Satisfiability Checking

Summary Winter Semester 2021

Merlin Denker

Disclaimer

This summary was written to repeat the lecture contents and to provide a resource in which I can easily lookup definitions when needed. It was published to allow other students to use it as learning material. Bear in my mind that this is not official material, it may not be complete and there might be mistakes in it. I am not affiliated with the chair i2 at RWTH Aachen University in any way.

This is based on the course ”Erfüllbarkeitsprüfung (Satisfiability checking)” held by Prof. Abraham at RWTH Aachen University in the winter semester 2021. Some of the content is taken directly from the lecture slides and I do not claim authorship of those parts. I tried to add a note whenever I added my own work or altered examples etc., but I might have missed some spots. Also, I did not add notes when I just changed the wording in minor ways.

Work in Progress: This document is currently work in progress and thus not finished yet. I will update it, when I continue work on it. I might however also cancel further work on it and instead work on a shorter document much like my summary of ”Betriebssysteme und Systemsoftware” that sums up only certain algorithms in informal and simple descriptions. This is not yet decided as of today.

Distribution

Distributing this document shall be free of charge. If you happen to find this on a paid website, please inform the author (merlin.denker@rwth-aachen.de). Yes, this has happened before with other documents I provided for learning purposes.

Contribution

If you find any mistakes, please report them to the author: merlin.denker@rwth-aachen.de

How to read this document

Definitions, Theorems, Corollaries, Lemmata and Examples will be enclosed in color coded boxes with enumerations. You can find a list of those in the Index at the end of the document.

Example Definition

This is what a definition will look like.

Example Theorem

This is what a theorem will look like.

Example Corollary

This is what a corollary will look like.

Example Lemma

This is what a lemma will look like.

Example Example

This is what a example will look like.

Also, when I added bigger blocks of information that were not part of the lecture, I enclosed them in an information box:

Example Info

This is what an info will look like.

Contents

1 Propositional Logic

1.1 Syntax of propositional logic

Definition 1 (Syntax of propositional logic)

AP is a set of (atomic) **propositions** (Boolean variables).

Let $a \in \mathsf{AP}$. The **abstract syntax** of well-formed propositional formulae is defined by:

$$\varphi := a \mid (\neg\varphi) \mid (\varphi \wedge \varphi)$$

We write PropForm for the set of all propositional logic formulae.

Syntactic sugar:

$$\begin{aligned}
\bot &:= (a \wedge \neg a) \\
\top &:= (a \vee \neg a) \\
(\varphi_1 \vee \varphi_2) &:= \neg((\neg\varphi_1) \wedge (\neg\varphi_2)) \\
(\varphi_1 \rightarrow \varphi_2) &:= ((\neg\varphi_1) \vee \varphi_2) \\
(\varphi_1 \leftrightarrow \varphi_2) &:= ((\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)) \\
(\varphi_1 \oplus \varphi_2) &:= (\varphi_1 \leftrightarrow (\neg\varphi_2))
\end{aligned}$$

We omit parantheses whenever we may restore them through operator precedence:
$\neg > \wedge > \vee > \rightarrow > \leftrightarrow$
where $a > b$ means a binds stronger than b.

We will also use the big Boolean notation, e.g.

$$\bigwedge_{i=1}^{5} x_i := x_1 \wedge x_2 \wedge x_3 \wedge x_4 \wedge x_5$$

1.2 Semantics of propositional logic

Definition 2 (Assignments)

An **interpretation** α maps variables to Boolean values:

$$\alpha : \mathsf{AP} \to \mathbb{B}$$

where $\mathbb{B} = \{0, 1\}$.
We call these interpretations **assignments** and use Assign to denote the set of all assignments.
Equivalently, we can see an assignment α as a set of variables ($\alpha \subseteq \mathsf{AP}$), defining the variables from the set to be true and the others false.
An assignment can also be seen as being of type $\alpha \in \mathbb{B}^{|\mathsf{AP}|}$, if we have an order on the propositions.
Info: This was defined slightly different in the lecture.

Example 1 (Assignments)

Let $\mathsf{AP} = \{a, b\}$. The following three statements are equivalent:

1. $\alpha(a) = 0, \alpha(b) = 1$
2. $\alpha = \{b\}$
3. $\alpha = 01$

Definition 3 (Projection)

Let $\alpha \in \mathsf{Assign}$, $A \subseteq \mathsf{AP}$.
A Projection is defined as "the assignments in α to the variables of A".

$$\alpha \mid_A := \{(a, \alpha(a)) \mid a \in A\}$$

The projection is basically a subset of an assignment.
Note: This was not defined in the lecture

Lemma 1 (Equivalence of projected assignments)

Let $\alpha_1, \alpha_2 \in \mathsf{Assign}$ and $\varphi \in \mathsf{PropForm}$.
Let $\mathsf{AP}(\varphi) \subseteq \mathsf{AP}$ be the atomic propositions in φ.
Then

$$\text{if } \alpha_1 \mid_{\mathsf{AP}(\varphi)} = \alpha_2 \mid_{\mathsf{AP}(\varphi)}\text{, then}$$
$$(\alpha_1 \text{ satisfies } \varphi) \text{ iff } (\alpha_2 \text{ satisfies } \varphi)$$

1.2.1 Truth tables

Truth tables define the semantics of the operators. They can be used to define the semantics of formulae inductively over their structure.

Convention: 0 = false, 1 = true

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \to q$	$p \leftrightarrow q$	$p \oplus q$
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	0	0	1
1	1	0	1	1	1	1	0

Each possible assignment is covered by a line of the truth table.
α **satisfies** φ iff in the line for α and the column for φ the entry is 1.

A total of 16 binary operators can be defined that have different meanings.

