Chapter 3

Complete Algorithms

Adnan Darwiche and Knot Pipatsrisawat

3.1. Introduction

This chapter is concerned with sound and complete algorithms for testing satisfiability, i.e., algorithms that are guaranteed to terminate with a correct decision on the satisfiability/unsatisfiability of the given CNF. One can distinguish between a few approaches on which complete satisfiability algorithms have been based. The first approach is based on existential quantification, where one successively eliminates variables from the CNF without changing the status of its satisfiability. When all variables have been eliminated, the satisfiability test is then reduced into a simple test on a trivial CNF. The second approach appeals to sound and complete inference rules, applying them successively until either a contradiction is found (unsatisfiable CNF) or until the CNF is closed under these rules without finding a contradiction (satisfiable CNF). The third approach is based on systematic search in the space of truth assignments, and is marked by its modest space requirements. The last approach we will discuss is based on combining search and inference, leading to algorithms that currently underly most modern complete SAT solvers.

We start in the next section by establishing some technical preliminaries that will be used throughout the chapter. We will follow by a treatment of algorithms that are based on existential quantification in Section 3.3 and then algorithms based on inference rules in Section 3.4. Algorithms based on search are treated in Section 3.5, while those based on the combination of search and inference are treated in Section 3.6. Note that some of the algorithms presented here could fall into more than one class, depending on the viewpoint used. Hence, the classification presented in Sections 3.3-3.6 is only one of the many possibilities.

3.2. Technical Preliminaries

A *clause* is a disjunction of literals over distinct variables.[1] A propositional sentence is in *conjunctive normal form (CNF)* if it has the form $\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n$, where each α_i is a clause. For example, the sentence

---

[1] The insistence that all literals in a clause be over distinct variables is not standard.

$$(A \vee B \vee \neg C) \wedge (\neg A \vee D) \wedge (B \vee C \vee D)$$

is in conjunctive normal form and contains three clauses. Note that according to our definition, a clause cannot contain the literals P and $\neg P$ simultaneously. Hence, a clause can never be valid. Note also that a clause with no literals, the *empty clause*, is inconsistent. Furthermore, a CNF with no clauses is valid.

A convenient way to notate sentences in CNF is using sets. Specifically, a clause $l_1 \vee l_2 \vee \ldots \vee l_m$ is expressed as a set of literals $\{l_1, l_2, \ldots, l_m\}$. Moreover, a conjunctive normal form $\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n$ is expressed as a set of clauses $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. For example, the CNF given above would be expressed as:

$$\{\ \{A, B, \neg C\},\ \{\neg A, D\},\ \{B, C, D\}\ \}.$$

This set–based notation will prove very helpful when expressing algorithms that operate on CNFs.

Given our notational conventions, a CNF Δ is valid if Δ is the empty set: $\Delta = \emptyset$. Moreover, a CNF Δ is inconsistent if Δ contains the empty set: $\emptyset \in \Delta$. These two cases correspond to common boundary conditions that arise in recursive algorithms on CNFs.

In general, a formula Δ is said to *imply* another formula Γ, denoted $\Delta \models \Gamma$, iff every assignment that satisfies Δ also satisfies Γ. Note that if a clause C_i is a subset of another clause C_j, $C_i \subseteq C_j$, then $C_i \models C_j$. We say in this case that clause C_i *subsumes* clause C_j, as there is no need to have clause C_j in a CNF that also contains clause C_i. One important fact that we will base our future discussions on is that if a CNF formula implies the empty clause, the formula is essentially unsatisfiable.

3.2.1. Resolution

One of the simplest complete algorithms for testing satisfiability is based on the *resolution* inference rule [Rob65], which is defined as follows. Let P be a Boolean variable, and suppose that Δ is a CNF which contains clauses C_i and C_j, where $P \in C_i$ and $\neg P \in C_j$. The resolution inference rule allows us to derive the clause $(C_i - \{P\}) \cup (C_j - \{\neg P\})$, which is called a *resolvent* that is obtained by *resolving* C_i and C_j. The resolvent of a resolution is a clause implied by the resolved clauses. For example, we can resolve clause $\{A, B, \neg C\}$ with clause $\{\neg B, D\}$ to obtain the resolvent $\{A, \neg C, D\}$. We will say here that $\{A, \neg C, D\}$ is a *B–resolvent* as it results from resolving two clauses on the literals B and $\neg B$. Note that if we can use resolution to derive the empty clause from any formula, it means that the formula implies the empty clause, and, thus, is unsatisfiable.

Resolution is sound but is incomplete in the sense that it is not guaranteed to derive every clause that is implied by the given CNF. However, resolution is *refutation complete* on CNFs, i.e., it is guaranteed to derive the empty clause if the given CNF is unsatisfiable. This result is the basis for using resolution as a complete algorithm for testing satisfiability: we keep applying resolution until either the empty clause is derived (unsatisfiable CNF) or until no more applications of resolution are possible (satisfiable CNF).

Consider now the following resolution trace.

1. $\{\neg P, R\}$
2. $\{\neg Q, R\}$
3. $\{\neg R\}$
4. $\{P, Q\}$

---

5. $\{\neg P\}$ 1, 3
6. $\{\neg Q\}$ 2, 3
7. $\{Q\}$ 4, 5
8. $\{\}$ 6, 7

The clauses before the line represent initial clauses, while clauses below the line represent resolvents, together with the identifiers of clauses used to obtain them. The above resolution trace shows that we can derive the empty clause from the initial set of Clauses (1–4). Hence, the original clauses, together, are unsatisfiable.

An important special case of resolution is called unit resolution. *Unit resolution* is a resolution strategy which requires that at least one of the resolved clauses has only one literal. Such clause is call a *unit clause*. Unit resolution is not refutation complete, which means that it may not derive the empty clause from an unsatisfiable CNF formula. Yet one can apply all possible unit resolution steps in time linear in the size of given CNF. Because of its efficiency, unit resolution is a key technique employed by a number of algorithms that we shall discuss later.

3.2.2. Conditioning

A number of algorithms that we shall define on CNFs make use of the *conditioning* operation. The process of conditioning a CNF Δ on a literal L amounts to replacing every occurrence of literal L by the constant true, replacing $\neg L$ by the constant false, and simplifying accordingly. The result of conditioning Δ on L will be denoted by $\Delta|L$ and can be described succinctly as follows:

$$\Delta|L = \{\alpha - \{\neg L\} \mid \alpha \in \Delta, \ L \notin \alpha\}.$$

To further explain this statement, note that the clauses in Δ can be partitioned into three sets:

1. The set of clauses α containing the literal L. Since we are replacing L by true, any clause that mentions L becomes satisfied and can be removed from the formula. As a result, these clauses do not appear in $\Delta|L$.
2. The set of clauses α containing the literal $\neg L$. Since we are replacing $\neg L$ by false, the occurrences of $\neg L$ in these clauses no longer have any effect. Therefore, these clauses appear in $\Delta|L$ with the literal $\neg L$ removed.
3. The set of clauses α that contain neither L nor $\neg L$. These clauses appear in $\Delta|L$ without change.

For example, if

$$\Delta = \ \{\,\{A, B, \neg C\},\ \{\neg A, D\},\ \{B, C, D\}\,\},$$

then

$$\Delta | C = \ \{\,\{A, B\},\ \{\neg A, D\}\,\},$$

and

$$\Delta | \neg C = \ \{\,\{\neg A, D\},\ \{B, D\}\,\}.$$

The definition of conditioning can be extended to multiple literals in the obvious way. For example, $\Delta | C A \neg D = \{\emptyset\}$ (an inconsistent CNF). Moreover, $\Delta | \neg C D = \emptyset$ (a valid CNF).

3.3. Satisfiability by Existential Quantification

We will now discuss a class of complete algorithms which is based on the concept of *existential quantification*. The result of existentially quantifying variable P from a formula Δ is denoted by $\exists P\, \Delta$ and defined as follows:

$$\exists P\, \Delta \ \overset{def}{=}\ (\Delta | P) \vee (\Delta | \neg P).$$

Consider for example the CNF:

$$\Delta = \{\,\{\neg A, B\}, \{\neg B, C\}\,\},$$

which effectively says that A implies B and B implies C. We then have $\Delta | B = \{\{C\}\}$, $\Delta | \neg B = \{\{\neg A\}\}$ and, hence, $\exists B \Delta = \{\{\neg A, C\}\}$ (i.e., A implies C).

Existential quantification satisfies a number of properties, but the most relevant for our purposes here is this: Δ is satisfiable if and only if $\exists P \Delta$ is satisfiable. Hence, one can replace a satisfiability test on Δ by another satisfiability test on $\exists P \Delta$, which has one fewer variable than Δ (if Δ mentions P). One can therefore existentially quantify all variables in the CNF Δ, one at a time, until we are left with a trivial CNF that contains no variables. This trivial CNF must therefore be either $\{\emptyset\}$, which is unsatisfiable, or $\{\}$, which is valid and, hence, satisfiable.

The above approach for testing satisfiability can be implemented in different ways, depending on how existential quantification is implemented. We will next discuss two such implementations, the first one leading to what is known as the Davis-Putnam algorithm (DP), and the second one leading to what is known as symbolic SAT solving.

3.3.1. The DP Algorithm

The DP algorithm [DP60] for testing satisfiability is based on the following observation. Suppose that Δ is a CNF, and let Γ be another CNF which results from adding to Δ all P–resolvents, and then throwing out all clauses that mention P

(hence, Γ does not mention variable P). It follows in this case that Γ is equivalent to $\exists P\Delta$.[2] Consider the following CNF,

$$\Delta = \{\{\neg A, B\}, \{\neg B, C\}\}.$$

There is only one B–resolvent in this case: $\{\neg A, C\}$. Adding this resolvent to Δ, and throwing out those clauses that mention B gives:

$$\Gamma = \{\{\neg A, C\}\}.$$

This is equivalent to $\exists B\Delta$ which can be confirmed by computing $\Delta|B \vee \Delta|\neg B$.

