Decision Procedures for Propositional Satisfiability and beyond

Propositional Solvers and their extensions for
Knowledge Representation and Formal Verification

Armando Tacchella

A dissertation in
Electronics and Computer Science Engineering

Thesis Supervisors: Chiar.mo Prof. Mauro Di Manzo
Chiar.mo Prof. Enrico Giunchiglia

Dipartimento di Informatica, Sistemistica e Telematica
Università degli Studi di Genova.

Contents

Chapter 1

Introduction

1.1 Research area, motivations and goals

1.2 Contribution of the thesis

1.3 Structure of the thesis

Part I

Propositional Satisfiability (SAT)

Chapter 2

Solving the satisfiability (SAT) problem

2.1 Propositional logic

2.2 Algorithms for SAT

2.3 The Davis-Logemann-Loveland (DLL) algorithm

- open literal
- unit propagation
- pure literal
- left, right split
- failed literal
- search tree, deep and shallow, depth of an assignment

Chapter 3

A survey of relevant contributions

3.1 Enhancing the basic DLL algorithm

3.1.1 Heuristics

3.1.2 Relaxations

3.1.3 Advanced look-back techniques

3.1.4 Advanced look-ahead techniques

3.1.5 Randomization

3.1.6 Preprocessing

3.2 DLL-based SAT solvers

3.2.1 POSIT

3.2.2 SATO

3.2.3 SATZ and EqSATZ

3.2.4 RelSAT

3.2.5 Boehm

3.2.6 GRASP

3.3 Experimental analysis

3.3.1 A taxonomy of available benchmarks

3.3.2 Designing a test set

3.3.3 A snapshot of DLL-based SAT solvers

Chapter 4

Anatomy of a SAT solver

In this chapter we present the basic components of a DLL-based SAT solver, from the data structure definitions to the most advanced look-back techniques, concluding with a new synthesis of the DLL algorithm closer to the real implementation of a SAT solver. In more detail, in section 4.1 we define all the basic components of the data structure and the conditions that they must obey; we present the primitive to extend the valuation of the propositions to the formula, and its counterpart, i.e., the primitive to *retract* the valuation of the propositions from the formula, restoring the conditions previous to the assignment. We present two such assign-retract pairs, one working in a stack-wise fashion (the last truth value extended is the first to be retracted) and the other one supporting dyamic reordering in the sequence of assignments. In sections 4.2 and 4.3 we present the most common look-ahead and look-back-techniques, introducing, when necessary, modifications to the basic data structure and primitives. Section 4.4 is devoted to relaxations ... Finally, in section 4.5 we conclude the picture by showing how the DLL algorithm can be implemented on top the data structures and primitives introduced in this chapter.

The conventions that we use to present data structures and algorithms are those of [CLR98] with some minor modifications/extensions for the sake of better clarity and compactness. We now recall the conventions briefly, pointing out any difference w.r.t. [CLR98].

1. Indentation indicates block structure: the body of functions, loops and **if-then-else** statements is formatted accordingly. We use and indentation schema (below on the right) that is slighlty different than the one of [CLR98] (below on the left):

while (test)	**while** (test) **do**
do (statement-1)	(statement-1)
...	...
(statement-n)	(statement-n)

   Also, when no confusion may arise, we abuse the notation and write **if-then-else** consisting of a *single* statement per block arranged in one or two lines like this:

if (test) **then** (statement-1) **else** (statement-2)

if (test) **then** (statement-1)
else (statment-2)

2. The looping constructs **while**, **for**, and **repeat** and the conditional constructs **if**, **then**, and **else** have the same interpretation as in Pascal.
3. The symbol "$\triangleright$ " indicates that the remainder of the line is a comment.
4. A multiple assignment of the form $i \leftarrow j \leftarrow e$ assigns to both variables i and j the value of the expression e, i.e., multiple assignments associate from the right.
5. Variables are local to the given procedure. We do not use global variables in the definition of our algorithms, i.e., all the procedures described are reentrant. Local variables are not declared and we expect them to be uninitialized.
6. Array elements are accessed by specifying the array name followed by the index in square brackets: $A[i]$ is the i-th element of the array A.
7. Compound data are organized into *objects* which are comprised of *fields*. Objects may have several *instances*, and a particular field is accessed using the field name followed by the name of the instance in square brackets. For example, the object *list* has the fields *car*, *cdr* and given an instance *my-list*, *car*[*my-list*] and *cdr*[*my-list*] access the corresponding fields. Variables representing arrays and objects are treated as *pointers* to those objects. The constant NIL denotes the null pointer.
8. Parameters are passed to a procedure *by value*: the called procedure receives its own copy of the the parameters. In case of object instances and arrays the *pointer* gets copied, so any change to the array elements or the instance fields is seen by the caller. We will use the keyword **var** to denote when parameters are passed *by reference*

We now spend a few words about arrays which play a special rôle in our procedures. As in [CLR98], for each array A we define the field *length*[A] that specifies the number of elements in A. Now, besides the usual direct-access primitives, we wish to use arrays also as stacks and/or lists. The approach resembles the one of [Str97] and the way arrays are implemented in the Standard Template Library (see the template class `vector`, chapter 16 of [Str97]). For each array A we define the following primitives:

PUSH(A, e)
1 $i \leftarrow$ *length*[A] $\leftarrow$ *length*[A] + 1
2 $A[i] \leftarrow e$
3 **return**

POP(A)
1 $i \leftarrow$ *length*[A]
2 *length*[A] $\leftarrow$ *length*[A] - 1
3 **return** $A[i]$

```
TOP(A)
 1 i ← length[A]
 2 return A[i]

FLUSH(A)
 1 length[A] ← 0
 2 return
```

which operate an array as a stack. Notice that each primitive runs in $O(1)$ since we add elements to the back of the array. This upper bound is always valid for POP, TOP and FLUSH, while in the case of PUSH we are assuming to have always enough room to add the elements. The primitive:

```
DELETE(A, i)
 1 bi ← length[A]
 2 A[i] = A[bi]
 3 length[A] ← length[A] - 1
 4 return
```

perfoms a typical list operation (a deletion) using an array and an $O(1)$ algorithm. Clearly, the procedure is not *stable*, i.e., after a deletion the contents of the array are not in the same order as they were before.

4.1 Data structure and primitives

In this section we present the (basic) data structure and the primitives to extend/retract valuations of the propositions. The data structure and the primitive EXTEND-PROP are heavily inspired to those presented in in [Fre95] about the DLL-based solver POSIT. Our implementation of RETRACT-PROP is inspired to analogous techniques that we find in state-of-the-art DLL-based solvers like RelSAT [BS97] and Satz [LA97]. The extensions ofEXTEND-PROP and RETRACT-PROP to algorithms that assign and retract values without following the stack order, is an original contribution of our work as far as we know.

4.1.1 Formula and search state

Let UN, PP, PN, RS, FL be five new constants. A proposition can then be described by the following object:

id the proposition unique identifier;

value the value of the proposition (either TRUE , FALSE , or UNDEF);

mode the mode of assignment (either UN, PP, PN, RS, or FL);

level the assignment level;

Pos an array of clauses for positive occurrences

Neg the same for negative occurrences.

Initially, for each proposition instance pi the following holds true:

$$value[pi] = \text{UNDEF} \tag{4.1}$$

$$(length[Pos[pi]] \neq 0) \vee (length[Neg[pi]] \neq 0) \tag{4.2}$$

Intuitively, the conditions state that each proposition is initially open (4.1) and it occurs at least in one clause, either positively or negatively (4.2). We require that the arrays $Pos[pi]$ and $Neg[pi]$ are indeed *sets* of clauses. So we state the following condition for each proposition instance pi:

$$\forall i \forall j ((i \neq j) \supset Pos[pi][i] \neq Pos[pi][j]) \tag{4.3}$$

where $1 < i < length[Pos[pi]]$, $1 < j < length[Pos[pi]]$, and the symmetric one:

$$\forall i \forall j ((i \neq j) \supset Neg[pi][i] \neq Neg[pi][j]) \tag{4.4}$$

where $1 < i < length[Neg[pi]]$, $1 < i < length[Neg[pi]]$. We also require that Pos and Neg are non-overlapping, but this can be enforced more elegantly when specifying the requirements for the clause instances (see below).

A clause can be described by the following object:

id the clause unique identifier;

open the number of open literals;

sub a pointer to a proposition;

Lits an array of literals, i.e., propositions with sign.

Given an index i such that $1 < i < length[Lits]$, $prop[Lits[i]]$ is a proposition instance and $sign[Lits[i]]$ is the sign of the occurrence: TRUE for positive occurrences and FALSE for negative ones. Initially, for each clause instance cli the following conditions hold:

$$sub[cli] = \text{NIL} \tag{4.5}$$

$$open[cli] = length[Lits[cli]] \tag{4.6}$$

$$open[cli] \neq 0 \tag{4.7}$$

In words, each clause is open (4.5), each literal in the clause is open (4.6), and the clause itself is not empty (4.7). We require that clause instances do not contain the same proposition twice, so for each instance cli:

$$\forall i \forall j ((i \neq j) \supset prop[Lits[cli][i]] \neq prop[Lits[cli][j]] \tag{4.8}$$

where $1 < i < length[Lits[cli]]$, $1 < j < length[Lits[cli]]$. Enforcing condition (4.8) preserves the generality of the algorithm since in propositional logic we have the facts $p \vee p \equiv p$ and $p \vee \neg p \equiv \top$: duplicate literals can be removed to yield an equivalent clause, and complementary literals result in a tautological clause which can be removed from the set of clauses to yield an equivalent set. Notice that (4.8), together with (4.3) and (4.4), also implies that $Pos[pi]$ and $Neg[pi]$ are pairwise disjoint sets for each proposition instance pi.

