, that even in the case of satisfaction the solver could learn new clauses. In [Zhang & Malik, 2002b] such an idea is presented. But it became apparent that these

improvements are only interesting for special sorts of problems. So the REDLOG Q-SAT solver does not utilise this technique.

3.2.3 The New Rules for UP and Empty Clauses

In the SAT case we had already rules for unit clauses and empty clauses. An unsatisfied clause is unit if all its variables but one are assigned and an unsatisfied clause is empty if all variables are assigned. In the Q-SAT case this is more complex because we must also deal with universal variables. We will state the new rules for unit and empty clauses in Q-SAT. Let $E(C)$ be the set of existential variables of a clause C of a CNF and $U(C)$ the set of universal variables of this clause C. $ql(l)$ is the quantification level of a variable.

Unit Clauses

The first important point to mention is, that only existential variables can be implied variables. A universal variable can never be implied because the formula must hold for all of its assignments. So the first two rules for a unit clause C are the same conditions as in the SAT case.

1. There exists $e \in E(C), e = nil$.
2. For any $e' \in E(C), e' \neq e : \; e' = \bot$

These two conditions deal with the existential variables. There may be only one unassigned existential variable in a unit clause. All other literals must evaluate to $\bot$. The third rule handles the universal variables:

3. For all $u \in U(C)$ with $u \neq \top$: $u = nil \Rightarrow ql(u) > ql(e)$

Certainly if a universal variable is evaluated to $\top$, the clause is satisfied and can be no unit clause. For all unassigned universal variables it must hold that their quantification level is higher than the one of the sole unassigned existential variable.

Example 3.5 *Let a, b, c be existential and x, y universal variables. The assignment should be $[a \longleftarrow \bot, c \longleftarrow \top, x \longleftarrow \bot]$. We are currently at level 5. The clause*

$$(a_{(2)} \vee b_{(5)} \vee \neg c_{(3)} \vee x_{(1)} \vee y_{(6)})$$

is unit because b is the sole unassigned existential variable and the only unassigned universal variable y is at a higher quantification level (6) than b (5). The clause

$$(a_{(2)} \vee b_{(5)} \vee \neg c_{(3)} \vee x_{(4)} \vee y_{(1)})$$

is not unit because the unassigned universal variable y is at a lower quantification level than b

---

Empty Clauses

The rule for empty clauses is altered in the following way: A clause C is empty if

1. for all $e \in E(C) : e = \bot$
2. for all $u \in U(C) : u \neq \top$

It is obvious that no existential variable may be evaluated to $\top$ because then the clause would be satisfied and therefore could not be empty. No universal variable may be evaluated to $\top$ for the same reasons. But a universal variable can be unassigned and the clause is still empty. This is because we must consider both branches of the universal variable. In one of which the clause will be surely empty.

Example 3.6 *Let a, b, c be existential and x, y universal variables. The assignment should be $[a \longleftarrow \bot, b \longleftarrow \bot c \longleftarrow \top, x \longleftarrow \bot,]$ and we are currently at level 5. The clause*

$$(a_{(2)} \vee b_{(5)} \vee \neg c_{(3)} \vee x_{(1)} \vee y_{(6)})$$

is an empty clause because all existential variables evaluate to $\bot$, x evaluates to $\bot$ and the universal variable y is not set.

The Stop Criterion for Resolution

Since the idea of clause learning is that the new learnt clauses get unit after the next backtrack, we must modify the stop criterion in line 4 of the analyseConflict procedure. We have to ensure that the new learnt clause is unit according to the three rules we stated above. So we stop if the new clause satisfies the following three conditions:

1. Only one existential variable e is at the highest decision level
2. e is in a decision level where the decision variable is existentially quantified
3. All universal variables u with $ql(u) < ql(e)$ are assigned to $\bot$ before the decision level of e

The first and the second rule ensure that the three conditions of the UP rule are satisfied and that the new clause is unit after backtracking. The second rule ensures that there is at least one unassignment of an existential variable. If there were none, the new learnt clause would have no effort. The backtrack level is again the highest level of a variable in the new learnt clause which is really smaller than the decision level.

We also need to consider two exceptions: If all existential variables in the resulting clause are at level 0 or if there are no existential literals in the new clause, we can stop immediately and return **false**.

3.2.4 Correctness of Resolution

At first glance resolution in the Q-SAT case seems to be exactly the same as in the SAT case. But there is a important difference. If two clauses are resolved in SAT, the distance between these two clauses is always one. By distance we refer to the number of literals having a positive occurrence in one clause and a negative occurence in the other clause. However in the Q-SAT case this distance can be higher than one, so one speaks of *long distance resolution*. To understand why such cases can arise, we will look at a small example.

