CS257: Introduction to Automated Reasoning

Normal Forms, DP

Stanford University

CENTAUR

Agenda

- NNF, DNF, CNF (CC Ch. 1.6)
- Tseitin Transformation (MI Ch. 1.6)
- SAT-solving overview
- DP (CC Ch. 1.7)

Next lecture: DPLL and CDCL.

We will focus on one important application of SAT, model checking, on 10/11.

You will write a SAT-based Sudoku solver in the homework.

* Some of the slides today are contributed by Clark Barrett, Emina Torlak, and Cesare Tinelli.

Normal forms

A **normal form** of formulas is a syntactic restriction such that every formula of the logic, there is an equivalent formula in the normal form.

Three normal forms are important for propositional logic.

- Negation normal form (NNF)
- Disjunctive normal form (DNF)
- Conjunctive normal form (CNF)

Negation normal form (NNF)

- Only logical connectives: $\wedge$, $\vee$, and $\neg$.
- $\neg$ only appear in literals

Atom := $\top$ | $\bot$ | Variable

Literal := Atom | $\neg$Atom

$\neg p \wedge q$ is in NNF, but $\neg(p \vee q)$ is not in NNF

Formula := Literal | Formula $\vee$ Formula | Formula $\wedge$ Formula

Every wff α (not containing $\leftrightarrow$) can be transformed into an equivalent NNF α' with linear increase in the **size** (i.e., # of symbols) of the formula:

- Rewrite $\rightarrow$: $(\alpha_1 \rightarrow \alpha_2) \Leftrightarrow (\neg\alpha_1 \vee \alpha_2)$
- Apply De Morgan's rules:
  - $\neg(\alpha_1 \vee \alpha_2)$: $\neg(\alpha_1 \vee \alpha_2) \Leftrightarrow (\neg\alpha_1 \wedge \neg\alpha_2)$
  - $\neg(\alpha_1 \wedge \alpha_2)$: $\neg(\alpha_1 \wedge \alpha_2) \Leftrightarrow (\neg\alpha_1 \vee \neg\alpha_2)$
- Rewrite double negations: $\neg\neg\alpha_1 \Leftrightarrow \alpha_1$
- $\neg\top \Leftrightarrow \bot$
- $\neg\bot \Leftrightarrow \top$

Question: what if the original formula contains $\leftrightarrow$?

$(\alpha_1 \leftrightarrow \alpha_2) \Leftrightarrow (\alpha_1 \rightarrow \alpha_2) \wedge (\alpha_2 \rightarrow \alpha_1)$

Disjunctive normal form (DNF)

- Formula is in NNF
- Formula is a conjunction of disjunctions of literals, i.e., of the form:

$$\bigvee_i (\bigwedge_j l_{ij})$$

Atom $:= \top \mid \bot \mid$ Variable

Literal $:=$ Atom $\mid$ $\neg$Atom

Clause $:=$ Literal $\mid$ Literal $\wedge$ Clause

Formula $:=$ Clause $\mid$ Clause $\vee$ Formula

e.g., $(p \wedge q \wedge \neg r) \vee (\neg p \wedge s) \vee (r \wedge \neg q \wedge \neg s)$

Every wff α can be transformed into an equivalent DNF α', while potentially exponentially increasing the **size** (# of terms) of the formula:

- Convert α to NNF
- Distribute $\wedge$ over $\vee$ (cause of exponential increase):
  - $\alpha_1 \wedge (\alpha_2 \vee \alpha_3) \Leftrightarrow (\alpha_1 \wedge \alpha_2) \vee (\alpha_1 \wedge \alpha_3)$
  - $(\alpha_1 \vee \alpha_2) \wedge \alpha_3 \Leftrightarrow (\alpha_1 \wedge \alpha_3) \vee (\alpha_2 \wedge \alpha_3)$
- Flatten out nested conjunctions and disjunctions.

Exercise

Translate the formula into DNF: $\neg((p \vee \neg q) \to r)$.