As a final requirement, we ensure that the combination of clauses and propositions instances correctly represents propositional formulas. Given a CNF formula φ, with $cl_1 \ldots cl_m$ clauses and $p_1 \ldots p_n$ propositions we impose that:

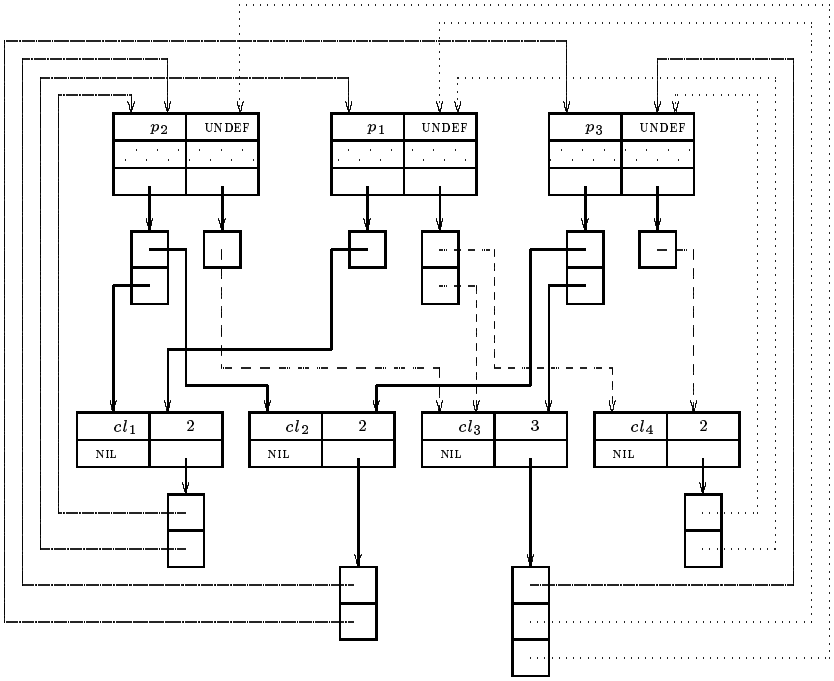

Figure 4.1: Data structure for the formula (4.9)

- for each $p \in \{p_1 \dots p_n\}$ there is a corresponding proposition instance pi which fullfills the conditions from (4.1) to (4.4) and such that the contents of $Pos[pi]$ and $Neg[pi]$ are exacty those of φ: for each positive (negative) occurrence of the proposition p in a clause $cl \in \{cl_1 \dots cl_m\}$ there exists $cli = Pos[pi][k]$ ($cli = Neg[pos][k]$) such that cli is the clause instance corresponding to cl; there is no $Pos[pi][k]$ ($Neg[pos][k]$) such that its corresponding clause does not have a positive (negative) occurrence of p;
- for each $cl \in \{cl_1 \dots cl_m\}$ there is a corresponding clause instance cli, which fullfills the conditions from (4.5) to (4.8) and such that the contents of $Lits[cli]$ are exactly those of cl: for every literal $l \in cl$ there exists $lo = Lits[cli][k]$ such that $prop[lo]$ is the proposition instance corresponding to $p = |l|$ and $sign[lo]$ is TRUE if $l = p$ and FALSE otherwise; there is no $Lits[cli][k]$ such that the literal corresponding to it does not occur in cl.

Consider the following formula in CNF:

$$\begin{array}{lcl} cl_1 & : & p_1 \vee p_2 \\ cl_2 & : & p_2 \vee p_3 \\ cl_3 & : & \neg p_1 \vee \neg p_2 \vee p_3 \\ cl_4 & : & \neg p_1 \vee \neg p_3 \end{array} \tag{4.9}$$

In figure 4.1 we show how this formula is represented, according to the objects and the conditions that we defined to represent clauses and propositions. In the figure, going from left to right, top to bottom, we first encounter the proposition instances, each one represented as a box with six divisions, corresponding to the

six fields of the objects. The two top-row cells in each box correspond to the fields *id* and *value*, which have been properly inizialized for each proposition instance: *id* contains an unique identifier (we used the very same propositional logic symbol) and *value* contains UNDEF for all the propositions according to condition (4.1). The two middle-row fields in the boxes represent *mode* and *level* respectively: they are shaded since we did not make any assumption on their initial value. The bottom-row fields represent *Pos* and *Neg* respectively: each one is an array of clause objects, and we depicted them as a sequence of cells, an arrow going out from each cell and pointing to the corresponding clause instance. To prevent clutter in the picture we used solid lines for *Pos* instances and dashed lines for *Neg* instances. For example, we see in (4.9) that p_2 occurs three times in the formula, twice positively (in cl_1 and cl_2) and once negatively (in cl_3). Accordingly, in the instance representing p_2, the two arrows departing from the field *Pos* are incident to the clause instances representing cl_1 (first from the left) and cl_2 (second from the left), while the single arrow departing from the field *Neg* is incident to the clause instance representing cl_3 (third from the left). Going back to our description of figure 4.1, we encounter the clause instances, each one represented as a box with four divisions, again corresponding to the fields of the object. In the top-row sit the fields *id* and *open*: for the first one, we use the same symbol as in example (4.9), while the second one is initialized to the length of each clause as stated in (4.6). In the bottow-row sit the fields *sub* and *Lits*, the first one initialized to NIL according to (4.5) and the second one holding the literals that occur in the clause. The array of literals is depicted as a sequence of cells, an arrow departing from each one and pointing the proposition instances corresponding to the propositions occurring in the clause. We used two different traits for the arrows, an almost-solid one and a dotted one: the reason of this goes beyond clutter prevention, and it denotes that we can somehow manage to encode the sign of the proposition instances being referenced without using an additional field for each element of *Lits*. In more detail, given a generic *pi* and i such that $1 < i < length[Lits[pi]]$, we have that $prop[Lits[pi][i]]$ and $sign[Lits[pi][i]]$ can be encoded in the same data field. The cautious reader may notice here the lack of symmetry between proposition instances, where we chose to have two separate arrays to encode the sign of the occurrences, and the clause instances, where we did with just one array plus some trick to encode the sign. The reason of this dicotomy will be clarified in the remaninder of this chapter, but we anticipate that this choice allows for much more elegant and slightly more efficient basic algorithms to be implemented.

Besides proposition and clause objects, we need some additional structure to effectively encode a propositional CNF formula and the associated search state. The ensemble of these two elements, that we define simply as *state*, can be described by the following object:

Props an array of propositions;

Clauses an array of clauses.

Stack a stack of propositions;

Unit a stack of clauses;

level the current search depth;

open the current number of open clauses.

Initially, the following conditions must hold for each state instance si:

$$length[Stack[si]] = 0 \quad (4.10)$$
$$level[si] = 0 \quad (4.11)$$
$$open[si] = length[Clauses[si]] \quad (4.12)$$

In words, the search stack is initially empty (4.10), the search did not begin yet (4.10), the number of open clauses coincides with the number of initial clauses (4.12). Initially, as well as during the search, the following requirements hold for each state instance si:

$$length[Props[si]] \neq 0 \quad (4.13)$$
$$length[Clauses[si]] \neq 0 \quad (4.14)$$

which mean that there is at least one proposition (4.13) and at least one clause (4.14), and:

$$\forall i \forall j((i \neq j) \supset id[Props[si][i]] \neq id[Props[si][j]] \quad (4.15)$$

where $1 < i < length[Props[fo]]$, $1 < j < length[Props[fo]]$. In words, we forbid to the same proposition instance to be stored twice in the array *Props*. Of course, we can have two different propositions that occur exactly in the same clauses with exactly the same sign, but this will do no harm. The same can be said for clauses, since adding the same clause twice decreases efficiency, but it does not jeopardize correctness. Finally, for each instance si we require that $Unit[si]$ contains all the clauses having one open literal:

$$\forall i \exists j((open[Clauses[si][i]] = 1) \supset (Unit[si][j] = Clauses[si][i])) \quad (4.16)$$

where $1 < i < length[Clauses[si]]$, $1 < j < length[Unit[si]]$ and that all the clauses contained in $Unit[si]$ have only one open literal:

$$\forall i\, open[Unit[si][i]] = 1 \quad (4.17)$$

where $1 < i < length[Unit[si]]$. Requirements (4.16) and (4.17) ensure that all unit clauses are properly detected and that the unit clause stack indeed contains unit clauses only. Notice that it is possible for the same clause to appear twice or more in the unit stack, but this is handled transparently by the look-ahead algorithm (see section 4.2 in this chapter).

In the following, when we speak about propositions, clauses, and (search) state it will be clear that we refer to the corresponding proposition, clause and state instances respectively. To denote instances we will use the identiers p, cl, s without the additional "i" that we used in this section to distinguish between the logic objects and the data structure elements. Given a state s and a proposition identifier pid we assume the existence of a primitive ID-REV(s, pid) which returns the unique proposition instance such that $id[p] = pid$.