Example 3.7 *Consider these two clauses of a larger CNF:*

$$(a_{(1)} \vee b_{(3)} \vee x_{(4)} \vee y_{(4)} \vee c_{(5)}) \wedge (a_{(1)} \vee \neg b_{(3)} \vee \neg x_{(4)} \vee \neg y_{(4)} \vee d_{(5)}) \tag{3.3}$$

where a, b, c, d are existential variables and x and y are universal. The numbers indicate the quantification level. Suppose the assignment $[c \longleftarrow \bot,\ d \longleftarrow \bot]$ at level 5. The other variables are all unassigned. At level 6 we choose variable a to be $\bot$. Now the first clause is unit and b must be assigned to $\top$. But in case we do so, the second clause conflicts. So we must do a resolution of the second clause with the reason for the assignment of b, which is the first clause. The result is

$$(a_{(1)} \vee x_{(4)} \vee \neg x_{(4)} \vee y_{(4)} \vee \neg y_{(4)} \vee c_{(5)} \vee c_{(5)}) \tag{3.4}$$

We see: because of the universal variables there is a distance of three (b, x, y) between the two clauses. The resulting clause is a tautology, because every choice for x or y will turn it **true**.

Consequently the question arises if we can handle these tautology clauses just as other clauses. Therefore it is important to emphasise, which services clauses in a CNF should provide. The first service is to provide information about implication rules (UP) and thus leading to new search spaces. The second service is to reveal conflicts. We demonstrated two rules for these: The UP rule and the empty clause rule. So now it must be shown that the tautology clauses preserve these rules. The central result of [Zhang & Malik, 2002a] is that if the two input clauses of resolution obey these two rules, the new clause also does — independent of the distance between the two original clauses. This result allows us to treat this tautology clauses as conventional clauses.

3.3 Comparison to QE and to Other Q-SAT Solvers

In this section some benchmarks will be presented. The first one is a classical benchmark from the Q-SAT research community, the "Bomb in the Toilet problem", a planning problem. The second benchmark is taken from [Seidl & Sturm, 2003] and computes series of a 2 Player game. The third benchmark is the "Chain"-benchmark from the Rintanen benchmark suite. We will compare the new Q-SAT procedure with the standard quantifier elimination and with existing Q-SAT solvers.

3.3.1 Comparison to QE

From the SAT solving examples we learnt, that the new CDCL algorithm is much faster than the existing QE procedure and that we can handle even large problems with the new solver. We will also see that the new Q-SAT procedure really increases the computational power of REDLOG. Most of the examples we will see in the benchmarks were not solvable with the standard quantifier elimination within acceptable time.

The Bomb in the Toilet Problem

First we will look at the bomb-in-the-toilet problem, which is a classical planning problem. We compute five sets of benchmarks, each with 3 blocks of quantifiers: An existential block, a universal block and again another existential block (table 3.2).

Benchmark	# u-vars	# e-vars	# clauses	QE	Q-SAT
toilet_a_02	2	16–88	39–408	310	< 10 ms
toilet_a_04	4	28–136	129–894	4780	< 10 ms
toilet_a_06	6	40–240	491–1724	—	0.9
toilet_a_08	8	52–660	2205–6929	—	29.4
toilet_a_10	10	64–640	10455–13137	—	1375.9

Table 3.2: Benchmark results for the bomb in the toilet problem (time in sec)

We see that this is a good example how fast Q-SAT can solve problems, compared to the standard quantifier elimination. Even in the set with 4 universal quantifiers there were two examples which exceeded the heap size with the standard QE.

The next benchmark shows an interesting problem. Before we introduced pure literal elimination as a pre-processing step, this examples had performed very badly with the new Q-SAT procedure. But with pure literal elimination these examples can be solved completly during pre-processing. Consequently this is a very effective step in pre-processing.

A Two-Player Game

Consider the formula

$$\varphi := \exists x_1 \forall x_2 ... Q_n x_n \left(\bigwedge_{i=1}^{n-1} \neg(x_i \vee x_{i+1}) \right) \tag{3.5}$$

The formula has n variables and $2(n-1)$ clauses and can be interpreted as a 2-person game: Two players, $\exists$ and $\forall$ move alternatingly and choose an assignment for the current variable x_i. $\exists$ begins the game. The goal of $\exists$ is to turn φ to **true**, $\forall$ tries to turn it to **false**. As one can see $\exists$ can win the game easily by assigning all x_i with i odd to $\bot$. Then every NAND computes to $\top$ and φ is **true**. Because every x_i with i odd has only negative occurrence in the formula, the x_i will be assigned to $\bot$ and the satisfied clauses are deleted. So after pre-processing, the formula has no more clauses and thus is **true**. The standard quantifier elimination performed also well on these examples. The results are presented in table 3.3

Benchmark	# u-vars	# e-vars	# clauses	QE	Q-SAT
n=10	5	5	18	< 10 ms	< 10 ms
n=50	25	25	98	0.02	< 10 ms
n=100	50	50	198	0.09	< 10 ms
n=250	125	125	498	0.8	< 10 ms
n=500	250	250	998	5.1	0.03
n=1000	500	500	1998	36	0.08
n=1500	750	750	2998	115.9	0.18

Table 3.3: Benchmark results for the 2 player game (time in sec)

The Chain Benchmark

The chain benchmark is from the Rintanen benchmark suite. It is — as bomb in the toilet — a planning problem and also consists of three blocks of quantifiers: An existential block, a universal block and another existential block. This benchmark was chosen to

demonstrate that also large examples with many variables, clauses and also a larger number of universal quantifiers can be solved in moderately time. None of the problems could be solved with the standard quantifier elimination. They all caused a low-heap error. The results are shown in table 3.4