Info 1 (List of binary operators)

We can enumerate all 16 possible binary operators by assigning numbers to them according to their output behaviour. Take the truth table above as a reference: When reading the results of an operators from top to bottom, they can be interpreted as a binary number. The operator $\oplus$ for example is assigned the number $0110_2 = 7_{10}$.
We fix the order of rows to that of the table above and get the following list of operators:

0. 0000 False (Contradiction)
1. 0001 $\wedge$ (AND, Conjunction)
2. 0010 $\nrightarrow$ (Material Nonimplication, Abjunction, Subtraction without carry)
3. 0011 p
4. 0100 $\nleftarrow$ (Converse Nonimplication)
5. 0101 q
6. 0110 $\oplus$ (XOR, Exclusive Disjunction, Addition without carry)
7. 0111 $\vee$ (OR, Disjunction)
8. 1000 $\downarrow$ (NOR)
9. 1001 $\leftrightarrow$ (XNOR, Equivalency, Material Biconditional)
10. 1010 $\neg q$
11. 1011 $\leftarrow$ (Converse)
12. 1100 $\neg p$
13. 1101 $\rightarrow$ (Implication, Material Conditional)
14. 1110 $\uparrow$ (NAND)
15. 1111 True (Tautology)

Info: This was not part of the lecture

1.2.2 Satisfying assignments

Definition 4 (sat function)

Idea: Each formula specifies a **set of assignments** satisfying it.

We define $\mathsf{sat} : \mathsf{PropForm} \to \mathsf{Assign}^n$ recursively:

$$\begin{aligned}
\mathsf{sat}(a) &= \{\alpha \mid \alpha(a) = 1\}, a \in \mathsf{AP} \\
\mathsf{sat}(\neg\varphi_1) &= \mathsf{Assign}\backslash\mathsf{sat}(\varphi_1) \\
\mathsf{sat}(\varphi_1 \wedge \varphi_2) &= \mathsf{sat}(\varphi_1) \cap \mathsf{sat}(\varphi_2) \\
\mathsf{sat}(\varphi_1 \vee \varphi_2) &= \mathsf{sat}(\varphi_1) \cup \mathsf{sat}(\varphi_2) \\
\mathsf{sat}(\varphi_1 \to \varphi_2) &= (\mathsf{Assign}\backslash\mathsf{sat}(\varphi_1)) \cup \mathsf{sat}(\varphi_2)
\end{aligned}$$

Definition 5 (Satisfaction relation)

1. Let a formula $\varphi \in \mathsf{PropForm}$ and an assignment $\alpha \in \mathsf{Assign}$.
   We define $\models \subseteq \mathsf{Assign} \times \mathsf{PropForm}$ by

   $$\alpha \models \varphi \text{ iff } \alpha \in \mathsf{sat}(\varphi)$$

2. Let a formula $\varphi \in \mathsf{PropForm}$ and an assignment set $T \subseteq \mathsf{Assign}^n$.
   We define $\models \subseteq \mathsf{Assign}^n \times \mathsf{PropForm}$ by

   $$T \models \varphi \text{ iff } T \subseteq \mathsf{sat}(\varphi)$$

3. Let two formulae $\varphi_1, \varphi_2 \in \mathsf{PropForm}$.
   We define $\models \subseteq \mathsf{PropForm} \times \mathsf{PropForm}$ by

   $$\varphi_1 \models \varphi_2 \text{ iff } \mathsf{sat}(\varphi_1) \subseteq \mathsf{sat}(\varphi_2)$$

1.3 Satisfiability and validity

1.3.1 Semantic classification of formulae

Corollary 1 (Semantic classification of formulae)

- A formula φ is called **valid** if $\mathsf{sat}(\varphi) = \mathsf{Assign}$. (Also called a **tautology**).
- A formula φ is called **satisfiable** if $\mathsf{sat}(\varphi) \neq \emptyset$.
- A formula φ is called **unsatisfiable** if $\mathsf{sat}(\varphi) = \emptyset$. (Also called a **contradiction**).

It holds that the set of all valid formulae is a subset of all satisfiable formulae.

We can write:

- $\models \varphi$ when φ is **valid**
- $\not\models \varphi$ when φ is **not valid**
- $\not\models \neg\varphi$ when φ is **satisfiable**
- $\models \neg\varphi$ when φ is **unsatisfiable**

1.3.2 Satisfiability problem for propositional logic

Definition 6 (Satisfiability problem for propositional logic)

Given an input propositional formula φ, decide whether φ is satisfiable.

This problem is decidable, but **NP-complete**.

1.4 Normal Forms

Definition 7 (Literal)

A **literal** is either a variable or the negation of a variable.

Let formulae $\varphi_1, \varphi_2 \in \mathsf{PropForm}$, a variable $a \in \mathsf{AP}$ and $\circ : (\mathbb{B} \times \mathbb{B}) \to \mathbb{B}$ a binary operator.

We define $\mathsf{lit} : \mathsf{PropForm} \times (\mathsf{AP} \cup \{\neg x \mid x \in \mathsf{AP}\})$ recursively:

$$\begin{aligned} \mathsf{lit}(a) &:= \{a\} \\ \mathsf{lit}(\neg a) &:= \{\neg a\} \\ \mathsf{lit}(\varphi_1 \circ \varphi_2) &:= \mathsf{lit}(\varphi_1) \cup \mathsf{lit}(\varphi_2) \end{aligned}$$

Info: This definition was only done informally in the lecture.

Note: Equivalent formulae can have different literals.

Definition 8 (Terms and Clauses)

A **term** is a conjunction of literals.

A **clause** is a disjunction of literals.

1.4.1 Negation Normal Form (NNF)

Definition 9 (Negation Normal Form (NNF))

A formula is in **Negation Normal Form (NNF)** iff

1. It contains only $\neg$, $\wedge$ and $\vee$ as connectives and
2. Only variables are negated

Example 2 (Negation Normal Form (NNF))

- $a \rightarrow b$ is **not in** NNF (due to the use of $\rightarrow$)
- $\neg(a \vee \neg b)$ is **not in** NNF (because a clause is negated)
- $\neg a \wedge b$ is **in** NNF

Every formula can be converted to NNF in linear time:

- Eliminate all connectives other than $\wedge$, $\vee$, $\neg$ and
- Use De Morgan and double-negation rules to push negations to operands.