4.1.2 Assigning and retracting truth values

In figure 4.2 we present the basic version of the primitive EXTEND-PROP-TRUE. As the name suggests, the primitive extends the valuation TRUE of a proposition

```
EXTEND-PROP-TRUE(s, p, m)
 1 ▷ Performing unit resolutions
 2 for i ← 1 to length[Neg[p]] do
 3     cl ← Neg[p][i]
 4     if sub[cl] = NIL then
 5         open[cl] ← open[cl] - 1
 6         if open[cl] = 1 then
 7             PUSH(Unit[s], cl)
 8         else if open[cl] = 0 then
 9             for j ← i downto 1 do
10                 cl ← Neg[p][i]
11                 if sub[cl] = NIL then
12                     open[cl] ← open[cl] + 1
13             return FALSE
14 ▷ Setting proposition fields
15 value[p] ← TRUE
16 mode[p] ← m
17 level[p] ← level[s]
18 PUSH(Stack[s], p)
19 ▷ Performing unit subsumptions
20 for i ← 1 to length[Pos[p]] do
21     cl ← Pos[p][i]
22     if sub[cl] = NIL then
23         sub[cl] ← p
24         open[s] ← open[s] - 1
25 return TRUE
```

Figure 4.2: Extending the state s with the valuation of the proposition p.

p to the current state s: it returns TRUE if successfull (no empty clause was detected) and FALSE otherwise. Additionally, the primitive sets the mode of propagation m according to the value supplied by the caller. As we shall see, the twin primitive EXTEND-PROP-FALSE is obtained by the one presented in figure 4.2 with symmetry arguments. Therefore we describe in detail EXTEND-PROP-TRUE only, but exactly the same considerations apply to EXTEND-PROP-FALSE.

In figure 4.2 we describe the procedure EXTEND-PROP-TRUE

- In lines 1-13 the primitive performs unit resolutions, i.e., for each open clause bearing a negative occurrence of the proposition p, the number of open literals is decreased by one (lines 2-5): if the clause becomes unary, it is pushed on the unit clauses stack (line 7); if the clause becomes empty, all the open literal counters changed so far are restored (lines 9-12) and the procedure returns FALSE (line 13).

- In lines 14-18 the primitive does some clerical work, setting the fields of the proposition and pushing it in the search stack (line 18).

- In lines 19-24 the primitive performs unit subsumptions, i.e., each open clause bearing a positive occurence of the proposition p is "freezed" by

```
RETRACT-PROP-TRUE(s, p)
 1 ▷ Resetting proposition fields
 2 value[p] ← UNDEF
 3 POP(Stack[s], p)
 4 ▷ Retracting unit resolutions
 5 for i ← 1 to length[Neg[p]] do
 6     cl ← Neg[p][i]
 7     if sub[cl] = NIL then
 8         open[cl] ← open[cl] + 1
 9 ▷ Retracting unit subsumptions
10 for i ← 1 to length[Pos[p]] do
11     cl ← Pos[p][i]
12     if sub[cl] = p then
13         sub[cl] ← NIL
14         open[s] ← open[s] + 1
15 return
```

Figure 4.3: Retracting the valuation of the proposition p from the state s.

storing p in the field *sub* (line 23); each time an open clause becomes satisfied, we decrement $open[s]$, the total number of open clauses in the current state (line 24)

- the primitive returns TRUE (line 25)

To obtain EXTEND-PROP-FALSE it is sufficient to swap lines 2-3 with lines 20-21 and to change line 15 to assign FALSE instead of TRUE , the remainder of the code being exactly the same.

In figure 4.3 we present the basic version of the primitive RETRACT-PROP-TRUE. As the name suggests, the primitive retracts the valuation TRUE of a proposition p from the current state s: it restores the state in exactly the same conditions it was before the propagation with EXTEND-PROP-TRUE occurred. As we shall see, the twin primitive RETRACT-PROP-FALSE is obtained by the one presented in figure 4.3 with symmetry arguments. Therefore we describe in detail RETRACT-PROP-TRUE only, but exactly the same considerations apply to RETRACT-PROP-FALSE.

In figure 4.3 we describe the procedure EXTEND-PROP-TRUE.

- In lines 1-3 the primitive does some clerical work, resetting the fields of the proposition and popping it from the search stack (line 3).
- In lines 4-8 the primitive retracts unit resolutions, i.e., for each open clause bearing a negative occurrence of the proposition p, the number of open literals is increased by one.
- In lines 9-14 the primitive retracts unit subsumptions, i.e., each clause that was satisfied by p is forced open; each time an open clause becomes open, we increment $open[s]$, the total number of open clauses in the current state (line 24)

```
EXTEND-PROP(s, p, v, m)
 1 if v = TRUE then
 2    EXTEND-PROP-TRUE(s, p, m)
 3 else
 4    EXTEND-PROP-FALSE(s, p, m)

RETRACT-PROP(s, p)
 1 if value[p] = TRUE then
 2    RETRACT-PROP-TRUE(s, p)
 3 else
 4    RETRACT-PROP-FALSE(s, p)
```

Figure 4.4: EXTEND-PROP and RETRACT-PROP choose, respectively, the value to propagate and to retract for the proposition p in the state s.

To obtain RETRACT-PROP-FALSE it is sufficient to swap lines 5-6 with lines 10-11, the remainder of the code being exactly the same.

In figure 4.4 we define EXTEND-PROP and RETRACT-PROP whose task is to call the appropriate primitive to do the job. In the case of EXTEND-PROP is the caller that determines the value of the propagation with the parameter v; for RETRACT-PROP the value is already stored in the omonimous field, so the user does not even need to supply it. The remainder of the parameters to be supplied is teh same for both functions.

To show how EXTEND-PROP and RETRACT-PROP work, we rewrite the example 4.9 as follows:

$$\begin{array}{lcl} (2, \text{NIL}) & : & p_1 \vee p_2 \\ (2, \text{NIL}) & : & p_2 \vee p_3 \\ (3, \text{NIL}) & : & \neg p_1 \vee \neg p_2 \vee p_3 \\ (2, \text{NIL}) & : & \neg p_1 \vee \neg p_3 \end{array} \tag{4.18}$$

where we show the pair (*open*, *sub*) with the current value of the fields for each clause. Given f as the state corresponding to the formula, the effect of EXTEND-PROP(f, ID-REV(f, p_1), TRUE ,LS) [1] is the following:

$$\begin{array}{lcl} (2, \text{ID-REV}(f, p_1)) & : & p_1 \vee p_2 \\ (2, \text{NIL}) & : & p_2 \vee p_3 \\ (2, \text{NIL}) & : & \neg p_1 \vee \neg p_2 \vee p_3 \\ (1, \text{NIL}) & : & \neg p_1 \vee \neg p_3 \end{array} \tag{4.19}$$

If p_1 is true then $p_1 \vee p_2$ is satisfied: EXTEND-PROP marked the clause with proposition corresponding to p_1. If p_1 is true, then $\neg p_1 \vee \neg p_2 \vee p_3 \equiv \neg p_2 \vee p_3$ and $\neg p_1 \vee \neg p_3 \equiv \neg p_3$: EXTEND-PROP decreased the number of open literals in both clauses. Notice that $length[Unit[f]] = 1$ because the last clause become unary. If we try to propagate p_3 to true with EXTEND-PROP(f, ID-REV(f, p_3), TRUE ,LS) the net effect is to leave the formula as in 4.19 but the return value is

[1] The choice of a mode here is totally irrelevant: we opted for LS but we could use any other mode at this point.

FALSE since propagating p_3 to true yields an empty clause. Therefore, it is not necessary to retract the proposition that caused the contradiction because this is done by EXTEND-PROP. So we propagate p_3 to false with EXTEND-PROP(f, ID-REV(f, p_3), FALSE ,LS) to yield:

$$\begin{array}{lcl} (2, \text{ID-REV}(f, p_1)) & : & p_1 \vee p_2 \\ (1, \text{NIL}\ ) & : & p_2 \vee p_3 \\ (2, \text{NIL}\ ) & : & \neg p_1 \vee \neg p_2 \vee p_3 \\ (1, \text{ID-REV}(f, p_3)) & : & \neg p_1 \vee \neg p_3 \end{array} \tag{4.20}$$

The last clause is freezed, and the second one is now equivalent to p_2. If, by mistake, we now try to retract the assignment of p_1 with EXTEND-PROP(f, ID-REV(f, p_1)) we yield:

$$\begin{array}{lcl} (2, \text{NIL}\ ) & : & p_1 \vee p_2 \\ (1, \text{NIL}\ ) & : & p_2 \vee p_3 \\ (3, \text{NIL}\ ) & : & \neg p_1 \vee \neg p_2 \vee p_3 \\ (1, \text{ID-REV}(f, p_3)) & : & \neg p_1 \vee \neg p_3 \end{array} \tag{4.21}$$

which is only apparently correct, because retracting p_3 will bring us in an inconsistent situation:

$$\begin{array}{lcl} (2, \text{NIL}\ ) & : & p_1 \vee p_2 \\ (1, \text{NIL}\ ) & : & p_2 \vee p_3 \\ (3, \text{NIL}\ ) & : & \neg p_1 \vee \neg p_2 \vee p_3 \\ (1, \text{NIL}\ ) & : & \neg p_1 \vee \neg p_3 \end{array} \tag{4.22}$$

Needless to say, if we had applied RETRACT-PROP in stack-wise order retracting p_3 first and the p_1, we would have restored the initial state correctly. It is to allow the user to retract propositions without respecting the stack order, that in the next subsection we present improved algorithms for extending/retracting truth values.