Benchmark	# u-vars	# e-vars	# clauses	QE	Q-SAT
`12v.13`	12	913	4852	—	0.97
`13v.14`	13	1067	5458	—	1.55
`14v.15`	14	1233	6424	—	2.89
`15v.16`	15	1411	7483	—	5.62
`16v.17`	16	1601	8638	—	11.28
`17v.18`	17	1803	9892	—	23.06
`18v.19`	18	2017	11248	—	47.23
`19v.20`	19	2243	12709	—	99.9
`20v.21`	20	2481	14278	—	206.9
`21v.22`	21	2731	15958	—	439.4

Table 3.4: Benchmark results for the chain benchmark (time in sec)

3.3.2 Comparison to Modern Q-SAT Solvers

For the REDLOG SAT solver we demonstrated that for various reasons it cannot compete with modern SAT solvers. The Q-SAT solver cannot reach the computational power of modern solvers but it can compete with some of them. Since the Q-SAT research community is still growing there are not as many long developed solvers as in the SAT community. This leads to the effect that we can compete with some of these solvers and even solve certain problems more efficiently than them.

Looking at the bomb in the toilet problem for instance, we can compare the runtime of the REDLOG solver for the complete ToiletA series (1500 seconds) with the results of Q-SAT solvers listed on QBFLIB [1], the homepage of the research community. In 2004 there were 16 different solvers which computed this benchmark suite. Three of them were slower, three of them were equal to REDLOG. Even in 2006 two solvers were still slower than our implementation.

For many of the examples, such as Rintanen's chain benchmark our results are slower than the best results of modern solvers about a factor between 10 and 50. Since the Q-SAT solver is also just a proof of concept implementation these results are very convincing.

After implementing the Q-SAT solver we can handle fully quantified formulas in Propositional Logic or their First-Order Logic counterparts. The last step is to allow free variables in the input formula too. Modern SAT or Q-SAT solvers cannot process such formulas. Consequently the output is no longer **true** or **false** but conditions in the free variables.

[1] www.qbflib.org

4 Quantifier Elimination with Parametric Q-SAT

In chapter 3 we demonstrated how to solve fully quantified satisfiability problems. But the Q-SAT solver can still not handle free variables. To reach the goal of performing quantifier elimination with SAT algorithms we have to take another step: we must allow free variables, so parameters. This leads to the introduction of *parametric Q-SAT*.

4.1 Parametric Q-SAT

4.1.1 The Parametric Q-SAT Problem

In contrast to SAT and Q-SAT the parametric Q-SAT problem does not decide the question whether a given formula is satisfiable or unsatisfiable with **true** or **false**. The output is a DNF (disjunctive normal form) of conditions for the free variable so that a satisfying assignment of the quantified variables exists. If for all possible assignments of the free variables a satisfying assignment of the quantified variables can be found, the output is **true**. If there is no assignment of the free variables to do so, the output is **false**.

Example 4.1 *In chapter 3 we considered that the formula* $\exists x\ \forall y((x \vee y) \wedge (\neg x \vee \neg y))$ *is* **false**. *If we alter the formula by adding a new unquantified variable to each clause, the following formula can emerge*

$$\exists x\ \forall y((x \vee y \vee \neg u) \wedge (\neg x \vee \neg y \vee w)) \tag{4.1}$$

Since now there are two free variables we must examine the formula under each assignment of these two:

1. $[u \longleftarrow \bot, w \longleftarrow \bot]$
   Since u *is assigned to* $\bot$ *the first clause is already satisfied. If we assign* x *to* $\bot$*, the second clause is also satisfied for every assignment of* y*. For this reason an* x *exists such that the formula holds for all* y*. Thus for this assignment of the free variables, we find a satisfying assignment of the quantified variables.*

2. $[u \longleftarrow \bot, w \longleftarrow \top]$
   In this case the choice of u *and* w *satisfies both clauses, so for every choice of* x *and* y *the formula holds.*

3. $[u \longleftarrow \top, w \longleftarrow \bot]$
   u *and* w *satisfies neither the first nor the second clause. This is the same case as in the original formula of chapter 2, which we know to be not satisfiable.*

4. $[u \longleftarrow \top, w \longleftarrow \top]$
   Here the second clause is satisfied because of $w \longleftarrow \top$*. So we choose* $x \longleftarrow \top$ *and for every choice of* y *the first clause is also satisfied.*

Satisfying assignments of the quantified variables are found in cases 1), 2) and 4) therefore the output of our parametric Q-SAT algorithm should be (in DNF)

$$(\neg u \wedge \neg w) \vee (\neg u \wedge w) \vee (u \wedge w)$$

If this formula holds for the free variables, a satisfying assignment of the quantified formulas exists so that the output of the formula is **true***.*

4.1.2 Parametric Q-SAT as Quantifier Elimination

With the implementation of parametric Q-SAT we are now able to handle arbitrary input formulas in First-Order Logic over the language of Boolean Algebras. If the input formula is not in CNF, we can convert it as mentioned in chapter 1. If the formula is fully existentially quantified, we start our SAT algorithm. If the formula is fully quantified but with existential and universal quantifiers, we go into the Q-SAT procedure. And if there are quantified and free variables, we start the parametric Q-SAT procedure.