Example 3 (Conversion to NNF)

Let $\varphi = \neg(a \rightarrow \neg b)$.

- eliminate '$\rightarrow$': $\varphi = \neg(\neg a \vee \neg b)$
- push negation using De Morgan: $\varphi = (\neg\neg a \wedge \neg\neg b)$
- use double-negation rule: $\varphi = (a \wedge b)$

Transformation to NNF can be done with an effor (time and space) that is linear in the size of the formular, if the formula does not contain nested if-and-only-if operators.

1.4.2 Disjunctive Normal Form (DNF)

Definition 10 (Disjunctive Normal Form (DNF))

A formula is in **Disjunctive Normal Form (DNF)** iff it is a disjunction of terms.
In other words, it is a formula of the form

$$\bigvee_i \left(\bigwedge_j l_{i,j} \right)$$

where $l_{i,j}$ is the j-th liteal in the i-th term.

Example 4 (Disjunctive Normal Form (DNF))

- $\varphi = (a \wedge \neg b \wedge c) \vee (\neg a \wedge d) \vee (b)$ is in DNF

DNF is a special case of NNF.

Every formula can be converted to DNF in **exponential** time and space:

1. Convert to NNF
2. Distribute conjunctions following the rule:
   $\models \varphi \wedge (\varphi_2 \vee \varphi_3) \leftrightarrow (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$

Example 5 (Conversion to DNF)

$$\begin{aligned}
\varphi &= (a \vee b) \wedge (\neg c \vee d) \\
&= ((a \vee b) \wedge (\neg c)) \vee ((a \vee b) \wedge d) \\
&= (a \wedge \neg c) \vee (b \wedge \neg c) \vee (a \wedge d) \vee (b \wedge d)
\end{aligned}$$

The complexity of the satisfiability check of a DNF formula is linear in time and space. A polynomial transformation into DNF is not possible, as it would violate the NP-completeness of the problem.

1.4.3 Conjunctive Normal Form (CNF)

Definition 11 (Conjunctive Normal Form (CNF))

A formula is in **Conjunctive Normal Form (DNF)** iff it is a conjunction of clauses.
In other words, it is a formula of the form

$$\bigwedge_j \left(\bigvee_i l_{i,j} \right)$$

where $l_{i,j}$ is the j-th liteal in the i-th clause.

Example 6 (Conjunctive Normal Form (CNF))

- $\varphi = (a \vee \neg b \vee c) \wedge (\neg a \vee d) \wedge (b)$ is in CNF

CNF is a special case of NNF.

Every formula can be converted to CNF in **exponential** time and space:

1. Convert to NNF
2. Distribute disjunctions following the rule:
   $\models \varphi \vee (\varphi_2 \wedge \varphi_3) \leftrightarrow (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$

1.4.4 Tseitin's Encoding

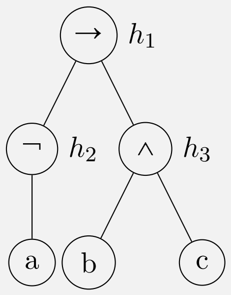
It converts formulas to CNF in **linear** time and space by introducing new variables. The original and the converted formula are **not equivalent** but **equi-satisfiable**. It works as follows:

1. Convert the formula into a parse tree
2. Associate a new auxiliary variable with each inner (non-leaf) node
3. Add constraints that define these new variables
4. Finally, enforce the truth of the root node

For a given formula $\varphi_n = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n)$ in DNF, the encoding will introduce n auxiliary variables $h_1, \cdots, h_n$, which add 3 clauses each. The top clause will be $(h_1 \vee \cdots \vee h_n)$. Hence, we can convert a DNF formula into an equi-satisfiable CNF formula, which has $3n + 1$ clauses instead of 2^n and uses $3n$ variables rather than $2n$.

Example 7 (Tseitin's Encoding)

Consider the formula $\varphi_1 = \neg a \rightarrow (b \wedge c)$.
First we draw the parse tree and associate new variables h_1, h_2 and h_3 to the inner nodes. For each variable we require it to be logically equivalent to the node it is associated with.



$$h_1 \leftrightarrow (h_2 \rightarrow h_3)$$
$$h_2 \leftrightarrow (\neg a)$$
$$h_3 \leftrightarrow (b \wedge c)$$

We create a conjunction of these constraints and enforce the truth of the root node by adding (h_1) to the conjunction. The result is the new formula φ_2 which is the Tseitin's encoding of φ_1:

$$\varphi_2 = (h_3 \leftrightarrow (b \wedge c)) \wedge (h_2 \leftrightarrow (\neg a)) \wedge (h_1 \leftrightarrow (h_2 \rightarrow h_3)) \wedge (h_1)$$

Each node's encoding has a CNF representation with 2 to 4 clauses:

$h_3 \leftrightarrow (b \wedge c)$	in CNF: $(\neg h_3 \vee b) \wedge (\neg h_2 \vee c) \wedge (h_2 \vee \neg b \vee \neg c)$
$h_2 \leftrightarrow (\neg a)$	in CNF: $(\neg h_2 \vee \neg a) \wedge (a \vee h_2)$
$h_1 \leftrightarrow (h_2 \rightarrow h_3)$	in CNF: $(h_1 \vee h_2) \wedge (h_1 \vee \neg h_3) \wedge (\neg h_1 \vee \neg h_2 \vee h_3)$

Replacing the constraint formulae in the Tseiting's encoding φ_2 with their respective CNF forms yields the desired CNF formula which is equi-satisfiable to φ_1.
Note: The example formula in the lecture was $\varphi = a \rightarrow (b \wedge c)$ (without a negation).