The DP algorithm, also known as *directional resolution* [DR94], uses the above observation to existentially quantify all variables from a CNF, one at a time. One way to implement the DP algorithm is using a mechanism known as *bucket elimination* [Dec97], which proceeds in two stages: *constructing and filling* a set of buckets, and then *processing* them in some order. Specifically, given a variable ordering π, we construct and fill buckets as follows:

- A *bucket* is constructed for each variable P and is *labeled* with variable P.
- Buckets are sorted top to bottom by their labels according to order π.
- Each clause α in the CNF is added to the first Bucket P from the top, such that variable P appears in clause α.

Consider for example the CNF

$$\Delta = \{\{\neg A, B\}, \{\neg A, C\}, \{\neg B, D\}, \{\neg C, \neg D\}, \{A, \neg C, E\}\},$$

and the variable order C, B, A, D, E. Constructing and filling buckets leads to:[3]

$$
\begin{aligned}
&C : \{\neg A, C\}, \{\neg C, \neg D\}, \{A, \neg C, E\} \\
&B : \{\neg A, B\}, \{\neg B, D\} \\
&A : \\
&D : \\
&E :
\end{aligned}
$$

Buckets are then processed from top to bottom. To process Bucket P, we generate all P–resolvents using only clauses in Bucket P, and then add these resolvents to corresponding buckets below Bucket P. That is, each resolvent α is added to the first Bucket P' below Bucket P, such that variable P' appears in α. Processing Bucket P can then be viewed as a process of existentially quantifying variable P, where the result of such quantifying is now stored in the buckets below Bucket P.

Continuing with the above example, processing Bucket C adds one C–resolvent

---

[2]This follows from the fact that prime implicates of the formula can be obtained by closing the formula under resolution. After that, existentially quantifying a variable amounts dropping all clauses that mention the variable. Interested readers are referred to [LLM03, Mar00].

[3]It is not uncommon for buckets to be empty. It is also possible for all clauses to fall in the same bucket.

Algorithm 1 DP(CNF Δ, variable order π): returns UNSATISFIABLE or SATISFI-ABLE.

```
1: for each variable V of Δ do
2:     create empty bucket B_V
3: for each clause C of Δ do
4:     V = first variable of C according to order π
5:     B_V = B_V ∪ {C}
6: for each variable V of Δ in order π do
7:     if B_V is not empty then
8:         for each V-resolvent C of clauses in B_V do
9:             if C is the empty clause then
10:                 return  UNSATISFIABLE
11:             U = first variable of clause C according to order π
12:             B_U = B_U ∪ {C}
13: return  SATISFIABLE
```

to Bucket A:

$$C : \{\neg A, C\},\ \{\neg C, \neg D\},\ \{A, \neg C, E\}$$
$$B : \{\neg A, B\},\ \{\neg B, D\}$$
$$A : \qquad\qquad\qquad\qquad\qquad \{\neg A, \neg D\}$$
$$D :$$
$$E :$$

The buckets below Bucket C will now contain the result of existentially quantifying variable C. Processing Bucket B adds one B–resolvent to Bucket A:

$$C : \{\neg A, C\},\ \{\neg C, \neg D\},\ \{A, \neg C, E\}$$
$$B : \{\neg A, B\},\ \{\neg B, D\}$$
$$A : \qquad\qquad\qquad\qquad \{\neg A, \neg D\},\ \{\neg A, D\}$$
$$D :$$
$$E :$$

At this stage, the buckets below Bucket B contain the resulting of existentially quantifying both variables C and B. Processing Bucket A, leads to no new resolvents. We therefore have $\exists C, B, A \Delta = \{\}$ and the original CNF is consistent.

Algorithm 1 contains the pseudocode for directional resolution. Note that the amount of work performed by directional resolution is quite dependent on the chosen variable order. For example, considering the same CNF used above with the variable order E, A, B, C, D leads to the following buckets:

$$E : \{A, \neg C, E\}$$
$$A : \{\neg A, B\}\ \{\neg A, C\}$$
$$B : \{\neg B, D\}$$
$$C : \{\neg C, \neg D\}$$
$$D :$$

Processing the above buckets yields no resolvents in this case! This is another proof for the satisfiability of the given CNF, which was obtained by doing less work due to the chosen variable order.

In the case that the formula is satisfiable, we can extract a satisfying assignment from the trace of directional resolution in time that is linear to the size of
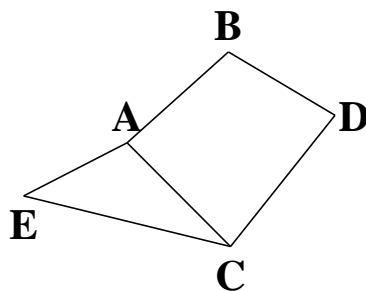
Figure 3.1. A connectivity graph for a CNF.

the formulas in all buckets [DR94]. To do so, we process the variables in the reverse order of the elimination (bottom bucket to top bucket). Let the elimination order be $\pi = V_1, V_2, ..., V_n$. For each variable V_i, if its bucket is empty, it can be assigned any value. Otherwise, we assign to V_i the value that, together with the values of $V_j, i < j < n$, satisfies all clauses in its bucket.

Using a suitable variable order, the time and space complexity of directional resolution can be shown to be $O(n \exp(w))$ [DR94], where n is the size of CNF Δ and w is the treewidth of its *connectivity graph:* an undirected graph G over the variables of Δ, where an edge exists between two variables in G iff these variables appear in the same clause in Δ. Figure 3.1 depicts the connectivity graph for CNF we considered earlier:

$$\Delta = \{\, \{\neg A, B\},\ \{\neg A, C\},\ \{\neg B, D\},\ \{\neg C, \neg D\},\ \{A, \neg C, E\} \,\}.$$

The complexity of this algorithms is discussed in more details in [DR94], which also shows that directional resolution becomes tractable on certain classes of CNF formula and that it can be used to answer some entailment queries on the formula.

3.3.2. Symbolic SAT Solving

The main problem observed with the DP algorithm is its space complexity, as it tends to generate too many clauses. A technique for dealing with this space complexity is to adopt a more compact representation of the resulting clauses. One such representation is known as *Binary Decision Diagrams (BDDs)* whose adoption leads to the class of *symbolic SAT algorithms* [CS00, AV01, MM02, PV04, FKS+04, JS04, HD04, SB06]. Many variations of this approach have been proposed over the years, but we discuss here only the basic underlying algorithm of symbolic SAT solving.

Figure 3.2 depicts an example BDD over three variables together with the models it encodes. Formally, a BDD is a rooted directed acyclic graph where there are at most two sinks, labeled with 0 and 1 respectively, and every other node is labeled with a Boolean variable and has exactly two children, distinguished as *low* and *high* [Bry86]. A BDD represents a propositional sentence whose models can be enumerated by taking all paths from the root to the 1-sink: taking the low
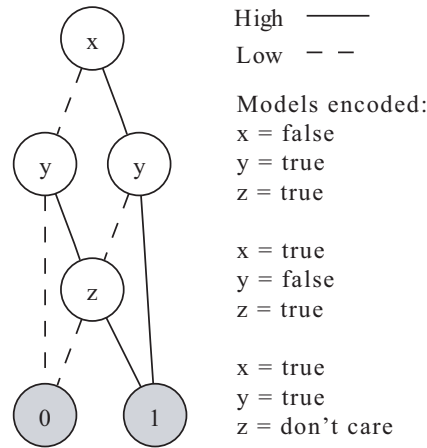
Figure 3.2. A BDD and the models it encodes.

(high) child of a node labeled with a variable X corresponds to assigning false
(true) to X; in case X does not appear on the path, it is marked as "don't care"
which means either value can be assigned.

In practice, BDDs are usually maintained so that variables appear in the same
order on any path from the root to a sink, leading to Ordered BDDs (OBDDs).
Two additional properties are often imposed: that there is no node whose two
children are identical, and that there are no isomorphic sub-graphs. Under these
conditions OBDDs are known to be a canonical form, meaning that there is a
unique OBDD for any propositional sentence under a given variable order [Bry86].
Moreover, any binary operation on two OBDDs with the same variable order can
be carried out using the *Apply* algorithm [Bry86], whose complexity is linear in
the product of the operand sizes. Hence, one can existentially quantify a variable
X from an OBDD by first conditioning it on X and then on $\neg X$, each of which
can be done in time linear in the OBDD size. One can then disjoin the resulting
OBDDs to obtain the result of existential quantification. The whole process can
therefore be implemented in time which is quadratic in the size of the OBDD.

To solve SAT, one may convert the CNF formula into the equivalent OBDD
and existentially quantify all variables from it. However, this naïve approach is
hardly practical as the size of the OBDD will be too large. Therefore, to make
this approach viable, symbolic SAT solving appeals to the following property of
existential quantification:

$$\exists X \Delta = (\exists X \Gamma_X) \wedge \Gamma,$$

where $\Delta = \Gamma_X \wedge \Gamma$ and Γ_X contains all clauses of CNF Δ that mention variable
X. Symbolic SAT solving starts by converting each clause C into a corresponding
OBDD, $OBBD(C)$, a process which can be done in time linear in the clause
size. To existentially quantify variable X, all OBDDs $\Gamma_1, \ldots, \Gamma_n$ that mention
variable X are conjoined together to produce one OBDD Γ_X, from which X is
existentially quantified. This technique is called *early quantification* [BCM$^+$90,

---

Algorithm 2 SYMBOLIC_SAT(CNF Δ, variable order π): returns UNSATISFIABLE or SATISFIABLE.