In any case the output of our algorithm is equivalent to the input formula. In the SAT and Q-SAT case this is **true** and **false** and in the parametric Q-SAT case this is a disjunction of conditions to the free variables or also **true** or **false**. But in every case the output will be quantifier free. Thus the result of our implementation is a set of tools to compute an equivalent quantifier free formula to every arbitrary input formula. A quantifier elimination procedure for First-Order Logic formulas over the language of Boolean Algebras emerged.

4.2 The Algorithm

The main idea of this new algorithm is to test all 2^n possible assignments but with the same technique as demonstrated in the DLL procedure. We try to cut the search tree as soon as possible. If we get an input formula φ with free variables, we split the set of all variables in a set of bound (quantified) variables $\mathcal{B}(\varphi)$ and free variables $\mathcal{F}(\varphi)$. The algorithm branches over all free variables and when all free variables are assigned, it performs the standard Q-SAT algorithm.

4.2.1 Operation of the Algorithm

Input: An arbitrary formula φ and the set of free variables $\mathcal{F}(\varphi)$
Output: A DNF of the conditions to the free variables

```
1  PQSAT(φ, F(φ), α);
2  if all x ∈ F(φ) are assigned then
3  |  Save the result of QSAT(φ) and assignment of α;
4  end
5  x := choose a free variable of F(φ);
6  α′ := α ∪ [x ⟵ ⊤];
7  if no conflict is reached after simplification then
8  |  PQSAT(φ, F(φ), α′);
9  end
10 α″ := α ∪ [x ⟵ ⊥];
11 if no conflict is reached after simplification then
12 |  PQSAT(φ, F(φ), α″);
13 end
```

Algorithm 11: The algorithm for parametric Q-SAT

As previously mentioned, the algorithm tries all possible assignments of the free variables. In order to do so it creates a binary tree over all free variables. If a leaf is reached, which means that all variables are assigned, it performs the Q-SAT algorithm (line 3), otherwise it branches after the next free variable and descends into both branches of the search tree. As already seen in the DLL procedure we also have a kind of early cuts in the tree here. If an assignment of free variables already leads to a conflict, we can cut the tree at this node.

Example 4.2 *Consider the formula*

$$\exists x\ \forall y((x \vee y \vee \neg u) \wedge (\neg x \vee \neg y \vee w) \wedge (u \vee w)) \tag{4.2}$$

This is essentially the formula from the last example with the additional clause $(u \vee w)$. *This resulted in the situation shown in figure 4.1. Without the last clause the result would*

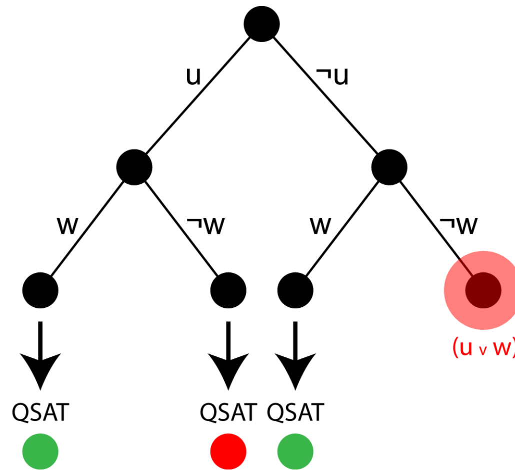

Figure 4.1: Search tree for a parametric Q-SAT problem

be $(\neg u \wedge \neg w) \vee (\neg u \wedge w) \vee (u \wedge w)$ *as in the last example. But it is obvious that* $(\neg u \wedge \neg w)$

cannot hold because of the last clause. However this can be realised during setting w and u. So we can cut the search tree over the free variables here and must not descend into the Q-SAT procedure. As a consequence the output for this example would be $(u \wedge w) \vee (\neg u \wedge w)$.

4.2.2 Complexity of the Algorithm

In principal the algorithm has a worst-case time complexity which is exponential in the number of free variables, because the complete search tree with 2^n leaves is required for n free variables. At the end of each leaf we must perform Q-SAT which is P-SPACE complete. The question arises if the runtime of the algorithm is acceptable for real world problems. But there are two important facts which are associated with conflict driven clause learning and UPL.

1. After a free variable is assigned to a certain value, UP is performed. So we can decide if there is an easyly recognisable empty clause caused by this assignment. If so, we cut the search tree at this node as seen in the last example. The tree is not only cut, when the assignment directly leads to an empty clause but also if it implies an empty clause.

2. We do not use the original clause for each Q-SAT run, but the one from the last run, meaning it is simplified and is extended with new learnt clauses. So after the first few runs the time needed for solving the formula decreases because its number of variables decreases and we have more clauses with higher conflict potential.

These two reasons lead to the effect, that the runtime of our algorithm is — in most examples — not exponential in the number of free variables but almost linear. We will consider some benchmarks for this in the next section.

