Lecture Notes for

Logic and Computability

Course Number: IND04033UF

Contact

Bettina Könighofer
Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology, Austria
bettina.koenighofer@iaik.tugraz.at

TU Graz
Graz University of Technology

Table of Contents

3 SAT Solvers

Given a propositional logic formula, a SAT solver determines whether there is an assignment to the variables in the formula that satisfies the formula. During the past decade, SAT solvers have been the subject of remarkable efficiency improvements. Many practical problems can be formulated as SAT instances, e.g., in the areas of high-level planning, scheduling, artificial intelligence, circuit testing, and software modeling. The best-performing implementations of SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm with conflict-driven clause learning (CDCL). In this chapter, we will discuss the basics of the DPLL algorithm with CDCL.

3.1 The SAT-Problem

We start this chapter by formalizing the SAT problem.

Definition - SAT-Problem. Given a propositional logic formula, the problem of determining whether there exists a model such that the formula evaluates to true is called the *Boolean Satisfiability Problem*, or the SAT problem for short.

The SAT problem represents the first decision problem to be proven *NP-complete*. Given its NP-completeness, it is very unlikely that there exists any polynomial algorithm for SAT. Nevertheless, there exists algorithm able to solve many interesting SAT instances.

3.2 The DPLL Algorithm

Introduced by Martin **D**avis, Hilary **P**utnam, Donald W. **L**oveland and George **L**ogemann in 1962, the DPLL algoritm forms the basis for most modern SAT solvers. The DPLL algorithm operates on formulas given in *conjunctive normal form (CNF)* and is a complete, backtracking-based *binary search* algorithm.

In its basis form, the algorithm searches for a satisfying total or partial assignment to the variables that makes the formula true. The algorithm starts with an empty assignment. Then, the algorithm assigns a truth value to a first variable, this splits the search space into two parts. Next, the algorithm assigns a value to second variable, and so on. Every assignment splits the search space into two parts, which are then searched recursively. In case the search in one of the parts fails, the algorithm backtracks and proceeds with a different part of the search space.

Notation

We introduce the following notation that we use in the algorithm for updating a variable assignment.

- ϕ is formula in CNF, e.g., $\phi = (a \vee b \vee \neg d) \wedge c$
- A is an assignment of truth values to variables in ϕ.
  - A can be given in set representation, e.g., $A := \{\neg a, b, d\}$, or
  - A can be given as conjunction of literals, e.g., $A := \neg a \wedge b \wedge d$.
  - A is called a *total (complete)* assignment if it assigns a value to all variables in ϕ.
  - A is called a *partial* assignment if is assigns values to some but not all variables in ϕ.
  - A clause can be *satisfied* by an assignment A, *conflicting* with the assignment A (all variables in the clause are given the opposite value in A), or there are some unassigned literals in A left.
- $\phi[A]$: ϕ with all variables assigned according to the corresponding truth values in A.
  - E.g., $\phi[A] = (\bot \vee \top \vee \neg\top) \wedge c = c$.

Basis DPLL Algorithm - Backtracking Binary Search

The basic algorithm for a recursive implementation of binary search for a satisfying assignment is given as in Listing 3.1.

The procedure starts the recursion with the empty assignment, i.e, $A := true$. In each recursion, the procedure first checks whether the current assignment A makes the formula ϕ false. To check this, we only need to check if *there is a clause that is conflicting with the assignment A*. If so, no extension of A will satisfy ϕ, and the procedure returns UNSAT. If not, we check whether the current assignment already satisfies all clauses in which case the algorithm returns SAT.

```
# sat($\phi$, $A$) = True iff $\phi$[A] is satisfiable
# sat($\phi$, $true$) = True iff $\phi$ is satisfiable
def sat($\phi$, $A$):
  if $\phi[A] = false$:
    return False
  if $\phi[A] = true$:
    return True

  # Some unassigned variables left
  l = pick unassigned variable
  if sat($\phi$, $A \wedge l$)
    return True
  if sat($\phi$, $A \wedge \neg l$)
    return True
  return False
```

Listing 3.1: DPLL algorithm

If A is not total, then there is at least one variable that is not yet assigned. The algorithm picks one of these unassigned variables, and chooses a truth value for it. The choice of the variable and the truth value is immaterial for the correctness of the algorithm, and can thus be a heuristic choice. We update the assignment to the new choice accordingly and perform the recursive call with the new assignment $A \wedge l$. The recursive call may have one of two outcomes: it may succeed in finding a satisfying assignment, in which case the recursion is aborted. In case the recursive call returns, we have thus failed to find a satisfying assignment in the part of the search space given by the assignment $A \wedge l$. The binary search algorithm then flips the truth value of the last decision to get the new assignment $A \wedge \neg l$, and performs a second recursive call in order to search the other half of the search space.