1: **for** each variable V of Δ **do**
2:     create empty bucket B_V
3: **for** each clause C of Δ **do**
4:     V = first variable of C according to order π
5:     $B_V = B_V \cup \{OBDD(C)\}$
6: **for** each variable V of Δ according to order π **do**
7:     **if** B_V is not empty **then**
8:         Γ_V = conjunction of all OBDDs in B_V
9:         **if** Γ_V = zero **then**
10:             **return** UNSATISFIABLE
11:         $\Gamma_V = \exists V \Gamma_V$
12:         U = first variable of Γ_V according to order π
13:         $B_U = B_U \cup \{\Gamma_V\}$
14: **return** SATISFIABLE

HKB96, TSL$^+$90]. The resulting OBDD $\exists X \Gamma_X$ will then replace the OBDDs $\Gamma_1, \ldots, \Gamma_n$, and the process is continued. If a zero-OBDD (one which is equivalent to false) is produced in the process, the algorithm terminates while declaring the original CNF unsatisfiable. However, if all variables are eliminated without producing a zero-OBDD, the original CNF is declared satisfiable.

The strategy for scheduling conjunctions and variable quantifications plays an important role in the performance of this algorithm. One goal is to try to minimize the size of intermediate OBDDs to reduce space requirement, which is often a major problem. Unfortunately, this problem is known to be NP-Hard [HKB96]. In [HD04], bucket elimination has been studied as a scheduling strategy. Algorithm 2 illustrates a symbolic SAT algorithm with bucket elimination. In [HD04], recursive decomposition [Dar01] and min-cut linear arrangement [AMS01] have been used as methods for ordering variables. Another approach for scheduling quantification is clustering [RAB$^+$95]. In [PV04], this approach was studied together with the maximum cardinality search (MCS) [TY84] as a variable ordering heuristic.

3.4. Satisfiability by Inference Rules

We have already discussed in Section 3.2 the use of the resolution inference rule in testing satisfiability. We will now discuss two additional approaches based on the use of inference rules.

3.4.1. Stålmarck's Algorithm

Stålmarck's algorithm was originally introduced in 1990 [SS90], and later patented in 1994 [Stå94], as an algorithm for checking tautology of arbitrary propositional formulas (not necessarily in CNF). To test the satisfiability of a CNF, we can equivalently check whether its negation is a tautology using Stålmarck's algorithm [SS90, Wid96, Har96]. Here, we will discuss Stålmarck's algorithm in its original form—as a tautology prover.

The algorithm starts with a preprocessing step in which it transforms implications into disjunctions, removes any double negations and simplifies the formula by applying rudimentary inference rules (such as removing true from conjunctions and false from disjunctions). If preprocessing is able to derive the truth value of the formula (as true or false), then the algorithm terminates. Otherwise, the algorithm continues on to the next step.

The preprocessed formula is converted into a conjunction of "triplets." Each triplet has the form $p \Leftrightarrow (q \otimes r)$, where $\otimes$ is either a conjunction or an equivalence and p must be a Boolean variable, while r and q are literals. During this process, new Boolean variables may need to be introduced to represent sub-formulas. For example, the formula $\neg((a \Leftrightarrow b \wedge c) \wedge (b \Leftrightarrow \neg c) \wedge a)$ may be transformed into

$$v_1 \Leftrightarrow (b \wedge c)$$
$$v_2 \Leftrightarrow (a \Leftrightarrow v_1)$$
$$v_3 \Leftrightarrow (b \Leftrightarrow \neg c)$$
$$v_4 \Leftrightarrow (v_3 \wedge a)$$
$$v_5 \Leftrightarrow (v_2 \wedge v_4)$$

Here, $\neg v_5$ is a literal whose truth value is equivalent to that of the original formula.[4] This conversion is meant to allow the algorithm to apply inference rules efficiently and to partially decompose the original formula into smaller sub-formulas that involve fewer variables. After this transformation, the algorithm assumes, for the sake of contradiction, that the whole formula has truth value false. For the remaining part, the algorithm tries to derive a contradiction, which would show that the original formula was actually a tautology.

In order to achieve this, the algorithm applies a set of inference rules called *simple rules* (or *propagation rule*) to the triplets to obtain more conclusions. Each conclusion either assigns a value to a variable or ties the value of 2 literals together. Some examples of simple rules are

$$p \Leftrightarrow (q \wedge r) \text{ :if } p = \neg q \text{ then } q = \text{true and } r = \text{false}$$
$$p \Leftrightarrow (q \wedge r) \text{ :if } p = \text{true then } q = \text{true and } r = \text{true}$$
$$p \Leftrightarrow (q \Leftrightarrow r) \text{ :if } p = q \text{ then } r = \text{true}$$
$$p \Leftrightarrow (q \Leftrightarrow r) \text{ :if } p = \text{true then } q = r.$$

The reader is referred to [Har96] for a complete set of simple rules used by the algorithm. The process of repeatedly and exhaustively applying simple rules to the formula is called *0–saturation*. After 0–saturation, if the truth value of the formula can be determined, then the algorithm terminates. Otherwise, the algorithm proceeds by applying the *dilemma rule*.

---

[4]This example is taken from [Har96].

The dilemma rule is a way of deducing more conclusions about a formula by making assumptions, which aid the application of simple rules. The rule operates as follows. Let Δ be the formula after 0–saturation. For each Boolean variable v in Δ, the algorithm 0–saturates $\Delta|v$ and $\Delta|\neg v$ to produce conclusions Γ_v and $\Gamma_{\neg v}$, respectively. Then, all conclusions that are common between Γ_v and $\Gamma_{\neg v}$ are added to Δ. In other words, the algorithm tries both values of the variable v and keeps all the conclusions derivable from both branches. These conclusions necessarily hold regardless of the value of v. The process is repeated for all variables until no new conclusions can be derived. The algorithm terminates as soon as contradicting conclusions are added to the knowledge base or the truth value of the formula can be determined from the conclusions. The process of applying the dilemma rule exhaustively on all variables is called 1–saturation. As one might expect, the algorithm can apply the dilemma rule with increasing depths of assignment. In particular, n–saturation involves case-splitting over all combinations of n variables simultaneously and gathering common conclusions. If we allow n to be large enough, the algorithm is guaranteed to either find a contradiction or reach a satisfying assignment. Because Stålmarck's algorithm n–saturates formulas starting from $n = 0$, it is oriented toward finding short proofs for the formula's satisfiability or unsatifiability.

3.4.2. HeerHugo

HeerHugo [GW00] is a satisfiability prover that was largely inspired by Stålmarck's algorithm. Although, strictly speaking, HeerHugo is not a tautology prover, many techniques used in this algorithm have lots of similarity to those of Stålmarck's algorithm.

According to this algorithm, the input formula is first converted into CNF with at most three literals per clause. Then, HeerHugo applies a set of basic inference rules, which are also called simple rules, to the resulting formula. Although these inference rules share the name with those rules in Stålmarck's algorithm, the two set of rules are rather different. Simple rules used in HeerHugo appear to be more powerful. They include unit resolution, subsumption, and a restricted form of resolution (see [GW00] for more details).

Resolution is selectively performed by HeerHugo, in order to existentially quantify some variables. This technique is similar to the one used in DP [DP60]. However, in HeerHugo, resolution is only carried out when it results in a smaller formula. Because of these rules, clauses are constantly added and removed from the knowledge base. Consequently, HeerHugo incorporates some subsumption and ad-hoc resolution rules for removing subsumed clauses and opportunistically generating stronger clauses.

During the process of applying simple rules, conclusions may be drawn from the formula. Like in Stålmarck's algorithm, a conclusion is either an assignment of a variable or an equivalence constraint between two literals. If a contradiction is found during this step, the formula is declared unsatisfiable. Otherwise, the algorithm continues with the *branch/merge* rule, which is essentially the same as the dilemma rule of Stålmarck's. The algorithm also terminates as soon as the truth value of the formula can be determined from the conclusions. The
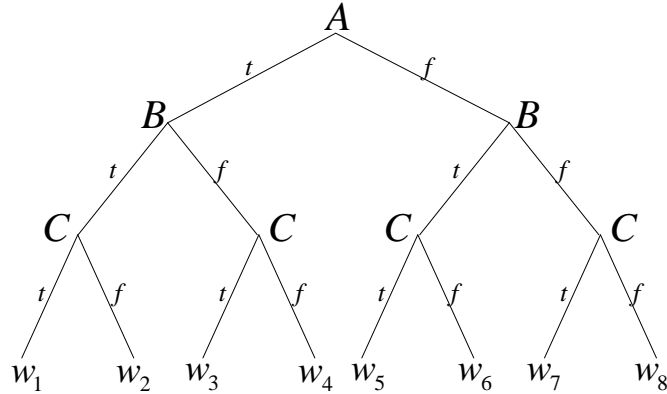
Figure 3.3. A search tree for enumerating all truth assignments over variables A, B and C.

completeness of HeerHugo is achieved by successively increasing the number of variables in the branch/merge rule, in a manner that is similar to applying n–saturation in Stålmarck's.

3.5. Satisfiability by Search: The DPLL Algorithm

We will discuss in this section the DPLL algorithm [DLL62], which is based on systematic search in the space of truth assignments. This algorithm is marked by its modest space requirements and was developed in response to the challenges posed by the space requirements of the DP algorithm.

Figure 3.3 depicts a search tree that enumerates the truth assignments over three variables. The first observation about this tree is that its leaves are in one–to–one correspondence with the truth assignments under consideration. Hence, testing satisfiability can be viewed as a process of searching for a leaf node that satisfies the given CNF. The second observation is that the tree has a finite depth of n, where n is the number of Boolean variables. Therefore, it is best to explore the tree using depth–first–search as given in Algorithm 3, which is called initially with depth $d = 0$. The algorithm makes use of the conditioning operator on CNFs, which leads to simplifying the CNF by either removing clauses or reducing their size.