4.3 Benchmarks

We took some of the standard "bomb in the toilet" benchmarks from the last chapter and increased the number of free variables. If an original formula was **true**, it must stay **true** also when former quantified variables are free now. If the original formula was **false** then turning existential variables into free variables does not change the truth value of the formula. But by turning universal variables into free variables the formula can become **true**.

An interesting fact is, that the speed of the quantifier elimination procedure increases with the number of free variables. This is because the quantifiers are eliminated by expanding them into disjunctions ($\exists$) and conjunctions ($\forall$) and simplifying the results. On the contrary the parametric Q-SAT procedure needs more time with an increasing number of free variables, because the search tree grows exponential. So the question is, whether there is intersection point of the two procedures. If so, we can try to find a criterion when to use Q-SAT and when to switch to the standard quantifier elimination.

The first benchmark (`toilet_a_04_01.4`) is a small one with 60 (4 universal and 56 existential) variables and 229 clauses. We increased the number of free variables from 0 to 50. In figure 4.2 we see an abrupt ascent at 35 free variables. This is the point when the second universal variable was turned into a free variable and the formula changed its truth value from **false** to **true**. From 40 variables on, the quantifier elimination is faster than the Q-SAT procedure.

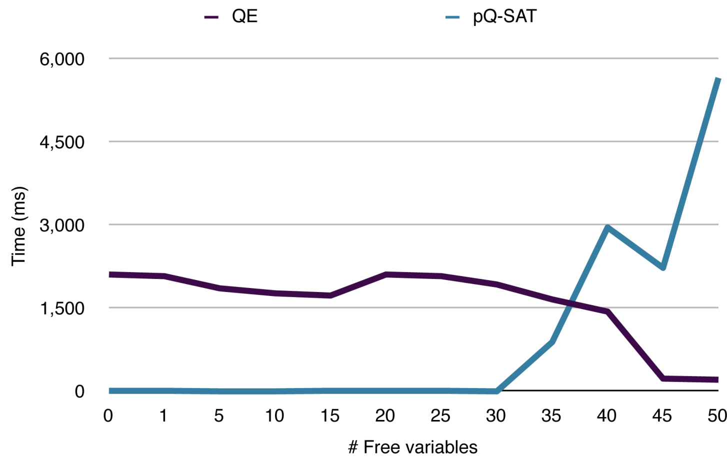

Figure 4.2: Small benchmark results parametric Q-SAT

As demonstrated in the last example, QE was faster when about $\frac{2}{3}$ of the variables were free. The next step is to verify this result on a larger example (`toilet_a_06_01.4`, 80 existential variable, 6 universal variables, 649 clauses) where QE initially requires 50000 ms versus 10 ms of the parametric Q-SAT. But in figure 4.3 we see that there is another intersection point of the two procedures at about 50 free variables which is again about $\frac{2}{3}$ of the variables.

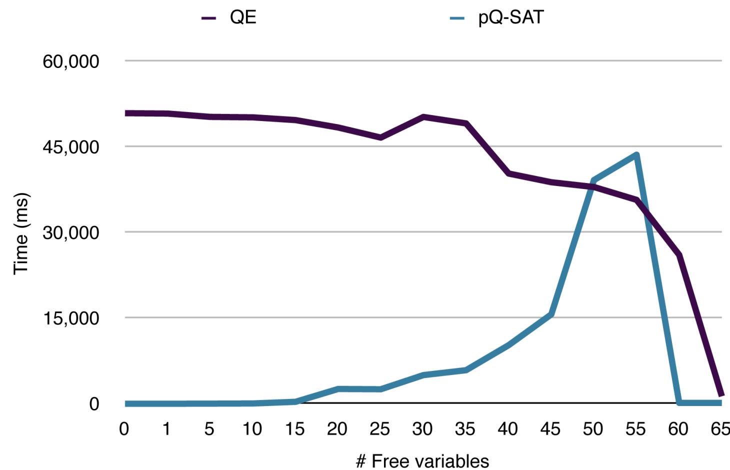

Figure 4.3: Middle benchmark results parametric Q-SAT

To test if this bound of $\frac{2}{3}$ also holds for examples with a great number of clauses, we take the `toilet_a_08_01.2` example with 2205 clauses, 52 existential variables and 8 universal variables. And the same observation can be made again: when about $\frac{2}{3}$ of the variables are free, the quantifier elimination performs much faster than the parametric Q-SAT procedure (figure 4.4). This leads to the conclusion that it is a good strategy to compute the ratio of free variables to quantified variables. If it is about $2:1$, the quantifier elimination procedure should be taken, otherwise the parametric Q-SAT. If there are only free variables, then it is a SAT problem and so we have to choose the SAT procedure.