The search with $A \wedge \neg l$ may fail as well, in which case the recursive call backtracks. In case we return to the top level of the recursion, we have exhausted the search tree without finding a satisfying assignment. We can then conclude that the formula is unsatisfiable.

Definition - Decision and Decision Level. Each time the algorithm assigns a new value to a literal l it makes a *decision*. The current depth of the binary search tree is called the *decision level*.

CNF as a Set of Clauses

We use a set of clauses C_ϕ as a shorthand for a formula ϕ in CNF. For instance, we write

$$C_\phi := \{\{\neg a, b\}, \{\neg b, c\}, \{\neg c, \neg a\}\}$$

to mean

$$\phi := (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee \neg a).$$

Using sets of clauses to represent formulas in CNF, we can efficiently implement *setting literals* and *checking for unsatisfiability.*

Setting Literals: In order to compute $\phi[l]$ for a literal l using set representation we perform two steps:

1. Remove all clauses that contain l, because these clauses evaluate to *true.*
2. Remove all literals $\neg l$ from all remaining clauses, because these literals evaluate to *false.*

Example. Given $\phi := (a \vee b \vee c) \wedge d$ and $A := \{a\}$. Compute $\phi[A]$ using set representation.

Formula ϕ in set representation gives us $C_\phi := \{\{a, b, c\}, \{d\}\}$. Since $a \in \{a, b, c\}$, we remove the first clause from C_ϕ, resulting in $C_\phi := \{\{d\}\}$ or $\phi[A] = d$.

Example. Given $\phi := (\neg a \vee b \vee c) \wedge d$ and $A := \{a\}$. Compute $\phi[A]$ using set representation.

Formula ϕ in set representation gives us $C_\phi := \{\{\neg a, b, c\}, \{d\}\}$. Under $A := \{a\}$ we remove $\neg a$ from the first clause, resulting in $C_\phi := \{\{b, c\}, \{d\}\}$. Therefore, $\phi[A] := (b \vee c) \wedge d$.

Checking for unsatisfying assignments. A formula ϕ under an assignment A evaluates to *false*, if using set representation at least one *clause becomes empty.*

Checking for satisfying assignments. A formula ϕ under an assignment A evaluates to *true*, if the resulting *set of clauses becomes empty*, i.e., all clauses evaluate to *true* and are removed from C.

Execution of basic DPLL-Algorithm

As an example, we execute the basic DPLL algorithm to find a satisfying assignment for the formula

$$\phi := (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee \neg a).$$

First, we represent ϕ using a set of clauses: $C_\phi := \{\{\neg a, b\}, \{\neg b, c\}, \{\neg c, \neg a\}\}$. In this example, the algorithm selects the literals for the next decision in *lexicographical order*, starting with the *positive value*, i.e., $(a < \neg a < b < \neg b < \dots)$.

Step	1	2	3	4	5	6	7	8	9
Decision Level	0	1	2	3	3	2	1	2	3
Assignment	-	a	a, b	a, b, c	$a, b, \neg c$	$a, \neg b$	$\neg a$	$\neg a, b$	$\neg a, b, c$
Cl. 1: $\neg a, b$	$\neg a, b$	b	✓	✓	✓	{} ✗	✓	✓	✓
Cl. 2: $\neg b, c$	$\neg b, c$	$\neg b, c$	c	✓	{} ✗	✓	$\neg b, c$	c	✓
Cl. 3: $\neg c, \neg a$	$\neg c, \neg a$	$\neg c$	$\neg c$	{} ✗	✓	$\neg c$	✓	✓	✓
Decision	a	b	c	$\neg c$	$\neg b$	$\neg a$	b	c	SAT

Table 3.1: Execution of the DPLL algorithm

Note, that the *order* is arbitrary and may also be different (e.g. *lexicographical order*, starting with the *negative value*: $(\neg a < a < \neg b < b < ...)$ or *non-lexicographical order*, starting with the *positive value*: $(b < \neg b < a < \neg a < c < \neg c < ...)$).