4.1.3 Stackwise vs. non-stackwise operation

As we outlined before, the primitives EXTEND-PROP and RETRACT-PROP presented in the previous subsection are suited for stack-wise operation only. In examples 4.18-4.21 we confirmed that using the primitives in the wrong way brings us to inconsistent states. In order to solve this, we need a minor modification of the clause objects defined in 4.1.1. Precisely, the field *sub* becomes now a stack *Sub*, its initial condition being for any clause *cl*:

$$length[Sub[cl]] = 0 \tag{4.23}$$

In stack-wise operations we need to record only the first literal that satisfies the clause. Indeed, many literals in the clause may be contemporarily assigned to TRUE in the current state. In case of stack-wise operations this does not matter: assignments are retracted in reverse order and each literal that satisfied the clause *after* the first one will be retracted *before* we retract the first one. This is not true when we do not follow the stack order. In this case we need to know, when retracting a proposition, if some other proposition in the current assignment is still satisfying the clause.

```
DB-Extend-Prop-true(s, p, m)
 1 ▷ Performing unit resolutions
 2 for i ← 1 to length[Neg[p]] do
 3     cl ← Neg[p][i]
 4     open[cl] ← open[cl] - 1
 5     if open[cl] = 1 then
 6         Push(Unit[s], cl)
 7     else if open[cl] = 0 then
 8         for j ← i downto 1 do
 9             cl ← Neg[p][i]
10             open[cl] ← open[cl] + 1
11         return false
12 ▷ Setting proposition fields
13 value[p] ← true
14 mode[p] ← m
15 level[p] ← level[s]
16 Push(Stack[s], p)
17 ▷ Performing unit subsumptions
18 for i ← 1 to length[Pos[p]] do
19     cl ← Pos[p][i]
20     if length[Sub[cl]] = 0 then
21         open[s] ← open[s] - 1
22     Push(Sub[cl], p)
23 return true
```

Figure 4.5: Redefining Extend-Prop-true for non-stackwise operation.

In figure 4.5 we present the basic version of the primitive DB-Extend-Prop-true, i.e., the modification of Extend-Prop-True to handle extend operations in algorithms that do not follow the stack order. As in the case of Extend-Prop-true the primitive extends the valuation true of a proposition p to the current state s: it returns true if successfull (no empty clause was detected) and false otherwise. Additionally, the primitive sets the mode of propagation m according to the value supplied by the caller. Again, the twin primitive DB-Extend-Prop-false is obtained by the one presented in figure 4.5 with symmetry arguments. Therefore we describe in detail Extend-Prop-true only, but exactly the same considerations apply to Extend-Prop-false.

In figure 4.5 we describe the procedure Extend-Prop-true.

- In lines 2-11 the primitive performs unit resolutions, i.e., for each clause bearing a negative occurrence of the proposition p, the number of open literals is decreased by one (lines 2-4): if the clause becomes unary, it is pushed on the unit clauses stack (line 6); if the clause becomes empty, all the open literal counters changed so far are restored (lines 7-10) and the procedure returns false (line 11).

- In lines 12-16 the primitive does some clerical work, setting the fields of the proposition and pushing it in the search stack (line 18).

```
DB-RETRACT-PROP-TRUE(s, p)
 1 ▷ Resetting proposition fields
 2 value[p] ← UNDEF
 3 DELETE(Stack[s], p)
 4 ▷ Retracting unit resolutions
 5 for i ← 1 to length[Neg[p]] do
 6     cl ← Neg[p][i]
 7     open[cl] ← open[cl] + 1
 8 ▷ Retracting unit subsumptions
 9 for i ← 1 to length[Pos[p]] do
10     cl ← Pos[p][i]
11     DELETE(Sub[cl], p)
12     if length[Sub[cl]] = 0 then
13         open[s] ← open[s] + 1
14 return
```

Figure 4.6: Redefining RETRACT-PROP-TRUE for non-stackwise operation.

- In lines 17-22 the primitive performs unit subsumptions, i.e., for each clause bearing a positive occurence of the proposition p, p itself is pushed in the stack Sub; each time an open clause becomes satisfied for the first time, we decrement $open[s]$, the total number of open clauses in the current state (lines 20-21)

- the primitive returns TRUE (line 23)

To obtain EXTEND-PROP-FALSE it is sufficient to swap lines 2-3 with lines 18-19 and to change line 15 to assign FALSE instead of TRUE , the remainder of the code being exactly the same.

In figure 4.6 we present the basic version of the primitive DB-RETRACT-PROP-TRUE, i.e., the modification of RETRACT-PROP-TRUE to handle retract operations in algorithms that do not follow the stack order. As in the case of RETRACT-PROP-TRUE the primitive retracts the valuation TRUE of a proposition p from the current state s: it restores the state in a consistent condition as if the propagation with DB-EXTEND-PROP-TRUE never occurred. Again, the twin primitive DB-RETRACT-PROP-FALSE is obtained by the one presented in figure 4.6 with symmetry arguments. Therefore we describe in detail RETRACT-PROP-TRUE only, but exactly the same considerations apply to RETRACT-PROP-FALSE.

In figure 4.6 we describe the procedure DB-RETRACT-PROP-TRUE.

- In lines 12-16 the primitive does some clerical work, resetting the fields of the proposition and deleting it from the search stack (line 18).

- In lines 4-7 the primitive retracts unit resolutions, i.e., for each clause bearing a negative occurrence of the proposition p, the number of open literals is increased by one.

- In lines 8-13 the primitive retracts unit subsumptions, i.e., for each clause bearing a positive occurence of the proposition p, p itself is deleted from

the stack *Sub*; each time a satsfied clause ceases to be so, we decrement *open*[*s*], the total number of open clauses in the current state (lines 12-13)

To obtain DB-RETRACT-PROP-FALSE it is sufficient to swap lines 5-6 with lines 9-10, the remainder of the code being exactly the same.

In the remainder of this chapter and in the following, we will always deal with stack-wise operation of the data structure. The look-ahead primitives presented in section 4.2 can be easily extended to deal with non stack-wise operation of the data structure, while the look-back ones (4.3) are inerently stack-wise: this is the place where we should intervene to create a non-stackwise version of the search algorithms. Once this is done, the DLL-core (described in 4.5) will do with minor modifications.

4.2 Implementing Look-ahead

In this section we present the two commonest look-ahead primitives: boolean constraint propagation (BCP, a.k.a. unit clause propagation) and monotone literal fixing (MLF, a.k.a. pure literal rule). In particular, BCP is so fundamental that we included a mechanism for constant time detection of unit clauses in our primitives (see previous section). As reported for example in [Fre95], BCP is by far the most useful look-ahead technique among those seen in the literature. Our presentation of the look-ahead primitive BCP parallels the one of [Fre95]. MLF did not receive as much attention as BCP, probably because its lack of effectivness on some kind of formulas (see again [Fre95]). As a result, many state-of-the-art solvers, like RelSAT [BS97] and GRASP [SS96] do not even implement MLF, and most solvers that do, like SATO [Zha97] and SATZ [LA97], rely on the heuristic to detect pure literals and to propagate them. Here we suggest an intermediate approach: we rely on a dedicated primitive MLF for propagation and on the heuristics for detection.

4.2.1 Boolean constraint propagation (BCP)

In figure 4.7 we present the basic version of the primitive BCP. As the name suggests, the primitive performs the propagation of boolean contstraints (unit clauses) in state *s*. The primitive assigns a truth value to the single open proposition of each open unary clause that occurs in the *Unit* stack. The value assigned depends on the sign of the literal corresponding to the open proposition in unary clauses. BCP relies on EXTEND-PROP to extend each single valuation to the formula, and EXTEND-PROP pushes new unit clauses on the stack, so the net effect for BCP is to loop until:

- the formula does not contain unary open clauses any more, or
- an empty clause was found

Incidentally, we notice that BCP alone is able to decide the subclass of CNF formulas consisting of Horn clauses only.

In figure 4.7 we describe the procedure BCP.

- In line 1, the result r is initialized to TRUE .

```
BCP(s)
 1 r ← TRUE
 2 while length[Unit[s]] > 0 and r = TRUE do
 3    cl = POP(Unit[s])
 4    if sub[cl] = NIL then
 5       ▷ Locating the single open literal in cl (if any)
 6       i ← 1
 7       repeat
 8          l = Lits[cl][i]
 9          i ← i + 1
10       until value[prop[l]] = UNDEF
11       ▷ Propagating the valuation of p
12       r ← EXTEND-PROP(s, prop[l], sign[l], UN)
13 return r
```

Figure 4.7: Boolean constraint (unit clause) propagation.

- In lines 3-12 sits the body of the main loop that spins (line 2) while there are still unit clauses to propagate and the result is TRUE ; inside the loop, clauses popped from the *Unit* stack (line 3) that did not loose their eligibility for BCP (line 4) are searched for their unique open literal (lines 6-10); whenever an open literal bl is found the corresponding proposition $prop[l]$ is propagated with value $sign[l]$ (line 12)
- In line 13 the result r is returned: r will be FALSE if a call to EXTEND-PROP returns FALSE and TRUE otherwise

Notice that the correctness of BCP relies on EXTEND-PROP not to push any *false units*, i.e., clauses cl where $open[cl] = 1$ and $sub[cl] =$ NIL , but $value[prop[l]] \neq$ UNDEF for each l in $Lits[cl]$. Such a thing may indeed happen if EXTEND-PROP performs unit resolutions only, but this is not the case of the primitives presented in section 4.1.

4.2.2 Monotone literal fixing (MLF)

In order to accomodate for monotone literal fixing to be performed independently from the detection we need to enhance the definition of state that we introduced in section 4.1 Precisely, we add a field *Pure* to the object representing the state, its runtime condition being for any state s:

$$\begin{aligned} &\forall i(p = Pure[i] \wedge \\ &\quad (\forall j\, sub[Pos[p][j]] \neq \text{NIL}\ ) \vee (\forall j\, sub[Neg[p][j]] \neq \text{NIL}\ \vee)) \end{aligned} \tag{4.24}$$

where $1 < i < length[Pure[si]]$, and k and j have the obvious bounds for each proposition p. The stack *Pure* can thus be filled in some place, e.g., the heuristic, and then discharged using the primitive that we now define.