We saw, that in most cases the parametric Q-SAT procedure is much faster than the quantifier elimination. Because real world problems have seldomly more than $\frac{2}{3}$ free vari-

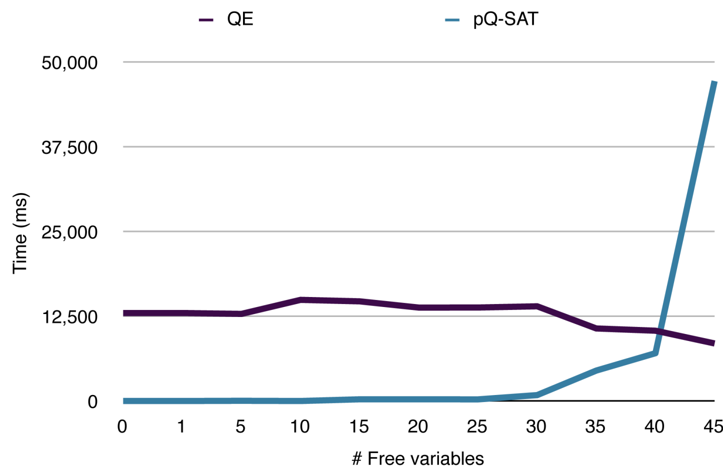

Figure 4.4: Large benchmark results parametric Q-SAT

ables, this new procedure is a real increase of the computational power of REDLOG. And there is also another feature of the parametric Q-SAT procedure: It always detects if a problem is a contradiction and so is equivalent to **false**. For larger examples, the quantifier elimination often yields a large number of conditions which can be simplified to **false**. But with the parametric Q-SAT algorithm the user can observe this immediately after the computation of the problem without running another quantifier elimination.

4.4 Further Ideas and Outlook

The algorithm for parametric Q-SAT is in principal an extended Version of DLL over the free variables and the standard algorithm for Q-SAT in the second step. So the idea arises to use the same concepts used for improving DLL for improving parametric Q-SAT. This means that we could also utilise special heuristics to choose the free variables as we utilised for branching. The idea is to produce fast conflict clauses only with free variables so that the search tree can be cut on early nodes. This could decrease the number of needed Q-SAT executions.

One could also introduce some sort of learning over the free variables. If a certain leaf of the search tree over the free variables led to the Q-SAT result **false**, one could try to decide which free variables led to this result. If we know that some variables did not affect the result, we know also the result if these variables are flipped and so we do not need to compute these cases again. This would also work for the Q-SAT result **true**. All free variables which are not the only literal in one of the clauses, turning it to **true**, can be flipped without changing the result. We think that this approach could reduce the number of needed Q-SAT executions dramatically.

The last idea is that some more powerful reductions could yield in earlier cuts in the search tree and so in less Q-SAT executions. After each assignment of a free variable only UP is performed at the moment, but we could use some simplifications strategies here such as the quantifier elimination procedure in REDLOG. These ideas are could be subject of following work concerning parametric Q-SAT.

A The DIMACS File Format

The DIMACS — the Center for Discrete Mathematics and Theoretical Computer Science — defined many standard file formats for problem instances such as graph colouring, traveling salesman or SAT for instance. There are two file formats: the *.cnf* format for SAT problems and the *.qdimacs* for Q-SAT problems. We will give a short overview of the file formats.

A.1 SAT Problems: .cnf

The file could start with a comment. Each line of the comment must begin with a single `c`. The first line of the problem description begins with `p cnf numvars numclauses`. `numvars` is the number of different variables and `numclauses` is the number of clauses of this instance. Then `numclauses` lines follow: Each line represents one clause and ends in `0`. The variables are represented by numbers from 1 to `numvars`. A negative number indicates a negative literal. The line `4 5 -8 -9 0` could be interpreted as the clause $(v4 \vee v5 \vee \neg v8 \vee \neg v9)$ for instance. A simple .cnf file is shown here:

```
c =============
c || Comment ||
c =============
p cnf 4 3
1 3 -4 0
4 -2 -3 1 0
2 0
```

This file represents the input formula

$$(v1 \vee v3 \vee \neg v4) \wedge (v4 \vee \neg v2 \vee \neg v3 \vee v1) \wedge (v2)$$

A.2 Q-SAT Problems: .qdimacs

The .qdimacs file format uses the same technique to describe the clauses as the .cnf format. But additionally the possibility to denote the variable's quantifiers is required. This is done before the description of the clauses start. Every block of quantifieres starts in a new line beginning with `a` for a universal quantifier and `e` for an existential quantifier. Then all the variables quantified at this level follow. The number `0` indicates the end of a line. We present a small example for a .qdimacs file.

```
c =============
c || Comment ||
c =============
p cnf 3 4
e 2 0
```

```
a 3 0
e 1 0
1 3 0
2 -3 0
-2 -3 1 0
2 0
```

This represents the input formula

$$\exists v2\ \forall v3\ \exists v1\ (v1 \vee v3) \wedge (v2 \vee \neg v3) \wedge (\neg v2 \vee \neg v3 \vee v1) \wedge (v2)$$

A parser for both file formats is implemented in REDLOG. The files can be loaded and solved in one step with `rlqsatdimacs` or only read and converted in a Pseudo Lisp Prefix formula with `rlreaddimacs`. Further descriptions for these two commands will be presented in the next chapter — the user manual.

B User Manual

We will give a short overview of the new commands in REDLOG concerning the SAT-Solving and its options and present some examples.

B.1 Reading Input Formulas

There are three ways of reading and solving an input formula:

1. Use a formula already present in Pseudo Lisp Prefix (`rlqsat`).
2. Read and solve a .cnf or .qdimacs file (`rlqsatdimacs`).
3. Read a .cnf or .qdimacs file and convert it to Pseudo Lisp Prefix (`rlreaddimacs`).