Table 3.1 states the individual steps performed by the algorithm. We give the current time step, the current decision level, the current assignment, the status of the individual clauses under the current assignment, and the next decision that the algorithm makes. The goal is to find an assignment that satisfies all clauses. In the individual steps we keep track about what clauses are already satisfied by the current assignment (we mark them with a ✓), and for the yet unsatisfied clauses which literals we still have left to satisfy the clause. We mark a contradicting clause with the empty set. A contracting clause causes the algorithm to perform the backtracking step. In detail, the following happens:

1. In step 1, the algorithm starts at decision level 0 and an empty assignment $A = \{\}$. The first decision of the algorithm is a.
2. In step 2, the decision level is incremented, the current partial assignment $A = \{a\}$. We evaluate the clauses under A and can remove $\neg a$ from the first clause and the third clause. Next, the algorithm checks whether there is *an empty clause* (which would mean that a clause is conflicting with the current assignment) or whether *all clauses are satisfied* under the assignment. Since neither is the case, the algorithm makes the second decision: b.
3. In step 3, we increment the decision level and $A = \{a\} \cup b = \{a, b\}$. Having b in A satisfies the first clause and the clause can be removed from the set of clauses that still need to be satisfied. From the second clause, we have to remove the $\neg b$ literal. Since the current assignment is neither satisfying nor conflicting, the algorithm makes the next decision c.
4. In step 4, the decision level is 3, and the assignment $A = \{a, b, c\}$ results in a conflict. The new assigned literal c makes the second clause true, but set representing the third clause becomes empty. The algorithm backtracks to decision level 2, flips its last decision and assigns the negative value to the variable c next.
5. In step 5, the decision level is again 3, since it performed a back tracking

step in the tree, before making the decision $\neg c$. The new assignment is $A = \{a, b, \neg c\}$. This assignment again results in a conflict, this time the second clause becomes empty. The algorithm has to backtrack to decision level 1, and flips the value for the variable b in the assignment, i.e, the next decision is $\neg b$.

6. The assignment $A = \{a, \neg b\}$ results in an empty first clause. Therefore, the algorithm backtracks one more time to decision level 0, and makes the decision $\neg a$.
7. $A = \{\neg a\}$ immediately satisfies the first and third clause and only the second clause remains to be satisfied. Following the lexicographical order, the next decision is b.
8. In step 8, A is $\{\neg a, b\}$, therefore we remove $\neg b$ from clause 2. The next decision is c.
9. In step 9, the total assignment $\{\neg a, b, c\}$ satisfies all three clauses. Therefore, the algorithm returns SAT and the satisfying assignment $A = \{\neg a, b, c\}$.

Figure 3.1 illustrates the binary search on which the algorithm performed the search and shows the decision levels.

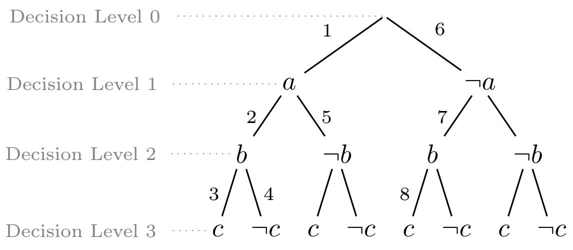

Figure 3.1: Binary Search Tree

Decision Heuristic

SAT solvers that implement DPLL make heuristic choices when they need to pick a variable and a value for a decision. There are several commonly used methods for making such decisions. A very basic way of performing decisions is a greedy approach that picks the decision that satisfies the largest number of clauses. This heuristic can be further improved in several different ways.

In this course for simplicity, we will always define the order for decision making for every example.

3.2.1 DPLL with Boolean Constraint Propagation (BCP)

We now discuss a series of improvements over the basic binary search algorithm. A standard optimization is called Boolean Constraint Propagation (BCP).

Definition - Unit Clause. A clause c is said to be a unit clause under some assignment A if the following two conditions hold:

1. The clause c is not satisfied by A.
2. All but one of the variables in c are given a value by A.

Therefore, there is a single literal left in the set representing the clause under the assignment.

The key observation is that in order to extend A to a satisfying assignment for any formula that contains a unit clause c, we must make the following assignment:

- If $l \in c$, then this literal must be set accordingly resulting in $A = A \wedge l$.