In figure 4.8 we present MLF. The primitive performs the fixing of monotone literals (pure literals) in state s. The primitive assigns a truth value to each proposition that occurs in the *Pure* stack, unless it was assigned by means

```
MLF(s)
 1 r ← TRUE
 2 while length[Pure[s]] > 0 do
 3    p = POP(Pure[s])
 4    if mode[p] = PP or mode[p] = PN then
 5       ▷ Propagating the valuation of p
 6       if mode[p] = PP then
 7          r ← EXTEND-PROP-TRUE(s, p, PP)
 8       else
 9          r ← EXTEND-PROP-FALSE(s, p, PN)
10 return r
```

Figure 4.8: Monotone literal fixing (pure literal).

of something else, namely, unit propagation, split or failed literal. The value assigned depends on whether the proposition occurs only positively or negatively: it is TRUE in the former case, FALSE in the latter. Unless EXTEND-PROP itself performs pure literal detection, MLF does not "propagate" pure literals in the same sense that BCP propagates unit clauses: once the original *Pure* stack is exhausted MLF stops even if new pure literals were created in the meantime. Also notice that assigning a proposition that occurs only positively (negatively) to TRUE (FALSE), never causes an inconsistency to arise. Therefore, the result of MLF will be always TRUE .

In figure 4.7 we describe the procedure MLF.

- In line 1, the result r is initialized to TRUE .
- In lines 3-9 sits the body of the main loop that spins (line 2) while there are still pure literals to propagate; inside the loop, propositions popped from the *Pure* stack (line 3) that did not loose their eligibility for MLF (line 4) are assigned a value according to the sign of their occurrences (lines 6-9)
- In line 13 the result r is returned: as noticed before, r is bound to be TRUE .

Notice that the correctness of MLF relies on the function performing the detection not to push any valued (non open) proposition: should this happen calling EXTEND-PROP on a valued proposition would disrupt the counters and bring to an inconsistent state. Also, for any proposition p pushed in *Pure*, the value of $mode[p]$ must correctly reflect the situation of p, i.e., it has to be $mode[p] =$ PP for positive pure literals and $mode[p] =$ PN for negative pure literals.

4.3 Implementing Look-back

In this section we present two stack-wise look-back primitives: chronological backtracking, the classic and simple algorithm that resumes the search from the most recent open branch, and conflict-directed backjumping with learning enhancement, a recent and more appealing combination of techniques. The

```
BACKTRACK(s, var v, var m)
 1 FLUSH(Unit[s])
 2 ▷ Go back in the search stack retracting valuations
 3 repeat
 4    p = TOP(Stack[s])
 5    if mode[p] ≠ LS then
 6       RETRACTPROP(p)
 7 until length[Stack[s]] = 0 or mode[p] = LS
 8 if length[Stack[s]] > 0 then
 9    ▷ An open branch was located
10    if value[p] = TRUE then v ← FALSE else v ← TRUE
11    m = RS
12    level[s] = level[p]
13    RETRACTPROP(s, p)
14 else
15    p ← 0
16 return p
```

Figure 4.9: Chronological backtracking.

original part of our work here is not the development of the techniques, but is their application on a destructive extend/retract operation of the data structures which is aligned to state of the art solvers like, e.g., RelSAT [BS97], SATZ [LA97], SATO [Zha97] and GRASP [SS96]. Of the DLL-based solvers cited, all but GRASP, use some kind of conflict-directed technique in order to prune the search space and to avoid local and global minima. In particular RelSAT code offered us the best source of inspiration for the algorithms presented in the subsection 4.3.2.

4.3.1 Chronological backtracking

In figure 4.7 we present the primitive BACKTRACK for chronological backtracking. The primitive restores the state s to the situation it was before the most recent assignment of a proposition p such that $mode[p] = \text{LS}$: the function return such proposition and sets suitably the by-reference parameters v and m. BACKTRACK relies on RETRACT-PROP to retract each single valuation from the formula, so the net effect for BACKTRACK is to loop until:

- the field *Stack* is empty
- a proposition p assigned such that $mode[p] = \text{LS}$ is found

Incidentally, we notice that BACKTRACK allows us to concretely implement a model of explicit recursion for our SAT solver.

In figure 4.9 we describe the procedure BACKTRACK.

- In lines 1-2, the stacks *Unit* and *Pure* are flushed since their content will not be significant any more at the end of backtracking.
- In lines 3-8 sits the body of the main loop that spins (line 8) until either *Stack* is empty or a proposition p such that Vmode$p$ = NIL is found; inside

the loop, when the top proposition of the stack was not propagated with LS it is retracted (therefore also popped from *Stack*).

- In lines 9-16, when an open branch on p is located, the value $value[p]$ is reversed (line 11) to obtain the new branching value v; after the new branching mode m and the new current level has been set (lines 12-13), the proposition is finally retracted (line 14); on the other hand, if *Stack* became empty, then no more branching is possible and p is set to NIL .
- In line 17 the result p is returned: p will be NIL if there are no more open branches, or will be the proposition assigned in the most recent open branch.

Notice that the correctness of BACKTRACK relies on the search algorithms to correctly set the mode of each proposition. In particular, UN, RS and FL share the meaning of a *deduction*, while LS has the meaning of a *decision* (terminology of [SS96]): as their name suggests, deductions are not to be traded in, whereas we may upturn our decisions.

4.3.2 Conflict-directed backjumping and learning

The complexity of the look-back techniques described in this section goes far beyond that of chronological backtracking. For one thing, backjumping and learning call for an extension of the data structure and, consequently, a modification of the primitives EXTEND-PROP, RETRACT-PROP and BCP. Moreover the look-back algorithm is itself more complicated that chronological backtracking. In this subsection we will describe the necessary changes to the data structure first, then we will present the modified versions of the aforementioned primitives, and finally we describe the algorithm for enhanced look-back with conflict analysis and learning.

For each proposition (object) p we need to add the following fields:

reason a clause, the reason (if any) of the propagation

Learned-pos an array for positive occurrences in learned clauses

Learned-neg the same for negative occurrences

Initially, for each proposition p we have the conditions:

$$reason[p] = \text{NIL} \tag{4.25}$$
$$length[Learned\text{-}pos[cl]] = 0 \tag{4.26}$$
$$length[Learned\text{-}neg[cl]] = 0 \tag{4.27}$$
$$\tag{4.28}$$

In words, for each proposition the reason is initially empty (**??**) and there are neither positive (

For each clause (object) cl we need to add the following fields:

l-tag the learned tag

ul-tag the learned unary clauses tag

Initially, for each clause cl we have the conditions:

$$l\text{-}tag[p] = 0 \tag{4.29}$$
$$ul\text{-}tag[p] = 0 \tag{4.30}$$
$$\tag{4.31}$$

In words, at the beginning each clause is not learned (**??**) and it is not a learned unary clause (**??**).

Let RELEVANCE and SIZE be two new constants. Also the state needs to be enhanced as follows:

type is the learning strategy, either RELEVANCE or SIZE

order is the learning order

Wr-lits an array of literals, the working reason

Wr-is-in membership flags for the working reason

Initially, for each proposition s we have the conditions:

$$order[s] \geq 1 \tag{4.32}$$
$$length[Wr\text{-}lits[s]] = 0 \tag{4.33}$$
$$length[Wr\text{-}is\text{-}in[s]] = length[Props[s]] \tag{4.34}$$
$$\tag{4.35}$$

In words, the learning order is at least one (**??**) for each proposition the reason is initially empty (**??**) and there are neither positive (

4.4 Implementing relaxations

4.4.1 Horn relaxation

4.4.2 Krom relaxation

4.5 The DLL algorithm revisited

```
EXTEND-PROP-TRUE(s, p, m)
 1 for i ← 1 to length[Neg[p]] do
 2     cl ← Neg[p][i]
 3     if sub[cl] = NIL then
 4         open[cl] ← open[cl] - 1
 5         if open[cl] = 1 then
 6             PUSH(Unit[s], cl)
 7         else if open[cl] = 0 then
 8             cl ← CHOOSE-EMPTY-CL(p, i, TRUE)
 9             INIT-WR(s, cl, p)
10             if LENGTH-WR(s) ≤ order[s] or learn[s] = RELEVANCE then
11                 LEARN-CLAUSE(s, MAKE-CLAUSE-FROM-WR(s))
12             for j ← i downto 1 do
13             cl ← Neg[p][i]
14             if sub[cl] = NIL then
15                 open[cl] ← open[cl] + 1
16             return FALSE
17 for i ← 1 to length[Learned-neg[p]] do
18     cl ← Learned-neg[p][i]
19     open[cl] ← open[cl] - 1
20     if open[cl] = 1 then
21         PUSH(Unit[s], cl)
22     else if open[cl] = 0 then
23         cl ← CHOOSE-EMPTY-CL(p, i, TRUE)
24         INIT-WR(s, cl, p)
25         if LENGTH-WR(s) ≤ order[s] or learn[s] = RELEVANCE then
26             LEARN-CLAUSE(s, MAKE-CLAUSE-FROM-WR(s))
27             for j ← i downto 1 do
28         for j ← i downto 1 do
29             cl ← Learned-neg[p][i]
30             open[cl] ← open[cl] + 1
31         for j ← length[Neg[p]] downto 1 do
32             cl ← Neg[p][i]
33             if sub[cl] = NIL then
34                 open[cl] ← open[cl] + 1
35             return FALSE
36 value[p] ← TRUE
37 mode[p] ← m
38 level[p] ← level[s]
39 PUSH(Stack[s], p)
40 for i ← 1 to length[Pos[p]] do
41     cl ← Pos[p][i]
42     if sub[cl] = NIL then
43         sub[cl] ← p
44         open[s] ← open[s] - 1
45 return TRUE
```