Consider now the CNF:

$$\Delta = \ \{\,\{\neg A, B\}, \{\neg B, C\}\,\},$$

and the search node labeled with **F** in Figure 3.4. At this node, Algorithm 3 will condition Δ on literals $A, \neg B$, leading to:

$$\Delta | A, \neg B = \ \{\,\{\mathsf{false}, \mathsf{false}\}, \{\mathsf{true}, C\}\,\} = \ \{\{\}\}.$$

Algorithm 3 DPLL-(CNF Δ, depth d): returns a set of literals or UNSATISFIABLE. Variables are named $P_1, P_2, \ldots$.

---

if $\Delta = \{\}$ **then**
   return $\{\}$
else if $\{\} \in \Delta$ **then**
   return UNSATISFIABLE
else if $\mathbf{L} =$DPLL-$(\Delta|P_{d+1}, d+1) \neq$ UNSATISFIABLE **then**
   return $\mathbf{L} \cup \{P_{d+1}\}$
else if $\mathbf{L} =$DPLL-$(\Delta|\neg P_{d+1}, d+1) \neq$ UNSATISFIABLE **then**
   return $\mathbf{L} \cup \{\neg P_{d+1}\}$
else
   return UNSATISFIABLE
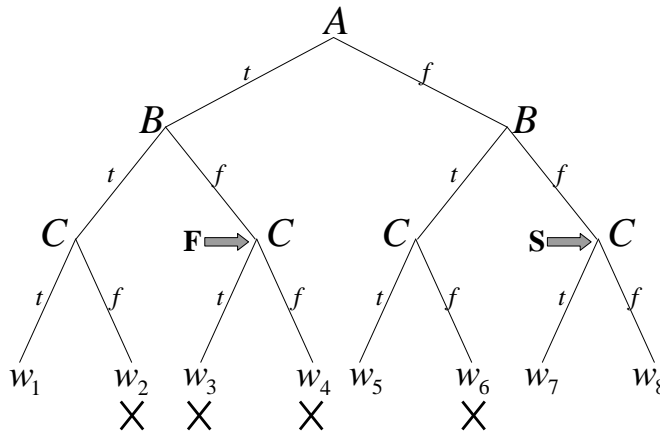
Figure 3.4. A search tree for that enumerates all truth assignments over variables A, B and C. Assignments marked by $\times$ are not models of the CNF $\{\,\{\neg A, B\}, \{\neg B, C\}\,\}$.

Hence, the algorithm will immediately conclude that neither truth assignment ω_3 nor ω_4 are models of Δ. The ability to detect contradictions at internal nodes in the search is quite significant as it allows one to dismiss all truth assignments that are characterized by that node, without having to visit each one of them explicitly. Algorithm 3 can also detect success at an internal node, which implies that all truth assignments characterized by that node are models of the CNF. Consider for example the internal node labelled with **S** in Figure 3.4. This node represents truth assignments ω_7 and ω_8, which are both models of Δ. This can be detected by conditioning Δ on literals $\neg A, \neg B$:

$$\Delta|\neg A, \neg B = \quad = \{\,\{\mathsf{true}, \mathsf{false}\}, \{\mathsf{true}, C\}\,\} = \quad \{\}.$$

Hence, all clauses are subsumed immediately after we set the values of A and B to false, and regardless of how we set the value of C. This allows us to conclude that both ω_7 and ω_8 are models of Δ, without having to inspect each of them
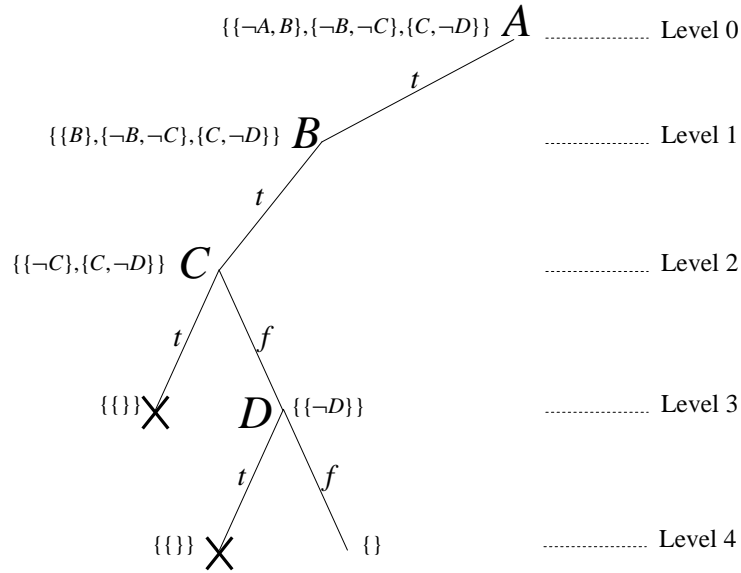
Figure 3.5. A termination tree, where each node is labelled by the corresponding CNF. The last node visited during the search is labelled with {}. The label × indicates the detection of a contradiction at the corresponding node.

individually. Given that Algorithm 3 may detect success before reaching a leaf node, it may return less than n literals, which would characterize a set of truth assignments in this case, all of which are guaranteed to be models of the input formula.

3.5.1. Termination Trees

One way to measure the amount of work done by Algorithm 3 is using the notion of a *termination tree:* the subset of search tree that is actually explored during search. Figure 3.5 depicts an example termination tree for the CNF:

$$\Delta = \{ \{\neg A, B\}, \{\neg B, \neg C\}, \{C, \neg D\} \}.$$

According to this figure, Δ is satisfiable and the truth assignment $A, B, \neg C, \neg D$ is a satisfying assignment. The figure also provides a trace of Algorithm DPLL- (Algorithm 3) as it shows the conditioned CNF at each node visited during the search process. The nodes in a termination tree belong to different levels as shown in Figure 3.5, with the root node being at Level 0. Termination trees are also useful in characterizing different satisfiability problems since the size and depth of tree appear to be good indicators of the problem character and difficulty.

3.5.2. Unit Resolution

Consider the termination tree of Figure 3.5, and corresponding CNF:

$$\Delta = \ \{ \{\neg A, B\},\ \{\neg B, \neg C\},\ \{C, \neg D\} \}.$$

Consider also the node at Level 1, which results from setting variable A to true, and its corresponding CNF:

$$\Delta | A = \ \{ \{B\},\ \{\neg B, \neg C\},\ \{C, \neg D\} \}.$$

This CNF is neither empty, nor contains the empty clause. Therefore, we cannot yet declare early success or early failure, which is why Algorithm 3 continues searching below Level 1 as shown in Figure 3.5. We will show now, however, that by employing unit resolution, we can indeed declare success and end the search at this node.

The *unit resolution technique* (or *unit propagation*) is quite simple: Before we perform the tests for success or failure on Lines 1 and 2 of Algorithm 3, we first close the CNF under unit resolution and collect all unit clauses in the CNF. We then assume that variables are set to satisfy these unit clauses. That is, if the unit clause $\{P\}$ appears in the CNF, we set P to true. And if the unit clause $\{\neg P\}$ appears in the CNF, we set P to false. We then simplify the CNF given these settings and check for either success (all clauses are subsumed) or failure (the empty clause is derived).

To incorporate unit resolution into our satisfiability algorithms, we will introduce a function UNIT-RESOLUTION, which applies to a CNF Δ and returns two results:

- **I**: a set of literals that were either present as unit clauses in Δ, or were derived from Δ by unit resolution.
- Γ: a new CNF which results from conditioning Δ on literals **I**.

For example, if the CNF

$$\Delta = \{ \{\neg A, \neg B\},\ \{B, C\},\ \{\neg C, D\},\ \{A\} \},$$

then $\mathbf{I} = \{A, \neg B, C, D\}$ and $\Gamma = \{\}$. Moreover, if

$$\Delta = \{ \{\neg A, \neg B\},\ \{B, C\},\ \{\neg C, D\},\ \{C\} \},$$

then $\mathbf{I} = \{C, D\}$ and $\Gamma = \{ \{\neg A, \neg B\} \}$. Unit resolution is a very important component of search-based SAT solving algorithms. Part 1, Chapter 4 discusses in details the modern implementation of unit resolution employed by many SAT solvers of this type.

Algorithm DPLL, for $\underline{D}$avis, $\underline{P}$utnam, $\underline{L}$ogemann and $\underline{L}$oveland, is a refinement on Algorithm 3 which incorporates unit resolution. Beyond applying unit resolution on Line 1, we have two additional changes to Algorithm 3. First of all, we no longer assume that variables are examined in the same order as we go down the search tree. Secondly, we no longer assume that a variable is set to true first, and then to false; see Line 7. The particular choice of a literal L on Line 7

Algorithm 4 DPLL(CNF Δ): returns a set of literals or UNSATISFIABLE.

```
1: (I, Γ) =UNIT-RESOLUTION(Δ)
2: if Γ = {} then
3:     return I
4: else if {} ∈ Γ then
5:     return UNSATISFIABLE
6: else
7:     choose a literal L in Γ
8:     if L =DPLL(Γ|L) ≠ UNSATISFIABLE then
9:         return L ∪ I ∪ {L}
10:    else if L =DPLL(Γ|¬L) ≠UNSATISFIABLE then
11:        return L ∪ I ∪ {¬L}
12:    else
13:        return UNSATISFIABLE
```

can have a dramatic impact on the running time of DPLL. Part 1, Chapter 7 is dedicated to heuristics for making this choice, which are known as *variable ordering* or *splitting* heuristics when choosing a variable, and *phase selection* heuristics when choosing a particular literal of that variable.

3.6. Satisfiability by Combining Search and Inference

We will now discuss a class of algorithms for satisfiability testing which form the basis of most modern complete SAT solvers. These algorithms are based on the DPLL algorithm, and have undergone many refinements over the last decade, making them the most efficient algorithms discussed thus far. Yet, these refinements have been significant enough to change the behavior of DPLL to the point where the new algorithms are best understood in terms of an interplay between search and inference. Early successful solvers employing this approach, such as GRASP [MSS99] and Rel_sat [BS97], gave rise to modern solvers namely Berk-Min [GN02], JeruSAT [DHN05a], MiniSAT [ES03], PicoSAT [Bie07], Rsat [PD07], Siege [Rya04], TiniSAT [Hua07a], and zChaff [MMZ+01]. We start this next section by discussing a main limitation of the DPLL algorithm, which serves as a key motivation for these refinements.