1.5 Deductive proof systems

Definition 12 (Deductive proof system)

A (deductive) **proof system** consists of a set of axioms and inference rules.

Inference rules are denoted as follow:

$$\frac{\text{Antecedents}}{\text{Consequents}}\ \text{(rule name)}$$

Meaning: If all antecedents hold then at least one of the consequents can be derived.

Axioms are inference rules with no antecedents.

Example 8 (Axioms and inference rules)

- (H1) is an axiom:

$$\frac{}{a \rightarrow (b \rightarrow a)}\ \text{(H1)}$$

- (Trans) and (M.P.) are inference rules:

$$\frac{a \rightarrow b \qquad b \rightarrow c}{a \rightarrow c}\ \text{(Trans)} \qquad\qquad \frac{a \rightarrow b \qquad a}{b}\ \text{(M.P.)}$$

1.5.1 Provability

Definition 13 (Provability Relation)

Let $\mathcal{H}$ be a proof system, $\Gamma \in \mathsf{PropForm}^n$ a set of formulae and $\varphi \in \mathsf{PropForm}$ a formula.

$\gamma \vdash_{\mathcal{H}} \varphi$ means: There is a proof of φ in system $\mathcal{H}$ whose premises are included in Γ.

$\vdash_{\mathcal{H}} \in (\mathsf{PropForm}^n \times \mathsf{PropForm})$ is called the **provability (derivability) relation**.

Note: This was defined more informal in the lecture.

Example 9 (Deductive proof)

Let $\Gamma = a \rightarrow b, b \rightarrow c, c \rightarrow d, d \rightarrow e, a$. We show that e can be derived using the inference rules (Trans) and (M.P.) as defined in the previous example:

$$\dfrac{\dfrac{\dfrac{a \rightarrow b \qquad b \rightarrow c}{a \rightarrow c}\ \text{(Trans)} \qquad \dfrac{c \rightarrow d \qquad d \rightarrow e}{c \rightarrow e}\ \text{(Trans)}}{a \rightarrow e}\ \text{(Trans)} \qquad a}{e}\ \text{(M.P)}$$

1.5.2 Soundness and completeness

Definition 14 (Soundness and completeness)

For a given proof system $\mathcal{H}$:

- **Soundness**: Does $\vdash$ conclude "correct" conclusions from premises?
- **Completeness:** Can we conclude all true statements with $\mathcal{H}$?

Correct here means correct with respect to the semantic definition of the logic. In the case of propositional logic truth tables give us this.

$$\begin{aligned}\text{Soundness of } \mathcal{H}\text{:} &\quad \text{if } \Gamma \vdash_{\mathcal{H}} \varphi \text{ then } \Gamma \models \varphi \\ \text{Completeness of } \mathcal{H}\text{:} &\quad \text{if } \Gamma \models \varphi \text{ then } \Gamma \vdash_{\mathcal{H}} \varphi\end{aligned}$$

Definition 15 (Hilbert axiom system (H))

Let H be (M.P.) together with the following axiom schemes:

$$\frac{}{a \to (b \to a)}\ (\mathrm{H1})$$

$$\frac{}{((a \to (b \to c)) \to ((a \to b) \to (a \to c)))}\ (\mathrm{H2})$$

$$\frac{}{(\neg b \to \neg a) \to (a \to b)}\ (\mathrm{H3})$$

Corollary 2 (Soundness and completeness of H)

H is **sound and complete** for propositional logic.

1.5.3 Resolution proof system

Definition 16 (Resolution proof system)

The resolution proof system contains only one inference rule for CNF formulae:

$$\frac{(l \lor l_1 \lor \dots \lor l_n) \qquad (\neg l \lor l'_1 \lor \dots \lor l'_m)}{(l_1 \lor \dots \lor l_n \lor l'_1 \lor \dots \lor l'_m)} \text{(Resolution)}$$

Example 10 (Resolution proof system)

Let $\varphi = (a_1 \lor a_3) \land (\neg a_1 \lor a_2 \lor a_5) \land (\neg a_1 \lor a_4) \land (\neg a_1 \lor \neg a_4)$.

We want to prove $\varphi \to (a_3)$:

$$\frac{(a_1 \lor a_3) \qquad \dfrac{(\neg a_1 \lor a_4) \qquad (\neg a_1 \lor \neg a_4)}{(\neg a_1)}}{(a_3)}$$

Resolution is a sound and complete proof system for CNF.

If the input formula is unsatisfiable, there exists **a proof of the empty clause**.

Example 11 (Resolution of the empty clause)

Let $\varphi = (a_1 \lor a_3) \land (\neg a_1 \lor a_2) \land (\neg a_1 \lor a_4) \land (\neg a_1 \lor \neg a_4) \land (\neg a_3)$.

$$\frac{\dfrac{\dfrac{(\neg a_1 \lor a_4) \qquad (\neg a_1 \lor \neg a_4)}{(\neg a_1)} \qquad (a_1 \lor a_3)}{a_3} \qquad \neg a_3}{()}$$