Submit your answers at

`https://pollev.com/andreww095`

NNF translation:

- Rewrite $\to$: $(\alpha_1 \to \alpha_2) \Leftrightarrow (\neg\alpha_1 \vee \alpha_2)$
- Apply De Morgan's rules:
  - $\neg(\alpha_1 \vee \alpha_2)$:
    $\neg(\alpha_1 \vee \alpha_2) \Leftrightarrow (\neg\alpha_1 \wedge \neg\alpha_2)$
  - $\neg(\alpha_1 \wedge \alpha_2)$:
    $\neg(\alpha_1 \wedge \alpha_2) \Leftrightarrow (\neg\alpha_1 \vee \neg\alpha_2)$
- Rewrite double negations: $\neg\neg\alpha_1 \Leftrightarrow \alpha_1$
- $\neg\top \Leftrightarrow \bot, \neg\bot \Leftrightarrow \top$

DNF translation:

- Convert α to NNF
- Distribute $\wedge$ over $\vee$ (cause of exponential increase):
  - $\alpha_1 \wedge (\alpha_2 \vee \alpha_3) \Leftrightarrow (\alpha_1 \wedge \alpha_2) \vee (\alpha_1 \wedge \alpha_3)$
  - $(\alpha_1 \vee \alpha_2) \wedge \alpha_3 \Leftrightarrow (\alpha_1 \wedge \alpha_3) \vee (\alpha_2 \wedge \alpha_3)$
- Flatten out nested conjunctions and disjunctions.

Exercise

Translate the formula into DNF: $\neg((p \vee \neg q) \to r)$.

- $\Leftrightarrow \neg(\neg(p \vee \neg q) \vee r)$ (Rewrite $\to$)
- $\Leftrightarrow \neg\neg(p \vee \neg q) \wedge \neg r$ (De Morgan's rules)
- $\Leftrightarrow (p \vee \neg q) \wedge \neg r$ (Rewrite $\neg\neg$)
- $\Leftrightarrow (p \wedge \neg r) \vee (\neg q \wedge \neg r)$ (Distribute $\wedge$ over $\vee$)

Conjunctive normal form (CNF)

- Formula is in NNF
- Formula is a disjunction of conjunctions of literals, i.e., of the form:

$$\bigwedge_i \left(\bigvee_j l_{ij}\right)$$

Atom := $\top$ | $\bot$ | Variable

Literal := Atom | $\neg$Atom

Clause := Literal | Literal $\wedge$ Clause

Formula := Clause | Clause $\vee$ Formula

e.g., $(p \vee q \vee \neg r) \wedge (\neg p \vee s) \wedge (r \vee \neg q \vee \neg s)$

Every wff α can be transformed into an equivalent CNF α', while potentially exponentially increasing the size of the formula:

- Convert α to NNF
- Distribute $\vee$ over $\wedge$ (cause of exponential increase):
  - $\alpha_1 \vee (\alpha_2 \wedge \alpha_3) \Leftrightarrow (\alpha_1 \vee \alpha_2) \wedge (\alpha_1 \vee \alpha_3)$
  - $(\alpha_1 \wedge \alpha_2) \vee \alpha_3 \Leftrightarrow (\alpha_1 \vee \alpha_3) \wedge (\alpha_2 \vee \alpha_3)$
- Flatten out nested conjunctions and disjunctions.

DNF vs. CNF for satisfiability-checking

DNF:

- Deciding satisfiability can be done in linear time with one traversal of the clauses.
  - The DNF is unsat. iff every clause contains both a literal and its negation.
- Converting into an equivalent DNF can result in exponential size increase.

CNF:

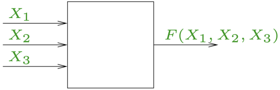
- Deciding satisfiability is hard.
- Converting into an equivalent CNF can result in exponential size increase.
- Converting into an **equi-satisfiable** (i.e., has the same satisfiability) CNF can be done with linear size increase!

Modern SAT solvers expect CNF input.

They choose to optimize the runtime of the decision procedure rather than the conversion procedure.

Boolean Gates

Consider an electrical device having n inputs and one output. Assume that to each input we apply a signal that is either T or F, and that this uniquely determines whether the output is T or F.