3.6.1. Chronological Backtracking

Algorithm DPLL is based on *chronological backtracking.* That is, if we try both values of a variable at level l, and each leads to a contradiction, we move to level $l - 1$, undoing all intermediate assignments in the process, and try another value at that level (if one remains to be tried). If there are no other values to try at level $l - 1$, we move to level $l - 2$, and so on. If we move all the way to Level 0, and each value there leads to a contradiction, we know that the CNF is inconsistent.

The process of moving from the current level to a lower level is known as *backtracking.* Moreover, if backtracking to level l is done only after having tried both values at level $l + 1$, then it is called *chronological backtracking,* which is the kind of backtracking used by DPLL. The problem with this type of backtracking is that it does not take into account the information of the contradiction that triggers the backtrack.
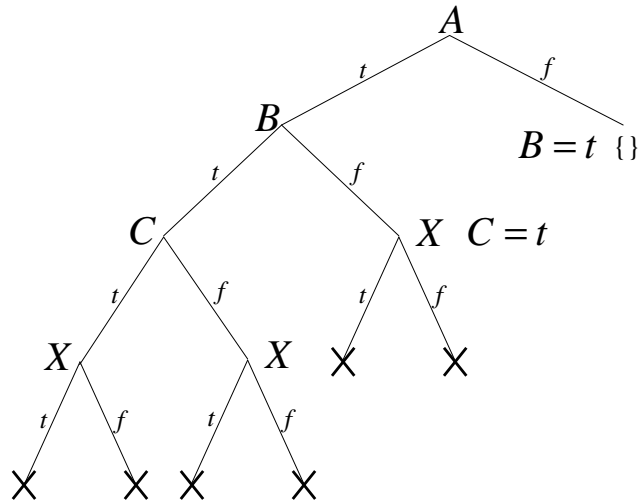
Figure 3.6. A termination tree. Assignments shown next to nodes are derived using unit resolution.

To consider a concrete example, let us look at how standard DPLL behaves on the following CNF, assuming a variable ordering of A, B, C, X, Y, Z:

$$\Delta = \begin{array}{l} 1.\ \{A, B\} \\ 2.\ \{B, C\} \\ 3.\ \{\neg A, \neg X, Y\} \\ 4.\ \{\neg A, X, Z\} \\ 5.\ \{\neg A, \neg Y, Z\} \\ 6.\ \{\neg A, X, \neg Z\} \\ 7.\ \{\neg A, \neg Y, \neg Z\} \end{array} \qquad (3.1)$$

Figure 3.6 depicts the termination tree for DPLL on the above CNF. We start first by setting A to true and explore the subtree below that node, only to find that all branches lead to contradictions. This basically means that $A = $ true is not a part of any solution. In other words, the formula $\Delta \wedge A$ is inconsistent. Note, however, that DPLL is unable to discover this fact in the knowledge until it has set variables B, C and X. Recall that DPLL uses unit resolution, which is not refutation complete. This explains why DPLL cannot detect the contradiction early on.

To provide more examples of the previous phenomena, note that DPLL is able to detect a contradiction in $\Delta|A, B, C, X$ and in $\Delta|A, B, C, \neg X$, which immediately implies a contradiction in $\Delta|A, B, C$. Yet DPLL is unable to detect a contradiction in $\Delta|A, B, C$.

When DPLL detects the contradiction in $\Delta|A, B, C, X$, it backtracks and tries the other value of X, leading also to a contradiction in $\Delta|A, B, C, \neg X$. DPLL then backtracks again and tries the second value of variable C. Again, no contradiction

is detected by DPLL in $\Delta|A, B, \neg C$, yet it later discovers that both $\Delta|A, B, \neg C, X$ and $\Delta|A, B, \neg C, \neg X$ are contradictory, leading it to backtrack again, and so on.

The key point here is that all these contradictions that are discovered deep in the search tree are actually caused by having set A to true at Level 0. That is, the settings of variables B and C at Levels 1 and 2 are irrelevant here. However, chronological backtracking is not aware of this fact. As a result, it tries different values of variables B, C, hoping to fix the contradiction. As shown in Figure 3.6, DPLL with chronological backtracking encounters 6 contradictions before realizing that $A =$ true was a wrong decision. In general, there can be many irrelevant variables between the level of conflict and the real cause of the conflict. Chronological backtracking may lead the solver to repeat the same conflict over and over again in different settings of irrelevant variables.

3.6.2. Non–Chronological Backtracking

Non–chronological backtracking addresses this problem by taking into account the set of variables that actually involve in the contradiction. Originally proposed as a technique for solving constraint satisfaction problems (CSPs), *non–chronological backtracking* is a method for the solver to quickly get out of the portion of the search space that contains no solution [SS77, Gas77, Pro93, BS97]. This process involves backtracking to a lower level l without necessarily trying every possibility between the current level and l.[5] Non–chronological backtracking is sometimes called for as it is not uncommon for the contradiction discovered at level l to be caused by variable settings that have been committed at levels much lower than l. In such a case, trying another value at level l, or at level $l - 1$, may be fruitless.

Non–chronological backtracking can be performed by first identifying every assignment that contributes to the derivation of the empty clause [BS97]. This set of assignments is referred to as the *conflict set* [Dec90]. Then, instead of backtracking to the last unflipped decision variable as in the case of chronological backtracking, non–chronological backtracking backtracks to the most recent decision variable that appears in the conflict set and tries its different value. Note that, during this process, all intermediate assignments (between the current level and the backtrack level) are erased.

In the above example, after the algorithm assigns $A =$ true, $B =$ true, $C =$ true, $X =$ true, unit resolution will derive $Y =$ true (Clause 3), $Z =$ true (Clause 5), and detect that Clause 7 becomes empty (all literals are already false). In this case, the conflict set is $\{A =$ true$, X =$ true$, Y =$ true$, Z =$ true$\}$. Note that B and C do not participate in this conflict. Non–chronological backtracking will backtrack to try a different value of X–the most recent decision variable in the conflict set. After setting $X =$ false, the algorithm will detect another conflict. This time, the conflict set will be $\{A =$ true$, X =$ true$, Z =$ true$\}$. Since we have exhausted the values for X, non–chronological backtracking will now backtrack to A and set $A =$ false and continue. Note that, this time, the algorithm is able to get out of the conflict without trying different values of B or C. Moreover, it only ran into 2 contradictions in the process.

---

[5]There are several variations of non–chronological backtracking used in CSP and SAT [Gas77, Pro93, MSS99, BS97, ZMMM01].

Non–chronological backtracking partially prevents the algorithm form repeating the same conflict. However, as soon as the algorithm backtracks past every variable in the conflict set (possibly due to a different conflict), it can still repeat the same mistake in the future.

One can address this problem by empowering unit resolution through the addition of clauses to the CNF. For example, the clause $\{\neg A, \neg X\}$ is implied by the above CNF. Moreover, if this clause were present in the CNF initially, then unit resolution would have discovered a contradiction immediately after setting A to true. This leaves the question of how to identify such clauses in the first place. As it turns out, each time unit resolution discovers a contradiction, there is an opportunity to identify a clause which is implied by the CNF and which would be allow unit resolution to realize new implications. This clause is known as a *conflict–driven clause* (or *conflict cluase*) [MSS99, BS97, ZMMM01, BKS04]. Adding it to the CNF will allow unit resolution to detect this contradiction earlier and to avoid the same mistake in the future.[6]

3.6.3. Non–Chronological Backtracking and Conflict–Driven Clauses

The use of non–chronological backtracking in modern SAT solvers is tightly coupled with the derivation of conflict–driven clauses. We will now discuss the method that modern SAT solvers use to derive conflict–driven clauses from conflicts and the specific way they perform non–chronological backtracking. The combination of these techniques makes sure that unit resolution is empowered every time a conflict arises and that the solver will not repeat any mistake. The identification of conflict–driven clauses is done through a process known as *conflict analysis,* which analyzes a trace of unit resolution known as the *implication graph.*

3.6.3.1. Implication Graph

An implication graph is a bookkeeping device that allows us to record dependencies among variable settings as they are established by unit resolution. Figure 3.7 depicts two implication graphs for the knowledge base in (3.1). Figure 3.7(a) is the implication graph after setting variables A, B, C and X to true.

Each node in an implication graph has the form $l/V{=}v$, which means that variable V has been set to value v at level l. Note that a variable is set either by a *decision* or by an *implication.* A variable is set by an implication if the setting is due to the application of unit resolution. Otherwise, it is set by a decision.

Whenever unit resolution is used to derive a variable assignment, it uses a clause and a number of other variable assignments to justify the implication. In this case, we must have an edge into the implied assignment from each of the assignments used to justify the implication. For example, in the implication graph of Figure 3.7(a), variable Y is set to true at Level 3 by using Clause 3 and the variable settings $A{=}$true and $X{=}$true. Hence, we have an edge from $A{=}$true to $Y{=}$true, and another from $X{=}$true to $Y{=}$true. Moreover, both of these edges are labeled with 3, which is the ID of clause used in implying $Y{=}$true.