Figure 4.10: Redefining EXTEND-PROP-TRUE for conflict-directed backjumping and learning.

```
RETRACT-PROP-TRUE(s, p)
 1 if mode[p] = RS then
 2    DELETE-CL(reason[p])
 3    reason[p] = NIL
 4 value[p] ← UNDEF
 5 POP(Stack[s], p)
 6 for i ← 1 to length[Neg[p]] do
 7    cl ← Neg[p][i]
 8    if sub[cl] = NIL then
 9       open[cl] ← open[cl] + 1
10 for i ← 1 to length[Learned-neg[p]] do
11    cl ← Learned-neg[p][i]
12    open[cl] ← open[cl] + 1
13    if open[cl] > order[s] then
14       UNLEARN-CLAUSE(s, cl)
15    else if open[cl] = 1 then
16       PUSH(Learned-unit[s], cl)
17       lu-tag[cl] = length[Learned-unit[s]]
18    else if open[cl] = 2 and lu-tag[cl] ≠ 0 then
19       DELETE(Learned-unit[s], cl)
20       lu-tag[cl] = 0
21 for i ← 1 to length[Pos[p]] do
22    cl ← Pos[p][i]
23    if sub[cl] = p then
24       sub[cl] ← NIL
25       open[s] ← open[s] + 1
26 return
```

Figure 4.11: Redefining RETRACT-PROP-TRUE for conflict-directed backjumping and learning.

```
BCP(s)
 1 r ← TRUE
 2 while length[Unit[s]] > 0 and r = TRUE do
 3    cl = POP(Unit[s])
 4    if sub[cl] = NIL then
 5       i ← 1
 6       repeat
 7          l = Lits[cl][i]
 8          i ← i + 1
 9       until i > length[Lits[cl]] or value[prop[l]] = UNDEF
10       if i ≤ length[Lits[cl]] then
11          reason[prop[l]] ← cl
12          r ← EXTEND-PROP(s, prop[l], sign[l], UN)
13 return r
```

Figure 4.12: Redefining BCP for conflict-directed backjumping and learning.

```
BACKTRACK(s, var v, var m)
 1 FLUSH(Unit[s])
 2 FLUSH(Pure[s])
 3 repeat
 4    p = TOP(Stack[s])
 5    if IS-MEMBER-WR(s, p) then
 6       if mode[p] ∈ {UN, FL, RS} then
 7          UPDATE-WR(s, p)
 8          ▷ Learn the clause because of size or relevance
 9          if LENGTH-WR(s) ≤ order[s] or learn[s] = RELEVANCE then
10             LEARN-CLAUSE(s, MAKE-CLAUSE-FROM-WR(s))
11          RETRACTPROP(s, p)
12       else if mode[p] = LS
13          if value[p] = TRUE then v ← FALSE else v ← TRUE
14          m = RS
15          level[s] = level[p] - 1
16          RETRACTPROP(s, p)
17          ▷ Propagate the learned unit clauses
18          if LEARNED-BCP(s) = TRUE then
19             if mode[p] = UNDEF then
20                level[s] = level[s] + 1
21                reason[p] = MAKE-CLAUSE-FROM-WR(s)
22                return p
23             else
24                return CHOOSE-LITERAL(s, v, m)
25          else
26             FLUSH(Unit[s])
27    else
28       ▷ The proposition was irrelevant to the conflict
29       RETRACTPROP(s, p)
30 until length[Stack[s]] = 0
31 return NIL
```

Figure 4.13: Conflict directed backjumping with learning.

```
EXTEND-PROP-FALSE(s, p, m)
 1 for i ← 1 to length[Pos[p]] do
 2     cl ← Pos[p][i]
 3     if sub[cl] = NIL then
 4         if pos[cl] = 2 then
 5         ▷ The clause is becoming Horn
 6             DELETE(NhClauses[s], cl)
 7             open[s] ← open[s] - 1
 8         pos[cl] ← pos[cl] - 1
 9         open[cl] ← open[cl] - 1
10         if open[cl] = 1 then
11             PUSH(Unit[s], cl)
12         else if open[cl] = 0 then
13             for j ← i downto 1 do
14                 cl ← Pos[p][i]
15                 if sub[cl] = NIL then
16                     if pos[cl] = 1 then
17                         PUSH(NhClauses[s], cl)
18                         open[s] ← open[s] + 1
19                     pos[cl] ← pos[cl] + 1
20                     open[cl] ← open[cl] + 1
21             return FALSE
22 value[p] ← TRUE
23 mode[p] ← m
24 level[p] ← level[s]
25 PUSH(Stack[s], p)
26 for i ← 1 to length[Neg[p]] do
27     cl ← Neg[p][i]
28     if sub[cl] = NIL then
29         sub[cl] ← p
30         ▷ Check out non-Horn clauses only
31         if pos[cl] > 1 then
32             open[s] ← open[s] - 1
33 return TRUE
```

Figure 4.14: Redefining EXTEND-PROP-FALSE to handle Horn relaxation.

```
EXTEND-PROP-TRUE(s, p, m)
 1 for i ← 1 to length[Neg[p]] do
 2     cl ← Neg[p][i]
 3     if sub[cl] = NIL then
 4         if open[cl] = 3 then
 5         ▷ The clause is becoming Krom (binary)
 6             DELETE(NkClauses[s], cl)
 7             open[s] ← open[s] - 1
 8         open[cl] ← open[cl] - 1
 9         if open[cl] = 1 then
10             PUSH(Unit[s], cl)
11         else if open[cl] = 0 then
12             for j ← i downto 1 do
13                 cl ← Neg[p][i]
14                 if sub[cl] = NIL then
15                     if open[cl] = 2 then
16                         PUSH(NkClauses[s], cl)
17                         open[s] ← open[s] + 1
18                     open[cl] ← open[cl] + 1
19             return FALSE
20 value[p] ← TRUE
21 mode[p] ← m
22 level[p] ← level[s]
23 PUSH(Stack[s], p)
24 for i ← 1 to length[Pos[p]] do
25     cl ← Pos[p][i]
26     if sub[cl] = NIL then
27         sub[cl] ← p
28         ▷ Check out non-Krom clauses only
29         if open[cl] > 2 then
30             open[s] ← open[s] - 1
31 return TRUE
```

Figure 4.15: Redefining EXTEND-PROP-TRUE to handle Krom (binary clauses) relaxation.

```
DLL-SOLVE(s)
 1 repeat
 2    if BCP(s) = TRUE and MLF(s) = TRUE then
 3       if open[s] = 0 then return TRUE
 4       p ← CHOOSE-LITERAL(s, v, m)
 5    else
 6       p ← BACKTRACK(s, v, m)
 7    if p ≠ NIL then
 8       EXTEND-PROP(s, p, v, m)
 9 until p = NIL
10 return (open[s] = 0)
```

Figure 4.16: Iterative version of the DLL algorithm

Chapter 5

Heuristics and optimizations

5.1 Search heuristics

5.1.1 General framework for greedy heuristics

5.1.2 Jeroslow Wang (JW) and 2-sided Jeroslow Wang (2JW)

5.1.3 Max occurrences in clauses of min size (MOMS)

5.1.4 Shortest non-horn first (SATO)

5.1.5 Lexycographic (Boehm)

5.1.6 General framework for BCP-based heuristics

5.1.7 Unit, Unirel and Unirel2 heuristics

5.1.8 Unit with tie breaking (Unitie)

5.1.9 Combining BCP-based greedy methods (Unimo)

```
CHOOSE-LITERAL-UNIMO(s, var v, var m)
 1 if level[s] < SWITCH then
 2    if branch[s] = MODEL then
 3       return BCP-HEUR(s, Model-props[s], var v, var m)
 4    else
 5       return BCP-HEUR(s, Props[s], var v, var m)
 6 else
 7    if branch[s] = MODEL then
 8       return GREEDY-HEUR(s, Model-props[s], var v, var m)
 9    else
10       return GREEDY-HEUR(s, Props[s], var v, var m)
```

```
GREEDY-HEUR(s, P, var v, var m)
 1 best-w ← 0
 2 best-p ← NIL
 3 for each p ∈ P do
 4    if value[p] = UNDEF then
 5       pos-w ← neg-w ← 0
 6       for each cl ∈ Pos[p] do
 7          if sub[cl] = NIL then
 8             len ← CLAUSE-LENGTH(cl)
 9             pos-w ← SCORE(pos-w, len)
10       for each cl ∈ Neg[p] do
11          if sub[cl] = NIL then
12             len ← CLAUSE-LENGTH-JW(cl)
13             neg-w ← SCORE(neg-w, len)
14       w ← COMBINE(pos-w, neg-w)
15       if w > best-w then
16          best-w ← w
17          best-p ← p
18          if neg-w > pos-w then v ← FALSE else v ← TRUE
19 m ← LS
20 level[s] ← level[s] + 1
21 return best-p
```