A possible implementation is given in Listing 3.2, which applies the unit rule exhaustively before making a decision. We say that an assignment of a variable that is made due to a unit clause is *not a decision* but *an implication*. Therefore, by applying BCP, *the decision level is not increased*.

```
# sat(ϕ, A) = True iff ϕ[A] is satisfiable
# sat(ϕ, true) = True iff ϕ is satisfiable
def sat(ϕ, A):
  while unit clause occurs:
    # l is only unassigned literal in unit clause
    A = A ∧ l

  if ϕ[A] = false:
    return False
  if ϕ[A] = true:
    return True

  # Some unassigned variables left
  l = pick unassigned variable
  if sat(ϕ, A ∧ l)
    return True
  if sat(ϕ, A ∧ ¬l)
    return True
  return False
```

Listing 3.2: DPLL algorithm with BCP

Example: Execution of DPLL-Algorithm with BCP

Given the formula $\phi := (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee \neg a)$ and the decision heuristic $(a < \neg a < b < \neg b < \dots)$, we execute DPLL with BCP and show the individual steps in Table 3.2.

The algorithm now checks whether there exists a unit clause before making a decision. In the table we assume, that we can extend the current assignment only by one literal per time step, therefore we can either make an implication because of a unit clause or make a decision if there is no unit clause.

In this example we apply BCP in step 2, 3, and 6 and we do not increase the decision level in these cases. Due to BCP, the algorithm was able to find with the same decision heuristic a satisfying assignment with a fewer number of time steps.

Step	1	2	3	4	5	6	7
Decision Level	0	1	1	1	1	2	2
Assignment	-	a	a, b	a, b, c	$\neg a$	$\neg a, b$	$\neg a, b, c$
Cl. 1: $\neg a, b$	$\neg a, b$	b	✓	✓	✓	✓	✓
Cl. 2: $\neg b, c$	$\neg b, c$	$\neg b, c$	c	✓	$\neg b, c$	c	✓
Cl. 3: $\neg c, \neg a$	3	$\neg c$	$\neg c$	$\{\}$ ✗	✓	✓	✓
BCP	-	b	c	-	-	c	-
Decision	a	-	-	$\neg a$	b	-	SAT

Table 3.2: Execution of the DPLL algorithm with BCP

3.2.2 DPLL with Pure Literals (PL)

The next optimization that we consider is the pure literal rule and is one of the standard techniques used in DPLL-based SAT solvers.

Definition - Pure Literal. A literal is *pure* if its negation does not appear in the formula.

As example consider the set of clauses $C_\phi := \{\{a, \neg b, c\}\{a, \neg c\}, \{b, \neg c\}\}$. The literal a is pure, since $\neg a$ is not contained in any clause.

The *pure literal rule* repeatedly sets a pure literal to true, until there are no more pure literals, thereby satisfying all clauses that contain the pure literal. Note that the order in which the pure literals are chosen does mot affect whether the procedure succeeds.

The pseudo-code of DPLL including the rule for pure literals is given in Listing 3.3. If an unassigned literal becomes pure, the algorithm sets this literal to *true*. Similar than for PCB, if a literal is set because of the pure literals rule, it does not count as decision and does not increase the decision level.

```
# sat(ϕ, A) = True iff ϕ[A] is satisfiable
# sat(ϕ, true) = True iff ϕ is satisfiable
def sat(ϕ, A):
  while unit clause occurs:
    # l is only unassigned literal in unit clause
    $A = A \wedge l$

  while pure literal l exists:
    $A = A \wedge l$

  if $\phi[A] = false$:
    return False
  if $\phi[A] = true$:
    return True

  # Some unassigned variables left
  l = pick unassigned variable
  if sat(ϕ, $A \wedge l$)
    return True
  if sat(ϕ, $A \wedge \neg l$)
    return True
  return False
```

Listing 3.3: DPLL algorithm with Pure Literals

Example: Execution of DPLL-Algorithm with Pure Literals

We apply the DPLL-algorithm with the pure-literals rule on the same formula $\phi := (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee \neg a)$ and decision heuristic $(a < \neg a < b < \neg b < \dots)$. The formula ϕ contains the pure literal $\neg a$, therefore the pure-literal rule is applied in the first step and satisfies the clauses 1 and 3. The single remaining second clause gives us two pure literals: $\neg b$ and c. The algorithm picks $\neg b$. The assignment $A = \{\neg a, \neg b\}$ satisfies all clauses.