---

[6]The idea of learning from conflicts originated from the sucessful applications in CSP [SS77, Gen84, Dav84, Dec86, dKW87].
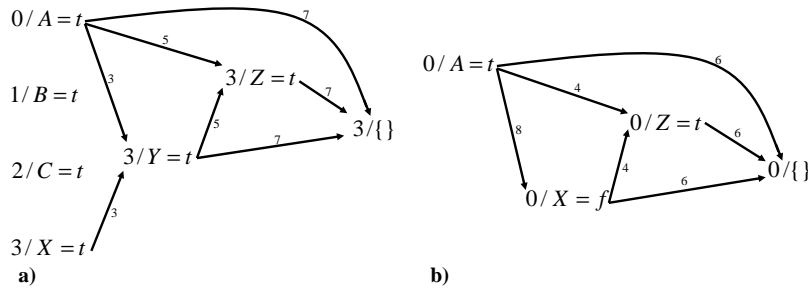
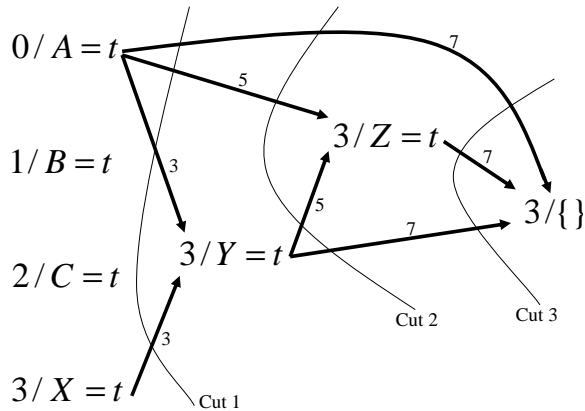**Figure 3.7.** Two implication graphs.

Figure 3.8. Three cuts in an implication graph, leading to three conflict sets.

An implication graph may also have a special node representing a contradiction derived by unit resolution. This is indicated by {} in the implication graph, with its parents being all variable settings that were used in deriving an empty clause. For example, in Figure 3.7(a), the settings A=true, Y=true and Z=true were used with Clause 7 to derive the empty clause.

3.6.3.2. Deriving a Conflict–Driven Clause

The process of deriving a conflict–driven clause involves identifying a conflict set from the implication graph and converting this set into a clause. A conflict set can be obtained in the following manner. Every *cut* in the implication graph defines a conflict set as long as that cut separates the decision variables (root nodes) from the contradiction (a leaf node) [MSS99, ZMMM01, Rya04].[7] Any node (variable assignment) with an outgoing edge that crosses the cut will then be in the conflict set. Intuitively, a conflict set contains assignments that are sufficient to cause the conflict. Figure 3.8 depicts a few cuts in the implication graph of

---

[7]Note that we may have multiple, disconnected components of the implication graph at any point in the search. Our analysis always concerns the component which contains the contradiction node.

Figure 3.7(a), leading to conflict sets $\{A\text{=true}, X\text{=true}\}$, $\{A\text{=true}, Y\text{=true}\}$ and $\{A\text{=true}, Y\text{=true}, Z\text{=true}\}$.

Given the multiplicity of these conflict sets, one is interested in choosing the most useful one. In practice, virtually all modern SAT solvers insist on selecting cuts that involve exactly one variable assigned at the level of conflict for a reason that we become apparent later. For the implication graph in Figure 3.7(a), the sets $\{Y = \text{true}, A = \text{true}\}$ and $\{X = \text{true}, A = \text{true}\}$ meet this criterion. Similarly, for the graph in Figure 3.7(b), $\{A = \text{true}\}$ is a conflict cut with such property. An actual implementation that analyzes the implication graph for the right conflict set with desired properties will be described in Part 1, Chapter 4.

Once a conflict set is identified, a conflict–driven clause can be obtained by simply negating the assignments in the set. For example, if the conflict set is $\{A = \text{true}, B = \text{false}, C = \text{false}\}$, then the conflict–driven clause derived is $\{\neg A, B, C\}$. This clause represents the previously-implicit constraint that the assignments in the conflict set cannot be made simultaneously ($A = \text{true}, B = \text{false}, C = \text{true}$ will now violate this new clause). Conflict–driven clauses generated from cuts that contain exactly one variable assigned at the level of conflict are said to be *asserting* [ZMMM01]. Modern SAT solvers insist on learning only asserting clauses. We will discuss later the properties of asserting clauses that make them useful and even necessary for the completeness of the algorithm. From now on, we assume that every conflict–driven clause is asserting, unless stated otherwise.

3.6.3.3. Learning a Conflict–Driven Clause and Backtracking

Once the conflict–driven clause is derived from the implication graph, it is added to the formula. The process of adding a conflict–driven clause to the CNF is known as *clause learning* [MSS99, BS97, ZMMM01, BKS04]. A key question in this regard is when exactly to add this clause. Consider the termination tree in Figure 3.6 and the left most leaf node corresponding to the CNF $\Delta | A, B, C, X$. Unit resolution discovers a contradiction in this CNF and by analyzing the implication graph in Figure 3.7(a), we can identify the conflict–driven clause $\neg A \vee \neg X$. The question now is: What to do next?

Since the contradiction was discovered after setting variable X to true, we know that we have to at least undo that decision. Modern SAT solvers, however, will undo all decisions made after the *assertion level,* which is the second highest level in a conflict–driven clause. For example, in the clause $\neg A \vee \neg X$, A was set at Level 0 and X was set at level 3. Hence, the assertion level is 0 in this case. If the clause contains only literals from one level, its assertion level is then -1 by definition. The assertion level is special in the sense that it is the deepest level at which adding the conflict–driven clause would allow unit resolution to derive a new implication using that clause. This is the reason why modern SAT solvers would actually backtrack all the way to the assertion level, add the conflict–driven clause to the CNF, apply unit resolution, and then continue the search process. This particular method of performing non–chronological backtracking is referred to as *far-backtracking* [SBK05].

Algorithm 5 DPLL+(CNF Δ): returns UNSATISFIABLE or SATISFIABLE.

1: $D \leftarrow ()$ {empty decision sequence}
2: $\Gamma \leftarrow \{\}$ {empty set of learned clauses}
3: **while true do**
4:    **if** unit resolution detects a contradiction in (Δ, Γ, D) **then**
5:       **if** $D = ()$ **then** {contradiction without any decisions}
6:          **return** UNSATISFIABLE
7:       **else** {backtrack to assertion level}
8:          $\alpha \leftarrow$ asserting clause
9:          $m \leftarrow$ assertion level of clause α
10:         $D \leftarrow$ first m decisions in D {erase decisions $\ell_{m+1}, \ldots$}
11:         add clause α to Γ
12:    **else** {unit resolution does not detect a contradiction}
13:       **if** ℓ is a literal where neither ℓ nor $\neg\ell$ are implied by unit resolution from (Δ, Γ, D) **then**
14:         $D \leftarrow D; \ell$ {add new decision to sequence D}
15:       **else**
16:         **return** SATISFIABLE

3.6.3.4. Clause Learning and Proof Complexity

The advantages of clause learning can also be shown from proof complexity perspective [CR79]. Whenever the CNF formula is unsatisfiable, a SAT solver can be thought of as a resolution proof engine that tries to produce a (short) proof for the unsatisfiability of the formula. From this viewpoint, the goal of the solver is to derive the empty clause. Generating short resolution proofs is difficult to implement in practice, because the right order of clauses to resolve is often hard to find. As a result, SAT solvers only apply a restricted version of resolution, which may generate longer proofs, to allow the solvers to run efficiently.

In [BKS04], the authors show that DPLL-based SAT solvers that utilize clause learning can generate exponentially shorter proofs than those that do not use clause learning. Moreover, it has been shown, in the same paper, that if the solver is allowed to branch on assigned variables and to use unlimited restarts, the resulting proof engine is theoretically as strong as an unrestricted resolution proof system.

3.6.4. Putting It All Together

Algorithm DPLL+ is the final SAT algorithm in this chapter which incorporates all techniques we discussed so far in this section. The algorithm maintains a triplet (Δ, Γ, D) consisting of the original CNF Δ, the set of learned clauses Γ and a decision sequence $D = (\ell_0, \ldots, \ell_n)$, where ℓ_i is a literal representing a decision made at level i. Algorithm DPLL+ starts initially with an empty decision sequence D and an empty set of learned clauses Γ. It then keeps adding decisions to the sequence D until a contradiction is detected by unit resolution. If the contradiction is detected in the context of some decisions, a conflict–driven clause α is constructed and added to the set of learned clauses, while backtracking to the assertion level of clause α. The process keeps repeating until either a contradiction is detected without any decisions (unsatisfiable), or until every variable is set to a value without a contradiction (satisfiable). Part 1, Chapter 4 discusses in details the implementation of this type of SAT solvers.

3.6.4.1. An Example

To consider a concrete example of DPLL+, let us go back to the CNF 3.1 shown again below:

$$\Delta = \begin{array}{ll} 1. & \{A, B\} \\ 2. & \{B, C\} \\ 3. & \{\neg A, \neg X, Y\} \\ 4. & \{\neg A, X, Z\} \\ 5. & \{\neg A, \neg Y, Z\} \\ 6. & \{\neg A, X, \neg Z\} \\ 7. & \{\neg A, \neg Y, \neg Z\} \end{array}$$

DPLL+ starts with an empty decision sequence and an empty set of learned clauses:

$$D = (),$$
$$\Gamma = \{\}.$$

Suppose now that the algorithm makes the following decisions in a row:

$$D = (A{=}\mathsf{true}, B{=}\mathsf{true}, C{=}\mathsf{true}, X{=}\mathsf{true}).$$

Unit resolution will not detect a contradiction until after the last decision $X{=}\mathsf{true}$ has been made. This triggers a conflict analysis based on the implication graph in Figure 3.7(a), leading to the conflict–driven clause

$$8. \quad \{\neg A, \neg X\},$$

whose assertion level is 0. Backtracking to the assertion level gives:

$$D = (A{=}\mathsf{true}),$$
$$\Gamma = \{\{\neg A, \neg X\}\}.$$

Unit resolution will then detect another contradiction, leading to conflict analysis based on the implication graph in Figure 3.7(b). The conflict–driven clause in this case is

$$9. \quad \{\neg A\}$$

with an assertion level of -1. Backtracking leads to:

$$D = (),$$
$$\Gamma = \{\{\neg A, \neg X\}, \{\neg A\}\}.$$

No contradiction is detected by unit resolution at this point, so the algorithm proceeds to add a new decision and so on.
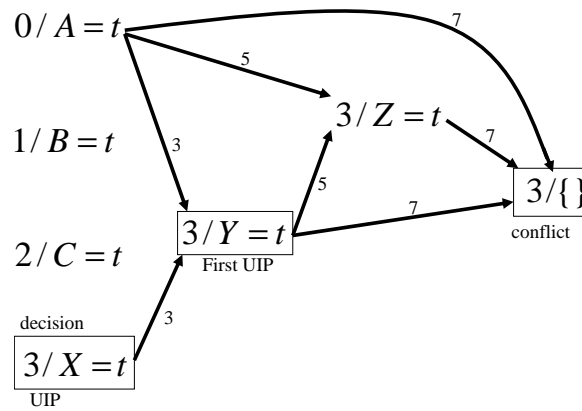
Figure 3.9. An example of a unique implication point (UIP).