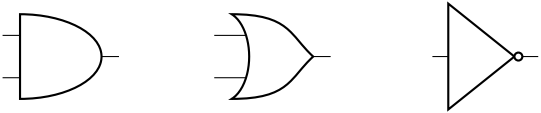


The behavior of such a device is described by a Boolean function:

$$F(X_1, \ldots, X_n) = \text{ the output signal given the input signals } X_1, \ldots, X_n.$$

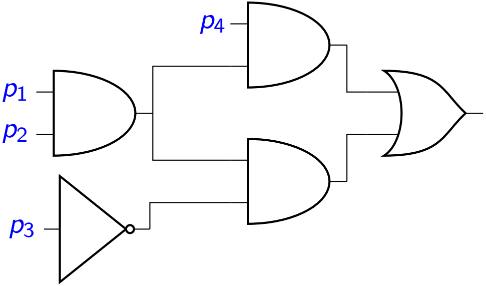
We call such a device a **Boolean gate**.

The most common Boolean gates are **AND**, **OR**, and **NOT** gates.



Boolean Circuits

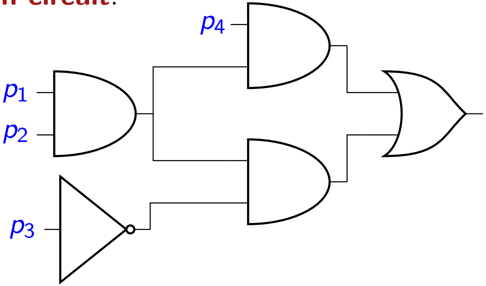
The inputs and outputs of Boolean gates can be connected together to form a **combinational boolean circuit**.



A combinational Boolean circuit corresponds to a **directed acyclic graph** (DAG) whose leaves are **inputs** and each of whose nodes is labeled with the name of a Boolean gate. One or more of the nodes may be identified as outputs.

Boolean Circuits

The inputs and outputs of Boolean gates can be connected together to form a **combinational Boolean circuit**.



There is a natural correspondence between Boolean circuits and formulas of propositional logic. The formula corresponding to the above circuit is:

$$(p_4 \wedge (p_1 \wedge p_2)) \vee ((p_1 \wedge p_2) \wedge \neg p_3).$$

A satisfying assignment for this formula gives the values that must be applied to the inputs of the circuit in order to set the output of the circuit to true.

Sharing Sub-formulas

$$(p_4 \wedge (p_1 \wedge p_2)) \vee ((p_1 \wedge p_2) \wedge \neg p_3)$$

There is an redundancy in the formula: the formula $(p_1 \wedge p_2)$ appears twice. For larger circuits, this sort of redundancy can result in an exponential blowup in formula size.

Since we are only concerned with the **satisfiability** of the formula, we can overcome this inefficiency by introducing new propositional symbols. These new symbols essentially act as placeholders for redundant sub-expressions.

Sharing Sub-formulas

Original formula:

$$(p_4 \land (p_1 \land p_2)) \lor ((p_1 \land p_2) \land \neg p_3)$$

New formula:

$$((p_4 \land p_5) \lor (p_5 \land \neg p_3)) \land (p_5 \leftrightarrow (p_1 \land p_2))$$

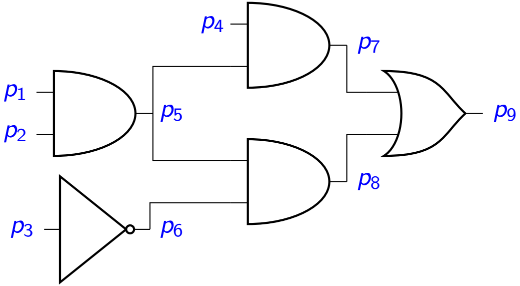
Discuss with your neighbors. Is the new formula logically equivalent to the original formula?

No, but it is equisatisfiable (i.e. the original formula is satisfiable iff the new formula is satisfiable).

Converting to CNF: Tseitin's Transformation

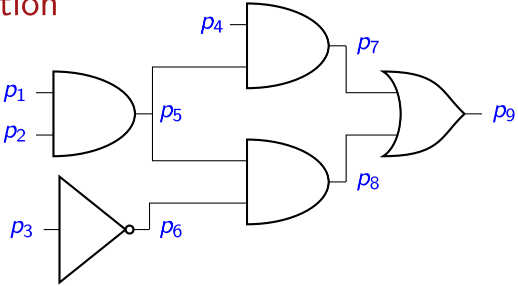
This same idea is behind a simple algorithm for converting any propositional formula (or an associated Boolean circuit) into an equisatisfiable formula in conjunctive normal form (CNF) in linear time. We will view the formula or circuit as a directed acyclic graph (DAG).