Step	1	2	3
Decision Level	0	0	0
Assignment	-	$\neg a$	$\neg a, \neg b$
Cl. 1: $\neg a, b$	$\neg a, b$	✓	✓
Cl. 2: $\neg b, c$	$\neg b, c$	$\neg b, c$	✓
Cl. 3: $\neg c, \neg a$	$\neg c, \neg a$	✓	✓
BCP	-	-	-
PL	$\neg a$	$\neg b$	-
Decision	-	-	SAT

Table 3.3: Execution of the DPLL-algorithm with BCP and PL.

Note that when filling out a column in the table, we go from top to bottom. This means, first we try to apply BCP. If no unit clause exist, we try to apply the pure-literal rule. Only if there is no pure literal either, we make a decision and increase the decision level.

3.2.3 Conflict-driven Clause Learning (CDCL)

The idea of conflict-driven clause learning is not to repeat steps that lead to a conflict.

Let us assume we are given a formula in CNF formula with the clauses

$$C_\phi := \{\{a, \neg c\}, \{b, \neg c\}, \{\neg a, \neg b, c\}, \{\neg a, \neg b\}, \{\neg a, b\}\{a, \neg b\}, \{a, b\}$$

and iterate trough the algorithm with the order $\neg c < c < \neg a < a < \neg b < b$.

When executing the DPLL-algorithm with the given decision heuristic, then the first decision is to set c to false and the second decision is to set a to false. Under the assignment $A = \{\neg a, \neg c\}$, the sixth and seventh clauses become unit clauses. The clause $\{a, \neg b\}$ implies $b = false$. But with $b = false$, the last clause $\{a, b\}$ becomes conflicting. If we set $b = true$, the clause $\{a, \neg b\}$ becomes conflicting. We therefore have to revert our last decision, and flip a to true. These execution steps are shown in Table 3.2.

Step	1	2	3	4	5
Decision Level	0	1	2	2	2
Assignment	-	$\neg c$	$\neg a, \neg c$	$\neg a, \neg b, \neg c$	$a, \neg c$
Cl. 1: $a, \neg c$	1	✓	✓	✓	...
Cl. 2: $b, \neg c$	2	✓	✓	✓	...
Cl. 3: $\neg a, \neg b, c$	3	$\neg a, \neg b$	✓	✓	...
Cl. 4: $\neg a, \neg b$	4	4	✓	✓	...
Cl. 5: $\neg a, b$	5	5	✓	✓	...
Cl. 6: $a, \neg b$	6	6	$\neg b$	✓	...
Cl. 7: a, b	7	7	b	{} ✗	...
BCP	-	-	$\neg b$	-	...
PL	-	-	-	-	...
Decision	$\neg c$	$\neg a$	-	a	...

Figure 3.2: DPLL-algorithm with BCP and PL.

Assume, that we proceed in the search with $A = \{\neg c, a\}$, but the search would again fail. The algorithm needs to backtrack to the decision on c, i.e., it will assign c to true after backtracking. The partial assignment is now $A = \{c\}$. *The algorithm will now re-do all the steps from before.* It will repeat the decision $a = false$ which result in conflicts, tracks back, flips the decision to $a = true$, and the search again fails. This effort is clearly wasted: we run into the same conflicts that we have seen before, since the variable c has nothing to do with the conflict. Figure 3.3 illustrates that searching the left part of the search tree was unnecessary effort.

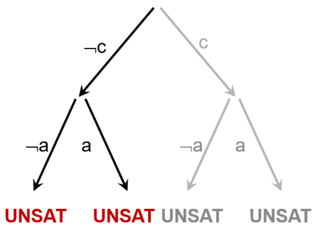

Figure 3.3: Search Tree for DPLL.

Conflict Graph

In order to not repeat steps that lead to the same conflicts, the data structure that CDCL maintains for this is the *conflict graph*, also called implication graph.

The conflict graph is a directed graph with labeled nodes. It is constructed as follows:

1. For every decision, create a new node that is labeled with that decision.
2. For every implication detected by BCP, create a new node that is labeled with that implication. Every implication detected by BCP is triggered by a unit clause. Create an edge from the nodes that correspond to the literals in the unit clause to the new node. We label the edge with the unit clause.
3. In case of a conflict, add a node labeled with $\perp$, and add edges from the nodes that correspond to the unit clauses causing the conflict. The node is called the conflict node.