You will choose the first way if you already have a formula in Lisp Prefix from another computation or because you typed it in. This way is required for parametric Q-SAT problems or problems which are not in CNF because there are no other file formats for these problems. If the formula `f` is given in Pseudo Lisp Prefix, you can solve it with the command `rlqsat f`. We will demonstrate this on a parametric Q-SAT example:

```
1: load redlog;

2: rlset ibalp;

*** turned off switch raise

{}

3: f := (X or Y or not U) and (not X or not Y or W);

f := (x = 1 or y = 1 or not(u = 1)) and (not(x = 1) or not(y = 1) or w = 1)

4: g := ex(x,all(y,all(u,f)));

g := ex x all y all u ((x = 1 or y = 1 or not(u = 1))

 and (not(x = 1) or not(y = 1) or w = 1))

5: rlqsat g;
Deleted redundant clauses: 0

w = 1
```

If you have an input file in a .cnf or .qdimacs format you can solve it with the command `rlqsatdimacs "filename.cnf"` or `rlqsatdimacs "filename.qdimacs"`. You do

not need to specify if it is a SAT or a Q-SAT problem. The system will choose the appropriate algorithm automatically. We will perform this action with the example from the last chapter:

```
1: load redlog;

2: rlset ibalp;

*** turned off switch raise

{}

3: rlqsatdimacs "test.qdimacs";
Reading 3 variables and 4 clauses
Q-SAT: Reading quantifiers
Deleted Redundant Clauses: 0
Deleted variables in pre-processing: 1

true
```

The last option is only to read a .cnf or .qdimacs file and returning its Pseudo Lisp Prefix counterpart. This is interesting when you want to change the formula within REDLOG or use other solving algorithms such as the quantifier elimination or the Kapur algorithm. The file is read with the command `rlreaddimacs "filename"`. This command also functions for both SAT and Q-SAT problems. We can read our last example and e.g. use quantifier elimination to solve it:

```
1: load redlog;

2: rlset ibalp;

*** turned off switch raise

{}

3: f := rlreaddimacs "test.qdimacs";
Reading 3 variables and 4 clauses
Q-SAT: Reading quantifiers
Deleted Redundant Clauses: 0

f := ex var2 all var3 ex var1 ((var1 = 1 or var3 = 1) and (var3 = 0 or var2 = 1)

 and (var2 = 0 or var3 = 0 or var1 = 1) and var2 = 1)
```

B.2 Options for the SAT Solver

As demonstrated in chapter 2 there are some options for the SAT solver. These options can be changed within the REDLOG system before each run of the solver. They are set to default values when the ibalp package is loaded the first time. Table B.1 gives an overview of the options and their values.

Option	Domain	Default	Description
Branching heuristic	{ *zmom*, *activity*, *dlcs* }	zmom	branching heuristic for variable selection
First restart after learnt clauses	$\mathbb{N}$	5	restart of the solver after a certain number of learnt clauses
First variable assignment	{0,1}	1	the value the variables are assigned in the first run (flipped every restart)
Increase factor for restarts	$\mathbb{R}^+$	1.2	increase the number of learnt clauses for a restart over time
Clause deletion after learnt clauses	$\mathbb{N}$	200	delete inactive clauses after a certain number of leanrt clauses

Table B.1: The options for the SAT solver

One can change the standard values with the command `rlqsatoptions`. The argument is a list of the 5 options given in the table. When the command is executed with the empty list `{}`, the current configuration is returned. When a new configuration is made, the new configuration is returned. This can be helpful to save and load configurations as demonstrated in the following example:

```
3: conf1 := rlqsatoptions {zmom,5,1,2,200};

conf1 := {zmom,5,1,2,200}

4: conf2 := rlqsatoptions {activity,10,0,2,500};

conf2 := {activity,10,0,2,500}

5: rlqsatoptions {};

{activity,10,0,2,500}

6: rlqsatoptions conf1;

{zmom,5,1,2,200}

7: rlqsatoptions {};

{zmom,5,1,2,200}
```

There is also a syntax checking for new options in order to avoid failures. The options have only impact on the SAT solver of REDLOG. The Q-SAT engine uses a fixed heuristic and does not have any features like restarts or clause deletion.

List of Algorithms

1 A basic version of the DLL algorithm 8
2 The basic Unit Propagation algorithm 8
3 Local search of Schöning's probabilistic k-SAT algorithm 10
4 Setting a variable to $\top$ 17
5 Unsetting a variable from $\top$ 18
6 The CDCL algorithm 22
7 The analyseConflict procedure 23
8 The CDCL procedure for Q-SAT 39
9 The analyseConflict procedure for Q-SAT 40
10 The analyseSAT procedure for Q-SAT 40
11 The algorithm for parametric Q-SAT 48

List of Figures

1.1 Search tree for a SAT problem 9

2.1 Situation without clause learning 21
2.2 Example: Deriving the empty clause 24
2.3 Example: The new situation after backtracking 25