2 SAT solving

2.1 Enumeration

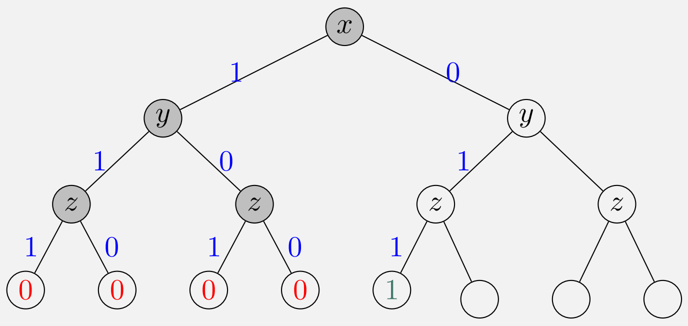
Definition 17 (Enumeration algorithm)

```
bool Enumeration(CNF_Formula $\varphi$) {
    trail.clear(); // stack of entries $(x,v,b)$ assigning value $v$
                   // to proposition $x$ and a flag $b$ stating whether
                   // $\neg v$ has already been processed for $x$
    while (true) {
        if (!decide()) {
            if (all clauses of $\varphi$ are satisfied
                by the assignment in trail) then return SAT;
            else if (!backtrack()) then return UNSAT
        }
    }
}
bool decide() {
    if (all variables are assigned) then return false;
    choose unassigned variable $x$ not yet in trail;
    choose value $v \in \{0,1\}$;
    trail.push($x$, $v$, false);
    return true
}
bool backtrack() {
    while (true) {
        if (trail.empty()) then return false;
        $(x,v,b)$ := trail.pop()
        if(!b) then {
            trail.push(($x$,$\neg v$, true));
            return true
        }
    }
}
```

Example 12 (Enumeration algorithm with static heuristic)

Let $\varphi = (\neg x \vee y \vee z) \wedge (y \vee \neg z) \wedge (\neg x \vee \neg y)$.

Static variable order $x < y < z$, sign: try positive first



Gray filled nodes have the b flag set. Leaf nodes resemble an evaluation of φ under the given assignment.

For unsatisfiable problems, all assignments need to be checked.

For satisfiable problems, variable and sign ordering might stringly influence running time.

In this case, trying negative first would yield a positive result in the first try.

2.1.1 Dynamic decision heuristics example (DLIS)

Definition 18 (Dynamic decision heuristics example (DLIS))

Choose an assignment that increases the number of satisfied clauses the most.

- For each literal l, let C_l be the number of unresolved clauses in which l appears.
- Let l be a literal for which C_l is maximal ($C_{l'} \leqslant C_l$ for all literals l').
- If l is a variable x then assign true to x.
- Otherwise if l is a negated variable $\neg x$ then assign false to x.

Requires $\mathcal{O}(\#\text{literals})$ queries for each decision

Example 13 (Dynamic decision heuristics example (DLIS))

Consider the formula $\varphi = (\neg x \vee y \vee z) \wedge (y \vee \neg z) \wedge (\neg x \vee \neg y)$ from the previous example.

Fallback literal order (in case of equal values): $\neg x < x < \neg z < z < \neg y < y$

Step	C_x	C_y	C_z	$C_{\neg x}$	$C_{\neg y}$	$C_{\neg z}$	Assignments
1	0	2	1	2	1	1	$x = 0$
2	0	1	0	0	0	1	$x = 0,\ z = 0$
3	0	0	0	0	0	0	$x = 0,\ z = 0,\ y = 0$

This assignment already satisfies the formula φ.

2.1.2 Jeroslow-Wang method

Definition 19 (Jeroslow-Wang method)

Compute the following **static** value for every literal l:

$$J(l) : \sum_{\text{clause } c \text{ in the CNF containing } l} 2^{-|c|}$$

where $|c|$ denotes the number of literals in a clause.

- Choose a literal l that maximizes $J(l)$.
- This gives an exponentially higher weight to literals in shorter clauses.

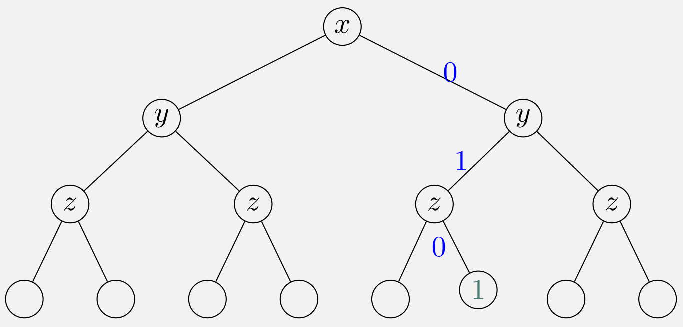
Example 14 (Jeroslow-Wang method)

Consider the formula $\varphi = (\neg x \vee y \vee z) \wedge (y \vee \neg z) \wedge (\neg x \vee \neg y)$ from the previous example.
Fallback literal order (in case of equal values): $\neg x < x < \neg z < z < \neg y < y$

$$J(x) = 0 \qquad J(y) = \frac{1}{8} + \frac{1}{4} \qquad J(z) = \frac{1}{8}$$

$$J(\neg x) = \frac{1}{8} + \frac{1}{4} \qquad J(\neg y) = \frac{1}{4} \qquad J(\neg z) = \frac{1}{4}$$

We decide $x = 0$, $y = 1$, $z = 0$ and instantly find a satisfying assignment:



2.2 Boolean constraint propagation (BCP)

We first introduce some terms and notation:

Definition 20 (Status of a clause)

Given a (partial) assignment, a clause can be

- **satisfied**: at least one literal is satisfied
- **unsatisfied**: all literals are assigned but none are satisfied
- **unit**: all but one literals are assigned but none are satisfied
- **unsresolved**: all other cases

Example 15 (Status of a clause)

Let $c = (x_1 \lor x_2 \lor x_3)$

x_1	x_2	x_3	Status of c
1	0		satisfied
0	0	0	unsatisfied
0	0		unit
	0		unresolved

In BCP unit clauses are used to imply consequences of decisions.

Definition 21 (Decision Level)

Decision Level (DL) is a counter for decisions.

Definition 22 (Antecedent)

Antecedent(l): unit clause implying the value of the literal l.

It is nil if the value was assigned via a decision.