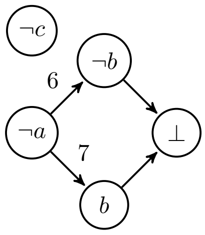

Figure 3.4: Conflict graph for step 4 of Table 3.2.

Figure 3.4 gives the state of the implication graph for our example when the first conflict is reached. The root nodes of the graph are the decisions on c and a. The node on the right-hand side labeled with $\perp$ is the node for the conflict. The inner nodes, labeled with values for $\neg b$ and b, were created for the implications detected by BCP.

The conflict graph reveals that the decision on the variable c does not has an impact on the conflict.

Clause Learning

CDCL generates *new clauses* from the existing set of clauses by traversing the implication graph. SAT solver implement different strategies on how the new clauses are generated and how many will be added per conflict.

In this course, we apply the following simple rule for generating a learned clause for a conflict:

Once we reach a conflict, we analyze the conflict by drawing a conflict graph. We form the new learned clause by *negating all decisions that are involved in the conflict.* These negated literals form the new learned clause and is added to the set of clauses.

Example. If we analyse the conflict in Figure 3.4, the learned clause would be $\{a\}$.

Backtracking Level: After a conflict is reached and a new learned clause is added, the DPLL algorithm needs to perform backtracking. Again, different strategies exists on which level the SAT solver backtracks after adding a learned clause.

In this course we apply the following rule for deciding the backtracking level:

After reaching a conflict and adding a learned clause, we backtrack to the level where the *newly added clause becomes a unit clause.* This way, the newly added clause is immediately used with BCP.

Example 1: Execution of DPLL-Algorithm with CDCL

We have given a similar formula ϕ as before with the clauses

$$C_\phi := \{\{a, \neg c\}, \{b, \neg c\}, \{\neg a, \neg b, c\}, \{\neg a, \neg b\}, \{\neg a, b\}\{a, \neg b\}, \{a, b\}$$

and decision heuristic $\neg c < c < \neg a < a < \neg b < b$. Table 3.4 states the individual steps. The conflict is reached in time step 4 and results in the conflict graph shown in Figure 3.5. The learned clause a is added, and the algorithm continues at decision level 0 and applies BCP.

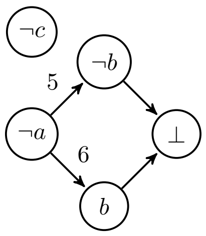

Figure 3.5: Conflict graph for step 4 of Table 3.4.

Step	1	2	3	4	(1)	5	6	7
Decision Level	0	1	2	2	0	0	0	0
Assignment	-	$\neg c$	$\neg a, \neg c$	$\neg a, \neg b, \neg c$	-	a	$a, \neg b$	$a, \neg b, \neg c$
Cl. 1: $a, \neg c$	1	✓	✓	✓	1	✓	✓	✓
Cl. 2: $b, \neg c$	2	✓	✓	✓	2	2	$\neg c$	✓
Cl. 3: $\neg a, \neg b, c$	3	$\neg a, \neg b$	✓	✓	3	$\neg b, c$	✓	✓
Cl. 4: $\neg a, \neg b$	4	4	✓	✓	4	$\neg b$	✓	✓
Cl. 5: $a, \neg b$	5	5	$\neg b$	✓	5	✓	✓	✓
Cl. 6: a, b	6	6	b	{} ✗	6	✓	✓	✓
Cl. 7: a				learned a	7	✓	✓	✓
BCP	-	-	$\neg b$	-	a	$\neg b$	$\neg c$	-
PL	-	-	-	-	-	-	-	-
Decision	$\neg c$	$\neg a$	-	-	-	-		SAT

Table 3.4: DPLL algorithm with clause learning. Clauses written in gray denote a learned clause.

Example 2: Execution of DPLL-Algorithm with Clause Learning

We have given the following formula in CNF in set representation:
$C_\phi = \{a, \neg c, \neg e\}, \{\neg a, \neg e\}, \{b, e\}, \{\neg b, d, e\}, \{\neg b, \neg d\}, \{c, \neg d\}, \{c, d\}$ with the decision heuristic $\neg a < a < \neg b < b < \neg c < c < \neg d < d < \neg e < e$.