Step 1: Label each non-leaf node of the DAG with a new propositional symbol.



Converting to CNF: Tseitin's Transformation

Step 2: Construct a conjunction of disjunctive clauses which relate the inputs of that node to its output (the new propositional symbol).



$(p_1 \wedge p_2) \leftrightarrow p_5$
$\Rightarrow ((p_1 \wedge p_2) \rightarrow p_5) \wedge (p_5 \rightarrow (p_1 \wedge p_2))$
$\Rightarrow (\neg(p_1 \wedge p_2) \vee p_5) \wedge (\neg p_5 \vee (p_1 \wedge p_2))$
$\Rightarrow (\neg p_1 \vee \neg p_2 \vee p_5) \wedge (\neg p_5 \vee p_1) \wedge (\neg p_5 \vee p_2)$

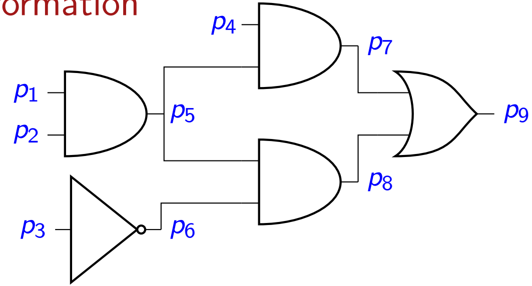
$(\neg p_3) \leftrightarrow p_6$
$\Rightarrow ((\neg p_3) \rightarrow p_6) \wedge (p_6 \rightarrow (\neg p_3))$
$\Rightarrow (p_3 \vee p_6) \wedge (\neg p_6 \vee \neg p_3)$

$(p_4 \wedge p_5) \leftrightarrow p_7$
$\Rightarrow (\neg p_4 \vee \neg p_5 \vee p_7) \wedge (\neg p_7 \vee p_4) \wedge (\neg p_7 \vee p_5)$

$(p_5 \wedge p_6) \leftrightarrow p_8$
$\Rightarrow (\neg p_5 \vee \neg p_6 \vee p_8) \wedge (\neg p_8 \vee p_5) \wedge (\neg p_8 \vee p_6)$

$(p_7 \vee p_8) \leftrightarrow p_9$
$\Rightarrow ((p_7 \vee p_8) \rightarrow p_9) \wedge (p_9 \rightarrow (p_7 \vee p_8))$
$\Rightarrow (\neg(p_7 \vee p_8) \vee p_9) \wedge (\neg p_9 \vee (p_7 \vee p_8))$
$\Rightarrow (\neg p_7 \vee p_9) \wedge (\neg p_8 \vee p_9) \wedge (\neg p_9 \vee p_7 \vee p_8)$

Converting to CNF: Tseitin's Transformation



Step 3: The conjunction of all of these clauses together with a single clause consisting of the symbol for the root node is satisfiable iff the original formula is satisfiable.

$\alpha := (p_4 \wedge (p_1 \wedge p_2)) \vee ((p_1 \wedge p_2) \wedge \neg p_3)$
$\Rightarrow$
$(\neg p_1 \vee \neg p_2 \vee p_5) \wedge (\neg p_5 \vee p_1) \wedge (\neg p_5 \vee p_2) \wedge$
$(p_3 \vee p_6) \wedge (\neg p_6 \vee \neg p_3) \wedge$
$(\neg p_4 \vee \neg p_5 \vee p_7) \wedge (\neg p_7 \vee p_4) \wedge (\neg p_7 \vee p_5) \wedge$
$(\neg p_5 \vee \neg p_6 \vee p_8) \wedge (\neg p_8 \vee p_5) \wedge (\neg p_8 \vee p_6) \wedge$
$(\neg p_7 \vee p_9) \wedge (\neg p_8 \vee p_9) \wedge (\neg p_9 \vee p_7 \vee p_8) \wedge$
(p_9)

Decision procedure for propositional logic

We will describe procedures for checking the satisfiability of a wff in propositional logic.

From now on, unless otherwise indicated, we assume formulas are in CNF.

We denote a formula in CNF as Δ, which can be regarded as a set of clauses $\{C_1, \dots C_n\}$. Each clause C_i can be regarded as a set of literals $\{l_1, \dots, l_n\}$.