2.2.1 DPLL algorithm

Definition 23 (DPLL algorithm)

The **Davis–Putnam–Logemann–Loveland (DPLL) algorithm** is defined as follows:

```
bool DPLL(CNF_Formula $\varphi$) {
    trail.clear(); // trail is a global stack of assignments
    if (!BCP()) then return UNSAT;
    while (true) {
        if (!decide()) then return SAT;
        while (!BCP())
            if(!backtrack()) then return UNSAT;
    }
}
bool decide() { same as in enumeration algorithm }
bool backtrack() { same as in enumeration algorithm }
bool BCP() {
    while (there is a unit clause implying that a variable $x$
            must be set to a value $v$)
        trail.push($x$,$v$,true);
    if (there is an unsatisfied clause) then return false;
    return true;
}
```

Example 16 (DPLL algorithm)

Consider the formula $\varphi = \underbrace{(\neg x \vee y \vee z)}_{c_1} \wedge \underbrace{(y \vee \neg z)}_{c_2} \wedge \underbrace{(\neg x \vee \neg y)}_{c_3}$.

Static variable order: $x < y < z$, sign: try positive first

1. **DL 0**: No implications are made and the formula is neither unsatisfied nor satisfied
2. **DL 1**:
   a) Decide $x = 1$
   b) c_3 implies $y = 0$
   c) c_1 implies $z = 1$
   d) c_2 is now unsatisfied
   e) backtrack to the last decision: unset z and y, set $x = 0$ and set the b flag for x
   f) No implications are made and the formula is neither unsatisfied nor satisfied
3. **DL 2**:
   a) Decide $y = 1$
   b) No implications are made and the formula is neither unsatisfied nor satisfied
4. **DL 3**:
   a) Decide $z = 1$
   b) The formula is now satisfied, return SAT

2.2.2 Watched Literals

For BCP, it would be a large effort to check the value of each literal in each clause for each propagation. The idea of watched literals is to only **watch two different literals** in each clause such that either one of them is true or both are unassigned. This way the clause is neither unit nor unsatisfied.

Definition 24 (Watched Literals)

Select two watched literals for each clause.

If a literal l gets true, we check each clause in which $\neg l$ is watched (which is now false):

- If the other watched literal of the clause is true, the clause is satisfied.
- Else, if we find a new literal to watch: Change the watched literal and we are done.
- Else, if the other watched literal is unassigned, the clause is now unit.
- Else, if the other watched literal is false, the clause is conflicting.

2.3 Conflict resolution and backtracking

2.3.1 Implication graph

We represent (partial) variable assignments in the form of an **implication graph**.

Definition 25 (Implication graph)

An **implication graph** is a labeled directed acyclic graph $G = (V, E, L)$, where

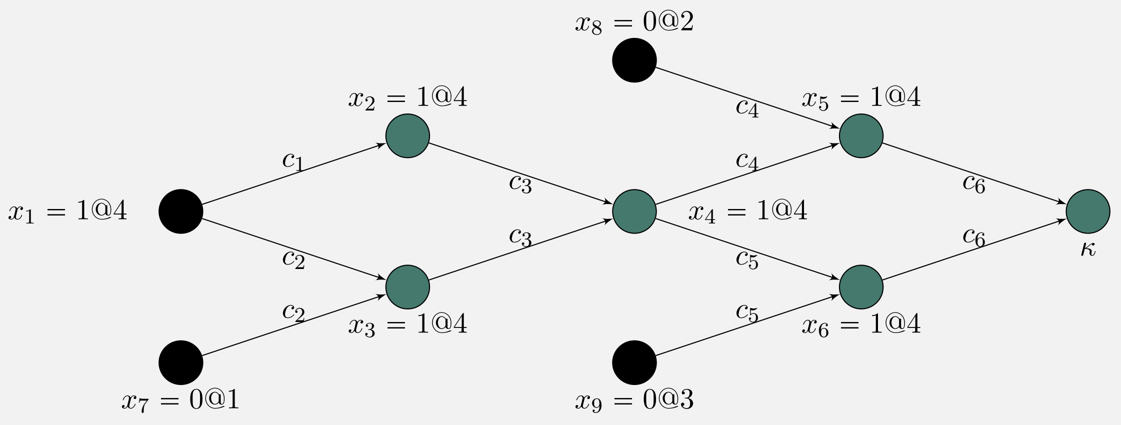
- V is a set of nodes, one for each currently assigned variable and an additional conflict node κ if there is a currently conflicting clause c_{confl}.
- L is a labeling function assigning a lable to each node.
  The conflict node (if any) is labelled by $L(\kappa) = \kappa$.
  Each other node n, representing that x is assigned $v \in \mathbb{B}$ at decision level d, is labeled with $L(n) = (x = c@d)$.
  We define $\mathsf{literal}(n) = x$ if $v = 1$ and $\mathsf{literal}(n) = \neg x$ if $v = 0$.
- $E = \{(n_i, n_j) \mid n_i, n_j \in V, n_i \neq n_j, \neg\mathsf{literal}(n_i) \in \mathsf{Antecedent}(\mathsf{literal}(n_j))\}$
  $\quad \cup \{(n, \kappa) \mid n, \kappa \in V, \neg\mathsf{literal}(n) \in c_{\mathrm{confl}}\}$
  is the set of directed edges where each edge (n_i, n_j) is labeled with $\mathsf{Antecedent}(\mathsf{literal}(n_j))$ if $n_j \neq \kappa$ and with c_{confl} otherwise.