3.6.4.2. More on Asserting Clauses

Modern SAT solvers insist on generating asserting conflict–driven clauses as they
are guaranteed to become unit clauses when added to the CNF. Recall that a
conflict–driven clause is asserting if it includes exactly one variable that has been
set at the decision level where the contradiction is found. A popular refinement
on this definition, however, appeals to the notion of *unique implication point
(UIP)* [MSS99, ZMMM01]. In particular, a UIP of a decision level in an impli-
cation graph is a variable setting at that decision level which lies on every path
from the decision variable of that level to the contradiction. Intuitively, a UIP of
a level is an assignment at the level that, by itself, is sufficient for implying the
contradiction. In Figure 3.9, the variable setting $3/Y$=true and $3/X$=true would
be UIPs as they lie on every path from the decision $3/X$=true to the contradiction
$3/\{\}$.

Note that there may be more than one UIP for a given level and contradiction.
In such a case, we will order the UIPs according to their distance to the contradic-
tion node. The first UIP is the one closest to the contradiction. Even though there
are many possible ways of generating asserting conflict–driven clauses from differ-
ent UIPs, asserting clauses that contain the first UIP of the conflict level are pre-
ferred because they tend to be shorter.[8] This scheme of deriving asserting clauses
is called the *1UIP scheme*. One could also derive clauses that contain the first
UIPs of other levels as well. However, the studies in [ZMMM01] and [DHN05b]
showed that insisting on the first UIPs of other levels tends to worsen the solver's
performance. Considering Figure 3.9, the conflict set $\{A$=true$, Y$=true$\}$ contains
the first UIP of the conflict level (Y), leading to the asserting conflict–driven
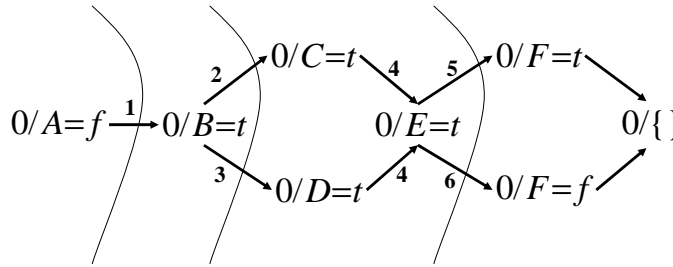clause $\neg A \vee \neg Y$.

Figure 3.10. An implication graph with three different asserting, conflict–driven clauses.

3.6.4.3. Repeating Decision Sequences

Consider now the following CNF:

$$\Delta = \begin{array}{l} 1.\ \{A, B\} \\ 2.\ \{\neg B, C\} \\ 3.\ \{\neg B, D\} \\ 4.\ \{\neg C, \neg D, E\} \\ 5.\ \{\neg E, F\} \\ 6.\ \{\neg E, \neg F\} \end{array}$$

After DPLL+ has decided $\neg A$, unit resolution will detect a contradiction as given by the implication graph in Figure 3.10. There are three asserting clauses in this case, $\{A\}, \{\neg B\}$ and $\{\neg E\}$, where $\{\neg E\}$ is the first UIP clause. Suppose that DPLL+ generates $\{\neg E\}$. DPLL+ will then erase the decision $\neg A$ and add clause $\{\neg E\}$ to the formula. Unit resolution will not detect a contradiction in this case. Moreover, since neither A nor $\neg A$ is implied by unit resolution, DPLL+ may decide $\neg A$ again!

This example illustrates the extent to which DPLL+ deviates from the standard depth–first search used by DPLL. In particular, in such a search no decision sequence will be examined more than once as the search tree is traversed in a systematic fashion, ensuring that no node in the tree will be visited more than once. The situation is different with DPLL+ which does not have a memory of what decision sequences it has considered, leading it to possibly consider the same decision sequence more than once. This, however, should not affect the completeness of the algorithm, because the assignments at the time of future visits will be different. In this sense, even though the algorithm repeats the decision sequence, it does so in a different context. A completeness proof of DPLL+ is presented in [ZM03, Rya04].

3.6.4.4. Deleting Conflict–Driven Clauses

The addition of conflict–driven clauses is quite useful as it empowers unit resolution, allowing it to detect certain contradictions that it could not detect before. The addition of conflict–driven clauses may carry a disadvantage though, as it

---

[8] A short clause is preferable because it potentially allows the solver to prune more search space.

may considerably increase the size of given CNF, possibly causing a space problem. Moreover, the more clauses the CNF has, the more time it takes for the solver to perform unit resolution. In practice, it is not uncommon for the number of added clauses to be much greater than the number of clauses in the original CNF on a difficult problem.

This issue can be dealt with in at least two ways. First, newly added conflict–driven clauses may actually subsume older conflict–driven clauses, which can be removed in this case. Second, one may decide to delete conflict–driven clauses if efficiency or space becomes an issue. In this case, the size, age, and activity of a conflict–driven clause are typically used in considering its deletion, with a preference towards deleting longer, older and less active clauses. Different conflict–driven clause deletion policies, mostly based on some heuristics, have been proposed in different SAT solvers [MSS99, MMZ+01, ES03].

3.6.4.5. Restarts

Another important technique employed by all modern SAT solvers in this category is *restarts* [GSC97]. When a SAT solver restarts, it abandons the current assignments and starts the search at the root of the search tree, while maintaining other information, notably conflict–directed clauses, obtained during earlier search. Restarting is a way of dealing with the heavy-tailed distribution of running time often found in combinatorial search [GSC97, BS00]. Intuitively, restarting is meant to prevent the solver from getting stuck in a part of the search space that contains no solution. The solver can often get into such situation because some incorrect assignments were committed early on and unit resolution was unable to detect them. If the search space is large, it may take very long for these incorrect assignments to be fixed. Hence, in practice, SAT solvers usually restart after a certain number of conflicts is detected (indicating that the current search space is difficult), hoping that, with additional information, they can make better early assignments. The extensive study of different restarting policies in [Hua07b] shows that it is advantageous to restart often. The policy that was found to be empirically best in [Hua07b] is also becoming a standard practice in state-of-the-art SAT solvers.

Note that restarts may compromise the solver's completeness, when it is employed in a solver that periodically deletes conflict–driven clauses. Unless the solver allows the number of conflict–driven clauses to grow sufficiently large or eventually allows a sufficiently long period without restarting, it may keep exploring the same assignments after each restart. Note that both restarting and clause deletion strategies are further discussed in Part 1, Chapter 4.

3.6.5. Other Inference Techniques

Other than the standard techniques mentioned in the previous sub-sections, over the years, other inference techniques have been studied in the context of DPLL-based algorithm. Even though most of these techniques are not present in leading general complete SAT solvers, they could provide considerable speedup on certain families of problems that exhibit the right structure. We discuss some of these techniques in this brief section.

One natural extension of unit resolution is binary resolution. Binary resolution is a resolution in which at least one resolved clause has size two. Binary resolution allows more implications to be realized at the expense of efficiency. In [VG01], Van Gelder presented a SAT solver, 2clVER, that utilized a limited form of binary resolution. Bacchus also studied, in [Bac02], the same form of binary resolution, termed BinRes, in DPLL-based solver called 2CLS+EQ. This solver also employed other rules such as hyper resolution (resolution involving more than two clause at the same time) and equality reduction (replacing equivalent literals). The solver was shown to be competitive with zChaff [MMZ$^+$01] (which only used unit resolution) on some families. Another solver that utilized equivalency reasoning was developed by Li [Li00]. The solver, called EqSatz, employed six inference rules aimed at deriving equivalence relations between literals and was shown to dominate other solvers on the DIMACS 32-bit parity problems [SKM97] (among others). SymChaff [Sab05], developed by Sabharwal, takes advantage of symmetry information (made explicit by the problem creator) to prune redundant parts of the search space to achieve exponential speedup.

3.6.6. Certifying SAT Algorithms

As satisfiability testing becomes vital for many applications, the need to ensure the correctness of the answers produced by SAT solving algorithms increases. In some applications, SAT algorithms are used to verify the correctness of hardware and software designs. Usually, unsatisfiability tells us that the design is free of certain types of bug. If the design is critical for safety or the success of the mission, it would be a good idea to check that the SAT solvers did not produce a buggy result.

Verifying that a SAT solving algorithm produces the right result for satisfiable instances is straightforward. We could require the SAT solving algorithm to output a satisfying assignment. The verifier then needs to check whether every clause in the instance mention at least a literal from the satisfying assignment. Verifying that a SAT solving algorithm produces the right result for unsatisfiable instances is more problematic, however. The complications are caused by the needs to make the verifying procedure simple and the "certificate" concise; see [Van02]. The main method used for showing unsatisfiability of a formula is to give a resolution proof of the empty clause from the original set of clauses.