Step	1	2	3	4	5	6	(2)	7
Decision Level	0	1	2	2	2	2	1	1
Assignment	-	$\neg a$	$\neg a, \neg b$	$\neg a, \neg b, e$	$\neg a, \neg b, \neg c, e$	$\neg a, \neg b, \neg c, \neg d, e$	$\neg a$	$\neg a, b$
Cl. 1: $a, \neg c, \neg e$	1	$\neg c, \neg e$	$\neg c, \neg e$	$\neg c$	✓	✓	$\neg c, \neg e$	$\neg c, \neg e$
Cl. 2: $\neg a, \neg e$	2	✓	✓	✓	✓	✓	✓	✓
Cl. 3: b, e	3	3	e	✓	✓	✓	3	✓
Cl. 4: $\neg b, d, e$	4	4	✓	✓	✓	✓	4	d, e
Cl. 5: $\neg b, \neg d$	5	5	✓	✓	✓	✓	5	$\neg d$
Cl. 6: $c, \neg d$	6	6	6	6	$\neg d$	✓	6	6
Cl. 7: c, d	7	7	7	7	d	$\{\}$ ✗	7	7
Cl. 8: a, b						$a \vee b$	b	✓
Cl. 9: a								
BCP	-	-	e	$\neg c$	$\neg d$	-	b	$\neg d$
PL	-	-	-	-	-	-	-	-
Decision	$\neg a$	$\neg b$	-	-	-	-	-	-

Step	8	9	10	(1)	11	12	13	14
Decision Level	1	1	1	0	0	0	0	0
Assignment	$\neg a, b, \neg d$	$\neg a, b, c, \neg d$	$\neg a, b, c, \neg d, \neg e$	-	a	$a, \neg e$	$a, b, \neg e$	$a, b, \neg d, \neg e$
Cl. 1: $a, \neg c, \neg e$	$\neg c, \neg e$	$\neg e$	✓	1	✓	✓	✓	✓
Cl. 2: $\neg a, \neg e$	✓	✓	✓	2	$\neg e$	✓	✓	✓
Cl. 3: b, e	✓	✓	✓	3	3	b	✓	✓
Cl. 4: $\neg b, d, e$	e	e	$\{\}$ ✗	4	4	$\neg b, d$	d	$\{\}$ ✗
Cl. 5: $\neg b, \neg d$	✓	✓	✓	5	5	5	$\neg d$	✓
Cl. 6: $c, \neg d$	✓	✓	✓	6	6	6	6	✓
Cl. 7: c, d	c	✓	✓	7	7	7	7	c
Cl. 8: a, b	✓	✓	✓	8	✓	✓	✓	✓
Cl. 9: a			a	9	✓	✓	✓	✓
BCP	c	$\neg e$	-	a	$\neg e$	b	$\neg d$	-
PL	-	-	-	-	-	-	-	-
Decision	-	-	-	-	-	-	-	UNSAT

Figure 3.6: DPLL algorithm with decisions, BCP, PL and clause learning.

We encounter a conflict at step 6. The corresponding conflict graph given in Figure 3.7 reveals that both decisions $\neg a$ and $\neg b$ can be blamed for the conflict. Therefore, we learn $\{a, b\}$ as our 8th clause. The algorithm backtracks to decision level 1. Here the new clause is a unit clause and we are able to set b by BCP.

In step 10, the next conflict is reached. From the conflict graph given in Figure 3.8 we learn the 9th clause: $\{a\}$. The algorithm backtracks to decision level 0 and applies BCP.

We again encounter a conflict in step 14. Since the algorithm reached a conflict at decision level 0, the algorithm determines and returns as result that the formula is unsatisfiable, i.e., there does not exist a satisfying assignment. For completeness, Figure 3.9 shows the conflict graph for the last conflict, which is no longer needed to generate a new clause. Note, that the graph does not contain a decision node (a node without an incoming edge).

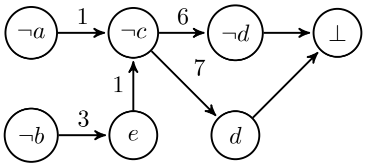

Figure 3.7: Conflict graph for step 6.