Example 17 (Implication graph)

$c_1 = (\neg x_1 \vee x_2)$ $\quad c_2 = (\neg x_1 \vee x_3)$ $\quad c_3 = (\neg x_2 \vee \neg x_3 \vee x_4)$

$c_4 = (\neg x_4 \vee x_5 \vee x_8)$ $\quad c_5 = (\neg x_4 \vee x_6 \vee x_9)$ $\quad c_6 = (\neg x_5 \vee \neg x_6)$

Decisions: $\{x_7 = 0@1, x_8 = 0@2, x_9 = 0@3, x_1 = 1@4\}$



2.3.2 Conflict resolution

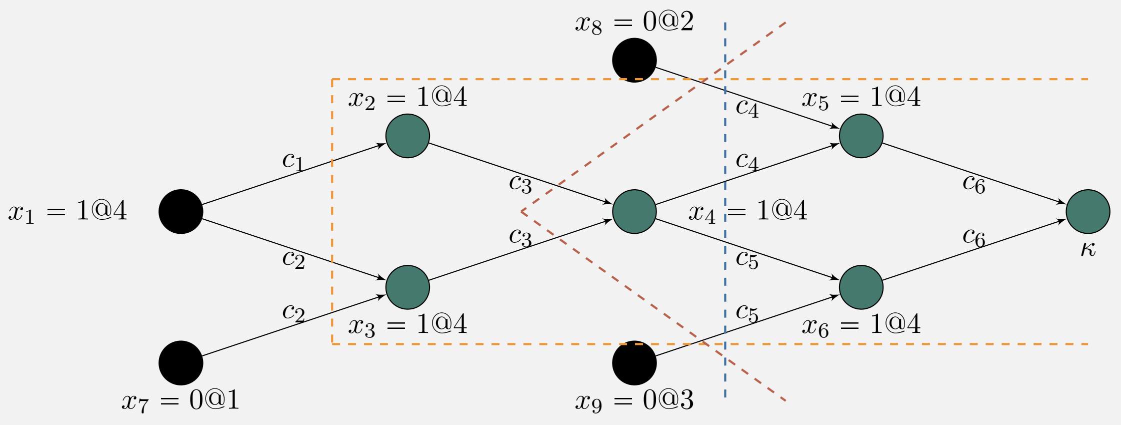
Definition 26 (Conflict clause)

Let $\varphi \in \mathsf{PropForm}$ in CNF and $\alpha \in \mathsf{Assign}$ a (partial) assignment such that $\alpha \not\models \varphi$ (φ is not satisfied).
Let L be a set of literals labeling nodes of the implication graph that form a cut in the implication graph, seperating a conflict node from the roots.
$\bigvee_{l \in L} \neg l$ is called a **conflict clause** if it is false under the current assignment α, but its satisfaction is necessary for the satisfaction of the formula φ.

The term **conflict clause** can easily be confused with the term **conflicting clause**. The latter is a special conflict clause, namely the one clause at which the conflict first arose. In the implication graph this is the clause at the edges which point at the conflict node κ. In the following example c_6 is the conflicting clause.

Example 18 (Cuts and conflict clauses)

We add 3 seperate cuts (marked by the colored dashed lines) into the implication graph from our previous example:



The conflict clauses corresponding to the cuts are as follows:

1. $(x_8 \vee \neg x_1 \vee x_7 \vee x_9)$
2. $(x_8 \vee \neg x_4 \vee x_9)$
3. $(x_8 \vee \neg x_2 \vee \neg x_3 \vee x_9)$

Which conflict clauses should we consider?

Definition 27 (Asserting clause)

An **asserting clause** is a conflict with a single literal from the current decision level.
Backtracking (to the right level) makes it a unit clause.

Definition 28 (Unique implication point (UIP))

Let an implication graph G with a conflict node κ.

A **unique implication point (UIP)** for κ in G is a node $n \neq \kappa$ in G such that **all paths from the last decision to κ go through n.**

The **first UIP** is the UIP closest to the conflict node.

There is always at least one UIP per decision level: The decision node. But there can also be multiple UIPs in one decision level, for example the node x_4 in the previous example is a UIP.

2.3.3 DPLL+CDCL algorithm

Definition 29 (DPLL+CDCL algorithm)

The **DPLL Conflict-driven clause learning (DPLL+CDCL) algorithm** is defined as follows:

```
if(!BCP()) return UNSAT;
while (true) {
    if (!decide()) return SAT;
    while(!BCP())
        if(!resolve_conflict()) return UNSAT;
}
```

When conflicts arise, this algorithm adds new clauses to the set of clauses, to avoid these conflicts in the future.

For conflict resolution as defined in the lecture, the (Binary Resolution) inference rule is used:

Definition 30 (Binary resolution)

The (Binary Resolution) inference rule is defined as follows:

$$\frac{(\beta \lor a_1 \lor \cdots \lor a_n) \qquad (\neg\beta \lor b_1 \lor \cdots \lor b_m)}{(_1 \lor \cdots \lor a_n \lor b_1 \lor \cdots \lor b_m)} \text{(Binary Resolution)}$$

Example 19 (Binary resolution)

$$\frac{(x_1 \lor x_2) \qquad (\neg x_1 \lor x_3 \lor x_4)}{(x_2 \lor x_3 \lor x_4)}$$

Definition 31 (Conflict resolution with binary resolution)

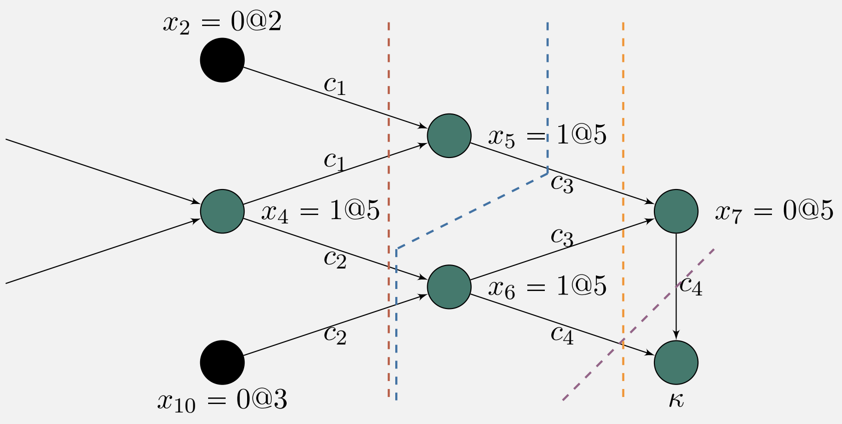
```
bool resolve_conflict() {
    if (current_decision_level = 0) then return false;
    cl := current_conflicting_clause;
    while (cl is not asserting) do {
        lit := last_assigned_literal(cl);
        var := variable_of_literal(lit);
        ante := antecedent(var);
        cl := binary_resolve(cl, ante, var);
    }
    add_clause_to_database(dl);
    return true;
}
```

Note: The lecture defined this as bool analyze_conflict(). It was renamed to fit the previous definition of DPLL+CDCL.