Δ is satisfiable if and only if there exists a variable assignment that satisfies each clause C_i.

Example: the CNF formula $\Delta := (p_1 \vee p_3) \wedge (\neg p_1 \vee p_2 \vee \neg p_3)$ can be represented as $\{\{p_1, p_3\}, \{\neg p_1, p_2, \neg p_3\}\}$.

$\{p_1 : 1, p_2 : 1, p_3 : 0\}$ is a satisfying assignment to Δ.

SAT Solver Overview: features

Software for tackling the satisfiability problem of CNF formulas are called **SAT-solvers**.

Two main categories of modern SAT solvers:

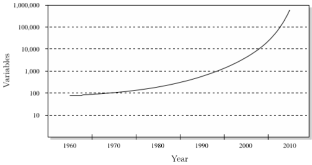
- Backtracking algorithms
  - traversing and backtracking on a binary tree
  - Sound and complete.
- Stochastic search
  - solver guesses a full assignment
  - if the formula is evaluated to false under this assignment, starts to flip values of variables according to some (greedy) heuristic.
  - Sound and incomplete.

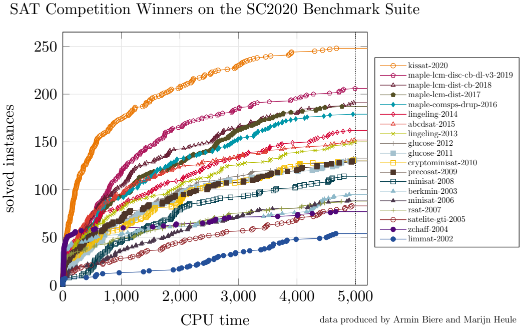
We focus on the former in this class.

SAT Solver Overview: performance

- How well do SAT solvers do in practice, since they're trying to solve an NP-compete problem?
  - Modern SAT solvers can solve many real-life CNF formulas with hundreds of thousands or even millions of variables in a reasonable amount of time.
  - There are also instances of problems two orders of magnitude smaller that these tools cannot solve.
  - In general, it is very hard to predict which instance is going to be hard to solve, without actually attempting to solve it
- SAT portfolio solvers: use machine-learning techniques to extract features of CNF formulas in order to select the most suitable SAT solver for the job

SAT Solver Overview: performance





- Left: Size of industrial CNF formulas (y-axis) that are regularly solved by SAT solvers in a few hours according to year (x-axis). Instances are generated for solving realistic problems like verification of circuits and planning problems.
- Right: Top contenders in annual SAT solver competitions from 2002-2020. A data point means some number of benchmarks (y-axis) was solved with some amount of time (x-axis). Num. instances solved within 20 minutes more than doubled in the decade.

The DIMACS format

A standard format for CNF formulas accepted by most (if not all) modern SAT solvers.

- Comment lines: Start with a lower-case letter **c**
- Problem line: **p cnf** $<$ **#variables** $>$ $<$ **#clauses** $>$
- Clause lines:
  - Each variable is assigned a unique index i greater than 0
  - A positive literal is represented by an index
  - A negative literal is represented by the negation of an index
  - A clause is represented as a list of literals
  - The value "0" is used to mark the end of a clause.

Example:

$$(p_1 \vee \neg p_3) \wedge (p_2 \vee p_3 \vee \neg p_1) \Rightarrow$$

```
c example.cnf
p cnf 3 2
1 -3 0
2 3 -1 0
```

Basic SAT solvers

- 1960: Davis-Putnam (DP) algorithm
- 1961: Davis–Putnam–Logemann–Loveland (DPLL) algorithm
- Modern SAT solver based on **Conflict-driven clause learning (CDCL)** (1996) is derived from DP and DPLL.

A key feature of CNF: Resolution

Starting from the initial set of clauses Δ, there is a simple inference rule, called **resolution**, by which new clauses can be derived:

$$\frac{p \in \mathcal{V} \quad p \in C_1 \quad \neg p \in C_2 \quad C_1, C_2 \in \Delta}{\Delta \cup \{(C_1 - \{p\}) \cup (C_2 - \{\neg p\})\}} \ (\textbf{resolution})$$

The rule reads: "**If C_1 and C_2 are satisfiable, then the clause below is satisfiable.**"

The new clause is called a **p-resolvant** (or simply **resolvant** when the context is clear) derived from C_1 and C_2.