3.1 Search tree for a Q-SAT problem 36
3.2 The complexity hierarchy 37

4.1 Search tree for a parametric Q-SAT problem 48
4.2 Small benchmark results parametric Q-SAT 50
4.3 Middle benchmark results parametric Q-SAT 50
4.4 Large benchmark results parametric Q-SAT 51

List of Tables

1.1 Values for t in dependence of k . . . 11

2.1 REDLOG commands . . . 14
2.2 The fields of a variable . . . 15
2.3 The fields of a clause . . . 16
2.4 Benchmark 1 for testing the branching heuristics . . . 20
2.5 CPU time in seconds for large examples . . . 21
2.6 The AIM benchmark suite . . . 29
2.7 The II benchmark suite . . . 29
2.8 Correct model verification of DLX processors . . . 32
2.9 The configuration of the SAT Solver for the benchmarks . . . 32
2.10 Benchmark results for small examples (time in sec) . . . 33
2.11 Benchmark results for large examples (time in sec) . . . 33

3.1 The additional fields of a variable for Q-SAT . . . 38
3.2 Benchmark results for the bomb in the toilet problem (time in sec) . . . 44
3.3 Benchmark results for the 2 player game (time in sec) . . . 44
3.4 Benchmark results for the chain benchmark (time in sec) . . . 45

B.1 The options for the SAT solver . . . 56

References

A.P. Kamath, N.K. Karmarkar, K.G. Ramakrishnan, & Resende, M.G.C. 1992. A continuous approach to inductive inference. *Mathematical programming*, **57**, 215–238.

Bacchus, F., & Winter, J. 2003. Effective preprocessing with hyper-resolution and equality reduction. *Pages 341–355 of: Sixth international symposium on theory and applications of satisfiability testing.*

Beame, Paul, Kautz, Henry A., & Sabharwal, Ashish. 2004. Towards understanding and harnessing the potential of clause learning. *J. artif. intell. res. (JAIR)*, **22**, 319–351.

Cook, Stephen A. 1971. The complexity of theorem-proving procedures. *Pages 151–158 of: STOC '71: Proceedings of the third annual ACM symposium on theory of computing.* New York, NY, USA: ACM Press.

Davis, Martin, Logemann, George, & Loveland, Donald. 1962. A machine program for theorem-proving. *Commun. ACM*, **5**(7), 394–397.

Dolzmann, Andreas, & Sturm, Thomas. 1997. Redlog: Computer algebra meets computer logic. *ACM SIGSAM bulletin*, **31(2)**, 2–9.

Dolzmann, Andreas, & Sturm, Thomas. 1999. *Redlog user manual.*

En, Niklas, & Srensson, Niklas. 2003. An extensible SAT-solver. *Pages 502–518 of:* Giunchiglia, Enrico, & Tacchella, Armando (eds), *SAT*. Lecture Notes in Computer Science, vol. 2919. Springer.

Fujiwara, H., & Toida, S. 1982. The complexity of fault detection problems for combinational logic circuits. *IEEE trans. comput.*, **31**(6), 555–560.

Marques-Silva, João. 1999. The impact of branching heuristics in propositional satisfiability algorithms.

Mitchell, David G. 2005. A SAT solver primer. *Pages 112–133 of: EATCS bulletin (the logic in computer science column)*, vol. 85.

Moskewicz, Matthew W., Madigan, Conor F., Zhao, Ying, Zhang, Lintao, & Malik, Sharad. 2001. Chaff: Engineering an Efficient SAT Solver. *In: Proceedings of the 38th design automation conference (DAC'01).*

Schoning, Uwe. 1999. A probabilistic algorithm for k-SAT and constraint satisfaction problems. *Pages 410–414 of: Proc. 40th sympos. on foundations of computer science.*

Seidl, Andreas, & Sturm, Thomas. 2003. Boolean quantification in a first-order context. *Pages 329–345 of:* Ganzha, V.G., Mayr, E.W., & Vorozhtsov, E.V. (eds), *Proceedings of the 6th international workshop on computer algebra in scientific computing, CASC'2003 (passau, germany, september 20–26, 2003).* Garching: Institut für Informatik, Technische Universität München.

Silva, J., & Sakallah, K. 1996. *Conflict analysis in search algorithms for satisfiability.*

Y. Asahiro, K. Iwama, & Miyano, E. 1996. Random generation of test instanzes with controlled attributes. *Cliques, coloring, and satisfiability: The second DIMACS implementation challenge*, **26**, 377–394.

Zhang, L., & Malik, S. 2003a. Cache performance of SAT solvers: A case study for efficient implementation of algorithms. *In: Sixth international conference on theory and applications of satisfiability testing.*

Zhang, Lintao, & Malik, Sharad. 2002a. Conflict driven learning in a quantified boolean satisfiability solver. *Pages 442–449 of: ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on computer-aided design.* New York, NY, USA: ACM.

Zhang, Lintao, & Malik, Sharad. 2002b. Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. *Pages 200–215 of: CP '02: Proceedings of the 8th international conference on principles and practice of constraint programming.* London, UK: Springer-Verlag.

Zhang, Lintao, & Malik, Sharad. 2003b. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. *Page 10880 of: DATE '03: Proceedings of the conference on design, automation and test in europe.* Washington, DC, USA: IEEE Computer Society.