One of the most basic certificate formats is the explicit resolution derivation proof. A certificate in this format contains the complete resolution proof of unsatisfiability (the empty clause). The proof is simply a sequence of resolutions. First of all, all variables and clauses in the original formula are indexed. Each resolution is made explicit by listing the indices of the two operands, the index of the variable to be resolved on, and the literals in the resolvent. This certificate format was referred to as `%RES` and was used for the verification track of the SAT'05 competition [BS05].

A closely related format called resolution proof trace (`%RPT`) [BS05] does not require the resolvent of each resolution to be materialized. Hence, the certificate will only list the clauses to be resolved for each resolution operation.

Zhang and Malik proposed in [ZM03] a less verbose variant which relies on the verifier's ability to reconstruct implication graphs in order to derive learned

clauses. According to this format, the certificate contains, for each learned clause generated by the solving algorithm, the indices of all clauses that were used in the derivation of the conflict–driven clause. The verifier then has the burden of reconstructing the conflict–driven clauses that are required to arrive at the proof of the empty clause.

Another certificate format was proposed by Goldberg and Novikov in [GN03]. This format lists all conflict–driven clauses generated by the SAT solving algorithm in the order of their creation. This type of certificate employs the fact that if a learned clause C is derived from the CNF Δ, then applying unit resolution to $\Delta \wedge \neg C$ will result in a contradiction. By listing every conflict–driven clause (including the empty clause at the end) in order, the verifier may check to make sure that each conflict–driven clause can actually be derived from the set of preceding clauses (including the original formula) by using unit resolution.

3.7. Conclusions

We have discussed in this chapter various complete algorithms for SAT solving. We categorized these algorithms based on their approaches for achieving completeness: by existential quantification, by inference rules, and by search. While much work has been done in each category, nowadays, it is the search-based approach that receives the most attention because of its efficiency and versatility on many real-world problems. We presented in this chapter the standard techniques used by leading contemporary SAT solvers and some less common inference techniques studied by others. Current research in this area focuses on both theory and applications of satisfiability testing. Recently, however, there is a growing emphasis on applying a SAT solver as a general-purpose problem solver for different problem domains (see Part 3). As a result, a branch of current research focuses on finding efficient ways to encode real-world problems compactly as CNF formulas. When it comes to using a SAT solver as a general-purpose solver, efficiency is the most important issue. Consequently, techniques that are theoretically attractive may not necessarily be adopted in practice, unless they can be efficiently implemented and shown to work on a broad set of problems.

References

[AMS01] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Faster sat and smaller bdds via common function structure. In *Technical Report #CSE-TR-445-01*. University of Michigan, November 2001.

[AV01] A. S. M. Aguirre and M. Y. Vardi. Random 3-SAT and BDDs: The plot thickens further. In *Principles and Practice of Constraint Programming*, pages 121–136, 2001.

[Bac02] F. Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *AAAI/IAAI*, pages 613–619, 2002.

[BCM$^+$90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Com-*

puter Science, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[Bie07] A. Biere. Picosat versions 535, 2007. Solver description, SAT Competition 2007.

[BKS04] P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BS97] R. J. J. Bayardo and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

[BS00] L. Baptista and J. P. M. Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *Principles and Practice of Constraint Programming*, pages 489–494, 2000.

[BS05] D. L. Berre and L. Simon, 2005. SAT'05 Competition Homepage, http://www.satcompetition.org/2005/.

[CR79] S. A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. *J. Symb. Log.*, 44(1):36–50, 1979.

[CS00] P. Chatalic and L. Simon. ZRes: The old Davis-Putnam procedure meets ZBDDs. In D. McAllester, editor, *17th International Conference on Automated Deduction (CADE'17)*, number 1831, pages 449–454, 2000.

[Dar01] A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 126(1-2):5–41, 2001.

[Dav84] R. Davis. Diagnostic reasoning based on structure and behavior. *Artif. Intell.*, 24(1-3):347–410, 1984.

[Dec86] R. Dechter. Learning while searching in constraint-satisfaction-problems. In *AAAI*, pages 178–185, 1986.

[Dec90] R. Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990.

[Dec97] R. Dechter. Bucket elimination: A unifying framework for processing hard and soft constraints. *Constraints: An International Journal*, 2:51–55, 1997.

[DHN05a] N. Dershowitz, Z. Hanna, and A. Nadel. A clause-based heuristic for sat solvers. In *Proceedings of SAT 2005*, pages 46–60, 2005.

[DHN05b] N. Dershowitz, Z. Hanna, and A. Nadel. Towards a better understanding of the functionality of a conflict-driven sat solver. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing*, 2005.

[dKW87] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artif. Intell.*, 32(1):97–130, 1987.

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

[DP60] M. Davis and H. Putnam. A computing procedure for quantification

theory. *Journal of the ACM*, 7:201–215, 1960.

[DR94] R. Dechter and I. Rish. Directional resolution: The davis putnam procedure, revisited. In *Principles of Knowledge Representation (KR-94)*, pages 134–145, 1994.

[ES03] N. Eén and N. Sörensson. An extensible sat-solver. In *Proceedings of SAT'03*, pages 502–518, 2003.

[FKS⁺04] J. Franco, M. Kouril, J. Schlipf, J. Ward, S. Weaver, M. Dransfield, and W. M. Vanfleet. Sbsat: a state-based, bdd-based satisfiability solver. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, pages 398–410, 2004.

[Gas77] J. Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *IJCAI*, page 457, 1977.

[Gen84] M. R. Genesereth. The use of design descriptions in automated diagnosis. *Artif. Intell.*, 24(1-3):411–436, 1984.

[GN02] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat-solver. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, page 142, Washington, DC, USA, 2002. IEEE Computer Society.

[GN03] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for cnf formulas. In *Proceedings of Design, Automation and Test in Europe (DATE2003).*, 2003.

[GSC97] C. P. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *Principles and Practice of Constraint Programming*, pages 121–135, 1997.

[GW00] J. F. Groote and J. P. Warners. The propositional formula checker heerhugo. *Journal of Automated Reasoning*, 24:101–125, 2000.

[Har96] J. Harrison. Stalmårck's algorithm as a hol derived rule. In *Proceedings of TPHOLs'96*, pages 221–234, 1996.

[HD04] J. Huang and A. Darwiche. Toward good elimination orders for symbolic sat solving. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 566–573, 2004.

[HKB96] R. Hojati, S. C. Krishnan, and R. K. Brayton. Early quantification and partitioned transition relations. In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 12–19, Washington, DC, USA, 1996. IEEE Computer Society.

[Hua07a] J. Huang. A case for simple sat solvers. In *CP*, pages 839–846, 2007.

[Hua07b] J. Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 2318–2323, 2007.

[JS04] H. Jin and F. Somenzi. Circus : Hybrid satifiability solver. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004.

[Li00] C. M. Li. Integrating equivalency reasoning into davis-putnam procedure. In *AAAI: 17th National Conference on Artificial Intelligence.* AAAI / MIT Press, 2000.

[LLM03] J. Lang, P. Liberatore, and P. Marquis. Propositional independence: Formula-variable independence and forgetting. *J. Artif. Intell. Res. (JAIR)*, 18:391–443, 2003.

[Mar00] P. Marquis. Consequence finding algorithms. *Handbook on Defeasible Reasonging and Uncertainty Management Systems, Volume 5: Algorithms for Uncertain and Defeasible Reasoning*, pages 41–145, 2000.

[MM02] D. Motter and I. Markov. A compressed breadth-first search for satisfiability. In *Proceedings of ALENEX 2002.*, 2002.

[MMZ$^+$01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. *39th Design Automation Conference (DAC)*, 2001.

[MSS99] J. Marques-Silva and K. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, (5):506–521, 1999.

[PD07] K. Pipatsrisawat and A. Darwiche. Rsat 2.0: Sat solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, 2007.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, (9(3)):268–299, August 1993.

[PV04] G. Pan and M. Vardi. Symbolic techniques in satisfiability solving. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004.

[RAB$^+$95] R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient bdd algorithms for fsm synthesis and verification. In *Proceedings of IEEE/ACM International Workshop on Logic Synthesis, Lake Tahoe, USA, May 1995.*, 1995.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[Rya04] L. Ryan. Efficient Algorithms for Clause-Learning SAT Solvers. Master's thesis, Simon Fraser University, 2004.

[Sab05] A. Sabharwal. Symchaff: A structure-aware satisfiability solver. In *AAAI*, pages 467–474, 2005.

[SB06] C. Sinz and A. Biere. Extended resolution proofs for conjoining bdds. In *CSR*, pages 600–611, 2006.

[SBK05] T. Sang, P. Beame, and H. A. Kautz. Heuristics for fast exact model counting. In *Proceedings of SAT'05*, pages 226–240, 2005.

[SKM97] B. Selman, H. A. Kautz, and D. A. McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 50–54, 1997.

[SS77] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9, October 1977.

[SS90] G. Stålmarck and M. Säflund. Modeling and verifying systems and software in propositional logic. In *B.K. Daniels, editor, Safety of Computer Control Systems, 1990 (SAFECOMP'90)*, 1990.

[Stå94] G. Stålmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from boolean formula., 1994. United States Patent number 5,276,897; see also Swedish Patent 467 076.

[TSL⁺90] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. In *Proceedings of IEEE International Conference on Computer-Aided Design (ICCAD-90).*, pages 130–133, 1990.

[TY84] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.

[Van02] A. Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *AMAI*, 2002.

[VG01] A. Van Gelder. Combining preorder and postorder resolution in a satisfiability solver. In *IEEE/ASL LICS Satisfiability Workshop*, Boston, 2001.

[Wid96] F. Widebäck. Stålmarck's notion of n-saturation. Technical Report NP-K-FW-200, Prover Technology AB, 1996.

[ZM03] L. Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10880, Washington, DC, USA, 2003. IEEE Computer Society.

[ZMMM01] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.