The resolvant can be added as a clause to Δ.

Example: Consider $\Delta := \{\{p_1, p_3\}, \{p_2, \neg p_3\}\}$. $\{p_1, p_2\}$ is a p_3-resolvant of the two clauses. Δ and $\Delta \cup \{p_1, p_2\}$ are equivalent. Δ and $\{p_1, p_2\}$ are equi-satisfiable.

Proof by resolution

Proving that a CNF is unsatisfiable can be done with just the resolution rule.

Example: Prove that the following CNF formula is unsatisfiable.

$$\begin{aligned}\Delta &:= \{C_1, C_2, C_3, C_4\} \\ &:= \{\{p_1, p_2\}, \{p_1, \neg p_2\}, \{\neg p_1, p_3\}, \{\neg p_1, \neg p_3\}\}\end{aligned}$$

1. $C_5 := \{p_1\}$ (C_1, C_2)
2. $C_6 := \{p_3\}$ (C_3, C_5)
3. $C_7 := \{\neg p_3\}$ (C_4, C_5)
4. $C_8 := \{\}$ (C_6, C_7)

A resolution proof is a proof by contradiction: if Δ is satisfiable, then C_8, the empty clause, is satisfiable.

Proof by resolution

Imagine a procedure for SAT with resolution:

- Apply resolution until either
  1. an empty clause is derived (return UNSAT)
  2. it can no longer be applied to produce new clauses (return SAT)

Let us then add the clashing clause rule, the satisfying rule and refuting rule:

$$\frac{\{\} \in \Delta}{\text{UNSAT}}\ (\textbf{unsat}) \qquad \frac{\text{Cannot apply Res. to produce new clauses}}{\text{SAT}}\ (\textbf{sat})$$

Is the resolution proof system refutation sound? Yes.
Is the resolution proof system solution sound? Yes.
Is the resolution proof system complete? Yes.
Is the resolution proof system terminating? Yes.

Unit resolution

The **unit resolution rule** is a special case of the resolution rule where one clause, called the **unit clause**, consists of a single literal (i.e., a variable or the negation of a variable):

$$\frac{p \in \mathcal{V} \quad C_1 := \{p\} \quad \neg p \in C_2 \quad C_1, C_2 \in \Delta}{\Delta \cup (C_2 - \{\neg p\})\}} \; (\textbf{unit resolution 1})$$

$$\frac{p \in \mathcal{V} \quad C_1 := \{\neg p\} \quad p \in C_2 \quad C_1, C_2 \in \Delta}{\Delta \cup (C_2 - \{p\})\}} \; (\textbf{unit resolution 2})$$

A proof system with unit resolution alone is incomplete (e.g., a CNF where each clause has length ≥ 2).

Modern SAT solvers use unit resolution in combination with backtracking search to implement a sound and complete procedure for deciding CNF formulas.

DP Algorithm

The DP algorithm leverages 4 satisfiabiliy-preserving transformations:

- Unit propagataion rule (or 1-literal rule)
- Pure literal rule (or affirmation-negation rule)
- Resolution rule (or rule for eliminiating atomic formulas)
- Clashing clause rule

The first two rules reduce the total number of literals in the formula. The third rule reduces the number of variables in the formula.

First algorithm to try something more sophisticated than the truth table.

By repeatedly applying these rules, eventually we obtain a formula containing an empty clause, indicating unsatisfiability, or a formula with no clauses, indicating satisfiability.

DP Algorithm: unit propagation rule

Also called the **1-literal rule**.

Premise: The cnf Δ contains a unit clause, $\{p\}$. We assume all double negations are collapsed (i.e., $\neg\neg p \Rightarrow p$).

Conclusion:

- Remove all instances of $\neg p$ from clauses in the formula (shortening the corresponding clauses).
- Remove all clauses containing p (including the unit clause itself).

Justification: The unit clause must be satisfied, because we have a CNF. This rule effectively assigns p to $true$. Thus, $\neg p$ cannot be used to satisfy another clause.

Example: $\Delta_0 := \{p_1\}, \{p_1, p_4\}, \{p_2, p_3, \neg p_1\}$

$\Delta_1 := \{p_4\}, \{p_2, p_3\}$ (unit propagation on p_1)

$\Delta_2 := \{p_2, p_3\}$ (unit propagation on p_4)

DP Algorithm: pure literal rule

Also called the **affirmation-negation rule**.