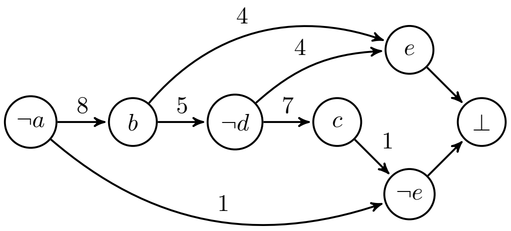

Figure 3.8: Conflict graph for step 10.

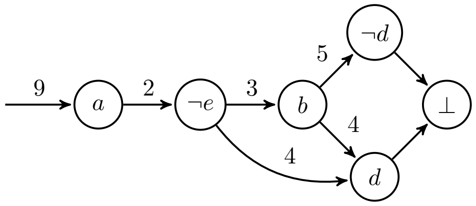

Figure 3.9: Conflict graph for step 14.

3.2.4 Resolution Proofs

We have seen that CDCL generates new clauses. We can proof that the new clauses are implied by the existing clauses using the resolution rule.

Definition - Resolution Rule. Let $c_1 = (\phi \vee a)$ and $c_2 = (\psi \vee \neg a)$ be two clauses, where ϕ and ψ denote disjunctions of arbitrary literals. Then the clause $\phi \vee \psi$ is implied by $c_1 \wedge c_2$.

The resolution rule is a derived natural deduction rule and can be written as follows:

$$\frac{(a \lor \phi) \qquad (\neg a \lor \psi)}{(\phi \lor \psi)}$$

Definition - Resolution Proof. A *resolution proof* is a natural deduction proof, that proofs the new clause from the existing clauses by applying the resolution rule only.

The resolution proof for a learned clause can be automatically generated traversing a conflict graph from the *conflict node* to the *root nodes* and applying the resolution rule on the clauses that are marked on the edges.

Example. Give the resolution proof for the learned clause $a \lor b$ from the conflict graph given in Figure 3.7.

To construct the resolution proof, we start by the conflict and apply the resolution rule on clause 6 containing the literal d and clause 7 containing the literal $\neg d$. The conclusion after applying the resolution rule is c. The formula c forms then the first premise for the next application of the resolution rule, and clause 1 forms the second premise. The result is $a \lor \neg e$. Finally, the formula $a \lor \neg e$ and clause 3 serve as premises for the last application of the resolution rule. The conclusion is the learned clause $a \lor b$ that we wanted to proof from a set of given clauses. Figure 3.10 gives the resolution proof in tree representation.

$$\dfrac{\dfrac{\dfrac{\textcircled{6}\ c \lor \neg d \qquad \textcircled{7}\ c \lor d}{c} \qquad \textcircled{1}\ a \lor \neg c \lor \neg e}{a \lor \neg e} \qquad \textcircled{3}\ b \lor e}{a \lor b}$$

Figure 3.10: Resolution proof for conflict graph in Figure 3.7.

Example. Give the resolution proof for the learned clause a from the conflict graph given in Figure 3.8.

The proof is given in Figure 3.11.

$$\dfrac{\dfrac{\dfrac{\dfrac{\textcircled{1}\ a \lor \neg c \lor \neg e \qquad \textcircled{4}\ \neg b \lor d \lor e}{a \lor \neg b \lor \neg c \lor d} \qquad \textcircled{7}\ c \lor d}{a \lor \neg b \lor d} \qquad \textcircled{5}\ \neg b \lor \neg d}{a \lor \neg b} \qquad \textcircled{8}\ a \lor b}{a}$$

Figure 3.11: Resolution proof for conflict graph in Figure 3.8.

Finally, the resolution proof can be used to automatically generate a proof that a formula is UNSAT from a given conflict graph at decision level 0.

Example. Show that the formla is UNSAT from the conflict graph given in Figure 3.9. The proof is given in Figure 3.12. Therefore, from a given set of clauses we proved that the formula is indeed UNSAT.

$$\dfrac{\dfrac{\dfrac{\dfrac{\textcircled{4}\ \neg b \lor d \lor e \qquad \textcircled{5}\ \neg b \lor \neg d}{\neg b \lor e} \qquad \textcircled{3}\ b \lor e}{e} \qquad \textcircled{2}\ \neg a \lor \neg e}{\neg a} \qquad \textcircled{9}\ a}{\bot}$$

Figure 3.12: Resolution proof for conflict graph in Figure 3.9.