Figure 5.1: General template for greedy heuristics

5.1.10 SATZ heuristic

5.1.11 RelSAT heuristic

5.2 Randomization

5.2.1 Noisy heuristics

5.2.2 Serch cut off and restart

5.3 Preprocessing

5.3.1 Failed literals propagation

5.3.2 k-literals simplification

5.3.3 Tail resolution

5.3.4 Clause subsumption

5.3.5 Binary clauses propagation

5.3.6 Adding short resolvents

```
CHOOSE-LITERAL-JW(s, var v, var m)
 1 if branch[s] = MODEL then
 2    return GREEDY-HEUR(s, Model-props[s], var v, var m)
 3 else
 4    return GREEDY-HEUR(s, Props[s], var v, var m)

CLAUSE-LENGTH-JW(cl)
 1 if open[cl] > MAXLEN then
 2    return 0
 3 else
 4    return (MAXLEN - length[Lits[cl]])

SCORE-JW(w, len)
 1 w ← w + EXP2(len)
 2 return w

COMBINE-JW(pos-w, neg-w)
 1 if neg-w > pos-w then
 2    return ← neg-w
 3 else
 4    return pos-w

COMBINE-2JW(pos-w, neg-w)
 1 return pos-w + neg-w
```

Figure 5.2: Obtaining Jeroslow-Wang heuristics from GREEDY-HEUR.

```
CHOOSE-LITERAL-MOMS(s, var v, var m)
 1 if branch[s] = MODEL then
 2    return GREEDY-HEUR(s, Model-props[s], var v, var m)
 3 else
 4    return GREEDY-HEUR(s, Props[s], var v, var m)

CLAUSE-LENGTH-MOMS(cl)
 1 if open[cl] > 2 then
 2    return W3
 3 else
 4    return W2

SCORE-MOMS(w, len)
 1 w ← w + len
 2 return w

COMBINE-MOMS(pos-w, neg-w)
 1 return (pos-w + 1) · (neg-w + 1)
```

Figure 5.3: Obtaining MOMS heuristic from GREEDY-HEUR.

```
CHOOSE-LITERAL-SATO(s, var v, var m)
 1 min-len ← length[Props[s]]
 2 for each cl ∈ NhClauses[s] do
 3    if sub[cl] = NIL then
 4       if open[cl] < min-len then
 5          min-len ← open[cl]
 6          FLUSH(C)
 7          PUSH(C, cl)
 8       else if length[C] < MAGIC and open[cl] = min-len then
 9          PUSH(C, cl)
10 FLUSH(P)
11 while length[P] < MAGIC do
12    cl = POP(C)
13    for each l ∈ Lits[cl] do
14       p ← prop[l]
15       if value[p] = UNDEF then
16          value[p] ← TRUE
17          PUSH(P, p)
18 for each p ∈ P do
19    value[p] ← UNDEF
20 return GREEDY-HEUR(s, P, v, m)
```

Figure 5.4: Shortest non-Horn first (SATO) heuristic

```
CHOOSE-LITERAL-BOEHM(s, var v, var m)
 1 if branch[s] = MODEL then P = Model-props[s] else P = Props[s]
 2 best-w ← best-wall ← 0
 3 best-p ← NIL
 4 min-len ← length[Props[s]]
 5 for each p ∈ P do
 6    if value[p] = UNDEF then
 7       pos-w ← pos-wall ← 0
 8       for each cl ∈ Neg[p] do
 9          if sub[cl] = NIL then
10             pos-wall ← pos-wall + 1
11             len ← open[cl]
12             if len < min-len then
13                best-p ← p
14                min-len ← len
15                pos-w ← 1
16             else if len = min-len then
17                pos-w ← pos-w + 1
18       neg-w ← neg-wall ← 0
19       for each cl ∈ Neg[p] do
20          if sub[cl] = NIL then
21             neg-wall ← neg-wall + 1
22             len ← open[cl]
23             if len < min-len then
24                best-p ← p
25                min-len ← len
26                neg-w ← 1
27             else if len = min-len then
28                neg-w ← neg-w + 1
29       w ← A · MAX(pos-w, neg-w) + B · MIN(pos-w, neg-w)
30       wall ← A · MAX(pos-wall, neg-wall) + B · MIN(pos-wall, neg-wall)
31       if best-p = p then
32          if neg-w > pos-w then v ← FALSE else v ← TRUE
33          best-w ← w
34          best-wall ← wall
35       else if pos-w > 0 or neg-w > 0 then
36          if (w > best-w) or (w = best-w and wall > best-wall) then
37             best-p ← p
38             if neg-w > pos-w then v ← FALSE else v ← TRUE
39             best-w ← w
40             best-wall ← wall
41 m ← LS
42 level[s] ← level[s] + 1
43 return best-p
```

Figure 5.5: Lexycographic (Boehm's) heuristic

```
LEAN-EXTEND-PROP-TRUE(ls, p)
 1 PUSH(Changed[ls], p)
 2 value[p] ← TRUE
 3 for each cl ∈ Neg[p] do
 4    if sub[cl] = NIL then
 5       PUSH(Managed[ls], cl)
 6       len ← open[cl] ← open[cl] - 1
 7       EXT-SCORE(ls, p, cl)
 8       if len = 1 then
 9          PUSH(Unit[ls], cl)
10       else if len = 0 then
11          INIT-WR(s, cl, p)
12          return FALSE
13 for each cl ∈ Learned-Neg[p] do
14    if sub[cl] = NIL then
15       PUSH(Managed[ls], cl)
16       len ← open[cl] ← open[cl] - 1
17       EXT-SCORE(ls, p, cl)
18       if len = 1 then
19          PUSH(Unit[ls], cl)
20       else if len = 0 then
21          INIT-WR(s, cl, p)
22          return FALSE
23 return TRUE
```

Figure 5.6: Fast EXTEND-PROP-TRUE for BCP-based heuristics.

```
LEAN-BCP(ls)
 1 r ← TRUE
 2 while length[Unit[s]] > 0 and r = TRUE do
 3    cl = POP(Unit[ls])
 4    i ← 1
 5    repeat
 6       l = Lits[cl][i]
 7       i ← i + 1
 8    until i > length[Lits[cl]] or value[prop[l]] = UNDEF
 9    if i ≤ length[Lits[cl]] then
10       reason[prop[l]] ← cl
11       BCP-SCORE(ls, p)
12       r ← LEAN-EXTEND-PROP(ls, prop[l])
13 return r
```

Figure 5.7: Fast BCP for BCP-based heuristics.

```
LEAN-BACKTRACK-FALSE(ls)
 1 while length[Managed[ls]] > 0 do
 2     cl ← POP(Managed[ls])
 3     open[cl] ← open[cl] + 1
 4 while length[Changed[ls]] > 1 do
 5     p ← POP(Changed[ls])
 6     UPDATE-WR(ls, p)
 7     value[p] ← UNDEF
 8 p ← POP(Changed[ls])
 9 value[p] ← UNDEF
10 return MAKE-CLAUSE-FROM-WR(ls)
```

Figure 5.8: Fast BACKTRACK for BCP-based heuristics (failed literal).

```
BCP-HEUR(s, P, var v, var m)
 1 repeat
 2    w ← best-w ← 0
 3    best-p ← NIL
 4    FLUSH(Best)
 5    for each p ∈ P do
 6       if value[p] = UNDEF then
 7          INIT-SCORES(ls, TRUE )
 8          LEAN-EXTEND-PROP-TRUE(ls, p)
 9          if LEAN-BCP(ls) = FALSE then
10             reason[p] ← LEAN-BACTRACK-FALSE(ls)
11             EXTEND-PROP-FALSE(s, p, FL)
12             if BCP(s) = FALSE then
13                return BACKTRACK(s, v, m)
14          else
15             LEAN-BACTRACK-TRUE(ls)
16             INIT-SCORES(ls, FALSE )
17             LEAN-EXTEND-PROP-FALSE(ls, p)
18             if LEAN-BCP(ls) = FALSE then
19                reason[p] ← LEAN-BACTRACK-FALSE(ls)
20                EXTEND-PROP-TRUE(s, p, FL)
21                BCP(s)
22             else
23                LEAN-BACTRACK-TRUE(ls)
24                w ← COMBINE(ls)
25                if w > best-w then
26                   best-w ← w
27                   FLUSH(Best)
28                   PUSH(Best, p)
29    best-p ← CHOOSE-BEST(Best, v)
30 until best-p = NIL or value[best-p] = UNDEF
31 m ← LS
32 level[s] ← level[s] + 1
33 return best-p
```

Figure 5.9: General template for BCP-based heuristic.

```
CHOOSE-LITERAL-UNIT(s, var v, var m)
 1 return BCP-HEUR(s, Props[s], var v, var m)

INIT-SCORES-UNIT(ls, v)
 1 if v = TRUE then
 2     pos-w[ls] ← 1
 3     value[ls] ← TRUE
 4 else
 5     neg-w[ls] ← 1
 6     value[ls] ← FALSE


BCP-SCORE-UNIT(ls, p)
 1 if value[ls] = TRUE then
 2     pos-w[ls] ← pos-w[ls] + 1
 3 else
 4     neg-w[ls] ← neg-w[ls] + 1


COMBINE-UNIT(ls)
 1 w ← pos-w[ls] · neg-w[ls] · SPREAD + pos-w[ls] + neg-w[ls] + 1
 2 return w
```