Example 20 (Conflict resolution with binary resolution)

Assignment order: x_4, x_5, x_6, x_7

$$c_1 = (\neg x_4 \vee x_2 \vee x_5) \qquad c_2 = (\neg x_4 \vee x_{10} \vee x_6)$$
$$c_3 = (\neg x_5 \vee \neg x_6 \vee \neg x_7) \quad c_4 = (\neg x_6 \vee x_7)$$



Starting with the current conflicting clause c_4, apply resolution with the antecedent of the last assigned literal, until we get an asserting clause:

1. $\text{T1} = \text{Res}(c_4, c_3, x_7) = (\neg x_5 \vee \neg x_6)$
2. $\text{T2} = \text{Res}(\text{T1}, c_2, x_6) = (\neg x_4 \vee \neg x_5 \vee x_{10})$
3. $\text{T3} = \text{Res}(\text{T2}, c_1, x_5) = (x_2 \vee \neg x_4 \vee x_{10}) =: c_5 \rightarrow$ Asserting clause reached

2.3.4 Unsatisfiable core & Resolution graph

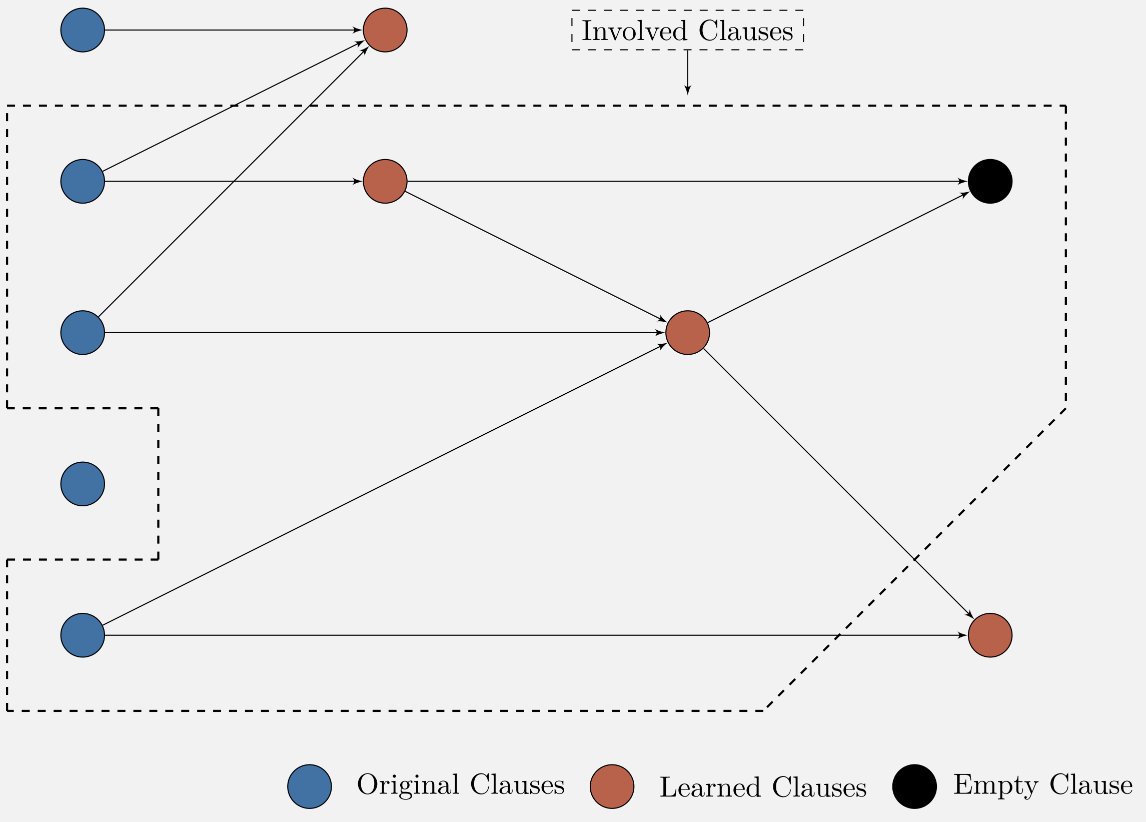
Definition 32 (Unsatisfiable core)

An **unsatisfiable core** of an unsatisfiable CNF formula is an unsatisfiable subset of the original set of clauses.

- The set of all original clauses is an unsatisfiable core
- The set of those original clauses that were used for resolution in conflict analysis during SAT-solving (inclusively the last conflict at decision level 0) gives us an unsatisfiable core which is in general much smaller.
- However, this unsatisfiable core is still not always minimal (i.e., we can remove clauses from it still having an unsatisfiable core).

A **resolution graph** gives us more information to get a minimal unsatisfiable core.

Example 21 (Resolution graph)



2.3.5 Termination

Theorem 1 (Termination of BCP and Conflict Resolution)

It is never the case that the solver enters decision level dl again with the same partial assignment.

Proof:
Define a partial order on partial assignments: $\alpha < \beta$ iff either a is an extension of β or α has more assignments at the smallest decision level at that α and β do not agree.
BCP decreases the order, conflict-driven backtracking also. Since the order always decreases during the search, the theorem holds.

2.4 Enumeration (decision) revisited

Index