Premise: A variable p appears only positively or only negatively in Δ.

Conclusion: delete all clauses containing that variable.

Justification: If a literal only ever appears positively/negatively, its atom can be assigned in a way that causes the literal to evaluate to true. Thus, all clauses containing this literal can be deleted since they are satisfied.

Example: $\Delta_0 := \{p_1, p_2, \neg p_3\}, \{\neg p_1, p_4\}, \{\neg p_3, \neg p_2\}, \{\neg p_3, \neg p_4\}$

$\Delta_1 := \{\neg p_1, p_4\}$ (pure literal rule on p_3)

DP Algorithm: resolution rule

Also called the **rule for eliminating atomic formulas**.

Premise: There exists two different clauses $C, C' \in \Delta$ and variable p, where $p \in C$ and $\neg p \in C'$.

Conclusion:

- Let P be the set of clauses in Δ where p occurs positively.
- Let N be the set of clauses in Δ where p occurs negatively.
- Replace the clauses in P and N with those obtained by resolution on p using all pairs of clauses from P and N.

Example: $\Delta_0 := \{p_1, p_2\}, \{\neg p_1, p_3\}, \{\neg p_1, \neg p_3, p_4\}$

$\Delta_1 := \{p_2, p_3\}, \{p_2, \neg p_3, p_4\}$ (resolution rule on p_1)

DP Algorithm: clashing clause rule

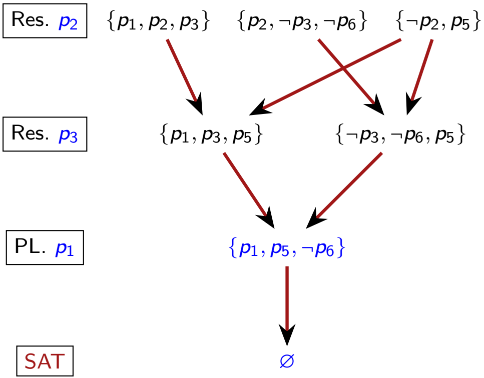
Premise: a clause $C \in \Delta$ contains both p and $\neg p$.

Conclusion: remove C from Δ.

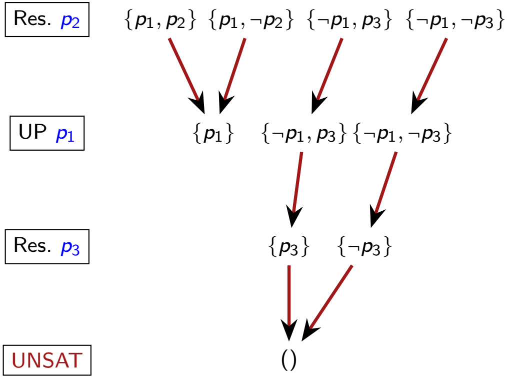
Justification: C is satisfied regardless of whether p is assigned 0 or 1.

DP Example

$\Delta := \{\{p_1, p_2, p_3\}, \{p_2, \neg p_3, \neg p_6\}, \{\neg p_2, p_5\}\}$



$\Delta := \{\{p_1, p_2\}, \{p_1, \neg p_2\}, \{\neg p_1, p_3\}, \{\neg p_1, \neg p_3\}\}$



From DP to DPLL

Resolution can increase the number of *clauses* but *not variables*.

Question: if a variable appears positively in 3 clauses and negatively in 3 clauses. How many clauses after applying resolution? 9 in the worst case.

In the worst case, the resolution rule can cause a quadratic expansion every time it is applied. For large formulas, this can quickly exhaust the available memory.

The DPLL algorithm replaces resolution with a splitting rule:

- Choose a literal l occurring in the formula.
- Let Δ be the current set of clauses.
- Test the satisfiability of $\Delta \cup \{l\}$.
  - If satisfiable, return True.
  - If unsatisfiable, backtrack and return the result of $\Delta \cup \{\neg l\}$ for satisfiability.

Search: find a satisfying assignment by guessing possible assignments one by one.

Deduction: Deduce new facts from a set of know facts.

Backtrack and make a different guess if we guessed wrong.

We discuss DPLL in more details next time.