Figure 5.10: Obtaining Unit heuristic from BCP-HEUR.

```
CHOOSE-LITERAL-UNIREL(s, var v, var m)
 1 return BCP-HEUR(s, Props[s], var v, var m)

INIT-SCORES-UNIREL(ls, v)
 1 if v = TRUE then
 2     pos-w[ls] ← 0
 3     value[ls] ← TRUE
 4 else
 5     neg-w[ls] ← 0
 6     value[ls] ← FALSE


BCP-SCORE-UNIREL(ls, p)
 1 if m-tag[p] > 0 then
 2     if value[ls] = TRUE then
 3         pos-w[ls] ← pos-w[ls] + 1
 4     else
 5         neg-w[ls] ← neg-w[ls] + 1


CHOOSE-LITERAL-UNIREL2(s, var v, var m)
 1 return BCP-HEUR(s, Model-props[s], var v, var m)
```

Figure 5.11: Obtaining Unirel and Unirel2 heuristics from BCP-HEUR.

```
CHOOSE-LITERAL-UNITIE(s, var v, var m)
 1 if branch[s] = MODEL then
 2    return BCP-HEUR(s, Model-props[s], var v, var m)
 3 else
 4    return BCP-HEUR(s, Props[s], var v, var m)

INIT-SCORES-UNITIE(ls, v)
 1 if v = TRUE then
 2    pos-w[ls] ← 1
 3    pos2-w[ls] ← 0
 4    value[ls] ← TRUE
 5 else
 6    neg-w[ls] ← 1
 7    neg2-w[ls] ← 0
 8    value[ls] ← FALSE

EXT-SCORE-UNITIE(ls, p, cl)
 1 if open[cl] = 2 then
 2 if value[ls] = TRUE then
 3    pos2-w[ls] ← pos2-w[ls] + 1
 4 else
 5    neg2-w[ls] ← neg2-w[ls] + 1

BCP-SCORE-UNITIE(ls, p)
 1 if value[ls] = TRUE then
 2    pos-w[ls] ← pos-w[ls] + 1
 3 else
 4    neg-w[ls] ← neg-w[ls] + 1

COMBINE-UNITIE(ls)
 1 w ← pos2-w[ls] · neg2-w[ls] · SPREAD + pos-w[ls] + neg-w[ls] + 1
 2 return w
```

Figure 5.12: Obtaining basic Unitie heuristic from BCP-HEUR.

```
CHOOSE-BEST-FIRST(Best, var v)
 1 if length[Best] = 0 then return NIL
 2 backup ← TOP(Best)
 3 v ← TRUE
 4 for each p ∈ Best do
 5    if value[p] = UNDEF then
 6       return p
 7 return backup

CHOOSE-BEST-RANDOM(Best, var v)
 1 if length[Best] = 0 then return NIL
 2 backup ← TOP(Best)
 3 v ← TRUE
 4 FLUSH(Best-open
 5 i ← RANDOM mod (length[Best])
 6 for each p ∈ Best do
 7    if value[p] = UNDEF then
 8       if i = 0 then
 9          return p
10       else
11          i ← i - 1
12          PUSH(Best-open, p)
13 if length[Best-open] = 0 then
14    return backup
15 else
16    return Best-open[i mod length[Best-open]]

CHOOSE-BEST-GREEDY(Best, var v)
 1 if length[Best] = 0 then return NIL
 2 best-w ← 0
 3 best-p ← TOP(Best)
 4 for each p ∈ Best do
 5    if value[p] = UNDEF then
 6       pos-w ← neg-w ← 0
 7       for each cl ∈ Pos[p] do
 8          if sub[cl] = NIL then pos-w ← pos-w + 1
 9       for each cl ∈ Neg[p] do
10          if sub[cl] = NIL then neg-w ← neg-w + 1
11       if pos-w > best-w or neg-w > best-w then
12          best-p ← p
13          if neg-w > pos-w then
14             best-w ← neg-w
15             v ← FALSE
16          else
17             best-w ← pos-w
18             v ← TRUE
19 return best-p
```

Figure 5.13: Strategies for breaking ties in CHOOSE-LITERAL-UNITIE.

```
CHOOSE-LITERAL-SATZ(s, var v, var m)
 1 repeat
 2    best-w ← 0
 3    best-p ← NIL
 4    for each p ∈ Props[s] do
 5       if value[p] = UNDEF then
 6          idp ← id[p]
 7          pw ← pwa ← Redp[ls][idp] ← 0
 8          nw ← nwa ← Redn[ls][idp] ← 0
 9          for each cl ∈ Pos[p] do
10             if sub[cl] = NIL then
11                if open[cl] = 2 then pw ← pw + 1 else pwa ← pwa + 1
12          Plen[ls][idp] ← pw
13          Plen-all[ls][idp] ← pwa
14          for each cl ∈ Neg[p] do
15             if sub[cl] = NIL then
16                if open[cl] = 2 then nw ← nw + 1 else nwa ← nwa + 1
17          Nlen[ls][idp] ← nw
18          Nlen-all[ls][idp] ← nwa
19          if PROP-4-1(pw, nw) then
20             r ← EXAMINE(ls, p)
21             if r = FALSE then return BACKTRACK(s, var v, var m)
22             else if r ≠ TRUE then PUSH(Chosen[ls], p)
23    if length[Chosen[ls]] < T then
24    for each p ∈ Props[s] do
25       if value[p] = UNDEF then
26          idp ← id[p]
27          if PROP-4-1(Plen[ls][idp], Plen[ls][idp]) = FALSE then
28             if PROP-3-1(Plen[ls][idp], Plen[ls][idp]) = TRUE then
29                r ← EXAMINE(ls, p)
30                if r = FALSE then return BACKTRACK(s, var v, var m)
31                else if r ≠ TRUE then PUSH(Chosen[ls], p)
32    if length[Chosen[ls]] < T then
33       FLUSH(Chosen[ls])
34       if branch[s] = MODEL then P = Model-props[s] else P = Props[s]
35       for each p ∈ P do
36          if value[p] = UNDEF then
37                r ← EXAMINE-0(ls, p)
38                if r = FALSE then return BACKTRACK(s, var v, var m)
39                else if r ≠ TRUE then PUSH(Chosen[ls], p)
40    for each p ∈ Chosen[ls] do
41       w ← pos2-w[ls] · neg2-w[ls] · SPREAD + pos-w[ls] + neg-w[ls] + 1
42       if w > best-w then
43          best-w ← w
44          best-p ← p
45 until best-p = NIL or value[best-p] = UNDEF
46 v ← TRUE
47 m ← LS
48 level[s] ← level[s] + 1
49 return best-p
```

Figure 5.14: SATZ heuristic.

Chapter 6

SIM: a DLL-based library of SAT solvers

6.1 Aims and design

6.2 Features

6.3 Putting SIM to the test

Part II

Modal logics

Chapter 7

Dealing with knowledge

7.1 Propositional modal logics

7.1.1 Classical modal logics

7.1.2 Standard modal logics

7.2 Algorithms for modal logics

7.2.1 Tableaux-based

7.2.2 Translation-based

7.2.3 SAT-based

Chapter 8

Contributions

8.1 Modal logic reasoners

8.1.1 DLP

8.1.2 TA

8.2 Experimental analysis

8.2.1 Open problems

8.2.2 Benchmarks in modal logics

Chapter 9

DLL-based decision procedures

Chapter 10

Implementing modal decision procedures

Chapter 11

*SAT: modal decision procedures on top of SAT-solvers

Part III

Quantified propositional logic (QSAT)

Chapter 12

Higher order satisfiability

Chapter 13

State of the art

13.1 Enhancing the DLL-based algorithm

13.1.1 Heuristics

13.1.2 Advanced look-ahead techniques

13.1.3 Preprocessing

13.2 QSAT solvers

13.2.1 Evaluate

13.2.2 Decide

13.2.3 QSolve

13.2.4 QKN

13.3 Experimental analysis

13.3.1 Available benchmarks

13.3.2 Designing a test set

13.3.3 A snapshot of DLL-based QSAT solvers

Chapter 14

From SAT to QSAT

14.1 Data structure and primitives

14.1.1 Formula and search state

14.1.2 Assigning and retracting truth values

14.2 Implementing Look-ahead

14.2.1 Extended BCP

14.2.2 Adapting MLF

14.2.3 Trivial truth

14.3 Implementing Look-back

14.3.1 Chronological backtracking

14.3.2 Conflict directed backtracking

14.3.3 A glimpse of learning

14.4 Search heuristics

14.4.1 Designing an heuristic for QSAT

14.4.2 Jeroslow Wang (JW) and 2-sided Jeroslow Wang (2JW)

14.4.3 Lexycographic

14.4.4 BCP-based

Chapter 15

QuBE: DLL-based procedure(s) for QSAT

Part IV

Bounded model checking (BMC)

Chapter 16

Model checking

Chapter 17

From SAT to BMC

Chapter 18

Applications

Chapter 19

Conclusions and future work

19.1 Wrapping up

19.2 Future development(s)

Appendix A

SIM system description

Appendix B

*SAT system description

Appendix C

QuBE system description

Bibliography

[BS97] Roberto J. Bayardo, Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)*, pages 203–208, Menlo Park, July 27–31 1997. AAAI Press.

[CLR98] Thomas H. Cormen, Charles E. Leiserson, and Ronald R. Rivest. *Introduction to Algorithms.* MIT Press, 1998.

[Fre95] Jon W. Freeman. *Improvements to propositional satisfiability search algorithms.* PhD thesis, University of Pennsylvania, 1995.

[LA97] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366–371, San Francisco, August 23–29 1997. Morgan Kaufmann Publishers.

[SS96] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. Technical report, University of Michigan, April 1996.

[Str97] Bjarne Stroustrup. *The C++ Programming Language - 3rd edition.* Addison-Wesley, 1997.

[Zha97] H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 272–275, Berlin, July13–17 1997. Springer.