Mathematical and Logical Foundations of Computer Science

Lecture 9 - Propositional Logic (SAT)

Vincent Rahli

(some slides were adapted from Rajesh Chitnis' slides)

University of Birmingham

Where are we?

- Symbolic logic
- **Propositional logic**
- Predicate logic

Today

- History of Computing
- SAT (first $\mathcal{NP}$-hard problem)
- Algorithms for SAT

Recap: Propositional logic syntax

Syntax:

$$P ::= a \mid P \wedge P \mid P \vee P \mid P \rightarrow P \mid \neg P$$

Two special atoms:

- $\top$ which stands for True
- $\bot$ which stands for False

We also introduced four connectives:

- $P \wedge Q$: we have a proof of both P and Q
- $P \vee Q$: we have a proof of at least one of P and Q
- $P \rightarrow Q$: if we have a proof of P then we have a proof of Q
- $\neg P$: stands for $P \rightarrow \bot$

Recap: Normal forms

Among the formulas equivalent to a given formula, some are of particular interest (the variables here stand for atoms):

- **Conjunctive Normal forms** (CNF)
  - $(A \lor B \lor C) \land (D \lor X) \land (\neg A)$
  - ANDs of ORs of literals (atoms or negations of atoms)
  - A **clause** in this context is a disjunction of literals
- **Disjunctive Normal Form** (DNF)
  - $(P \land Q \land A) \lor (R \land \neg Q) \lor (\neg A)$
  - ORs of ANDs of literals
  - A **clause** in this context is a conjunction of literals

Theorem: Every proposition is equivalent to a formula in CNF!

Theorem: Every proposition is equivalent to a formula in DNF!

Recap: Every proposition can be expressed in DNF

Every proposition can be expressed in DNF (ORs of ANDs)!

Express $(P \to Q) \wedge Q$ in DNF

We do it using a truth table

P	Q	$(P \to Q)$	$(P \to Q) \wedge Q$
T	**T**	**T**	**T**
T	**F**	**F**	**F**
F	**T**	**T**	**T**
F	**F**	**T**	**F**

- Enumerate all the **T** rows from the conclusion column
  - Row 1 gives $P \wedge Q$
  - Row 3 gives $\neg P \wedge Q$
- Take **OR** of these formulas
- **Final answer** is $(P \wedge Q) \vee (\neg P \wedge Q)$

Recap: Every formula can be expressed in CNF

Every proposition can be expressed in CNF (ANDs of ORs)!

Express $(P \to Q) \wedge Q$ in CNF

We do it by using a truth table

P	Q	$(P \to Q)$	$(P \to Q) \wedge Q$
T	**T**	**T**	**T**
T	**F**	**F**	**F**
F	**T**	**T**	**T**
F	**F**	**T**	**F**

- Enumerate all the **F** rows from the conclusion column
  - Row 2 gives $P \wedge \neg Q$
  - Row 4 gives $\neg P \wedge \neg Q$
- Do **AND** of negations of each of these formulas
- We obtain $\neg(P \wedge \neg Q) \wedge \neg(\neg P \wedge \neg Q)$
- **Finally**: equivalent to $(\neg P \vee Q) \wedge (P \vee Q)$ by De Morgan

Satisfiability of CNF formulas

Problem definition: Given a CNF formula can we set **T** or **F** value to each variable to satisfy the formula?

- **Example**: Consider the formula $(A \vee \neg B) \wedge (C \vee B)$
- Is it satisfiable?
- Satisfiable by setting $A = \mathbf{T}$, $B = \mathbf{F}$ and $C = \mathbf{T}$
- Known as **CNF Satisfiability** or simply **SAT**

First a bit of history

History of Computing

1930s

- Alan Turing invented the Turing Machine in 1936
- Mathematical model of computable functions (as abstract machines)
- Basis of modern computers
- Biography: *Alan Turing: The Enigma*
- Movie: *The Imitation Game*

1940s and 1950s

- Code-breaking by Allies in Bletchley Park

  Go visit the *National Museum of Computing*
- Should not really have been breakable

  Made use of manual & hardware errors
- Alan Turing was heavily involved

History of Computing

1960s

- People began to look at **general** ways to solve a problem, rather than solving given instance!
  - Is $(A \vee \neg B) \wedge (\neg A) \wedge (B \vee Z \vee \neg X)$ satisfiable?
  - How (fast) can we check in general if a CNF formula is satisfiable?
- Many **known** problems had **polynomial** running time
  - Actually even n^4 or smaller
- Polynomial time became accepted as **standard of efficiency**
  - $\mathcal{P}$: The class of problems solvable in polynomial time (in size of input)
- **Claim**: Any exponential (ultimately) **beats** any polynomial
  - $1.0000000001^n > n^{10000000000000000000000000}$ if n is large enough

History of Computing

1970s

- But still **many** problems no one knew how to solve in polynomial time!
- **CNF satisfiability (SAT)**
  - Say we have N atoms and M clauses
  - No known algorithm to solve in time polynomial in N and M
  - **Brute force**: does 2^N truth assignments, and checks in N time if each of the M clauses is satisfied
  - So, total running time is $2^N \cdot N \cdot M$
  - Note that the input size is $N + M$
- Can we design a polynomial time algorithm for SAT?
- Or show that such an algorithm cannot exist?
- $\mathcal{NP}$: class of problems where we can verify a potential solution in polynomial time

$\mathcal{P}$ vs. $\mathcal{NP}$

$\mathcal{P}$: the class of problems which we can solve in polynomial time
$\mathcal{NP}$: the class of problems where we can verify a potential solution/answer in polynomial time

Clearly, $\mathcal{P} \subseteq \mathcal{NP}$ (solving is a (hard) way of verifying)

What about the other direction? Is $\mathcal{P} = \mathcal{NP}$?

- Status unknown!
- Million dollar question

What do most people believe?

- $\mathcal{P}$ is not equal to $\mathcal{NP}$

Why haven't we been able to prove it then?

- Hard to rule out all possible polytime algorithms?

Hardness for the class $\mathcal{NP}$

$\mathcal{NP}$: the class of problems where we can verify a potential solution/answer in polynomial time

Definition: A problem is $\mathcal{NP}$-hard if it is **at least as hard as** any problem in $\mathcal{NP}$.

More precisely, a problem X is $\mathcal{NP}$-hard if any problem $Y \in \mathcal{NP}$ can be solved

- using an oracle for solving X
- plus a polynomial overhead for translating between X and Y

If $\mathcal{P} \neq \mathcal{NP}$ then a problem being $\mathcal{NP}$-hard means it cannot be solved in polynomial time!

Great, except no one knew how to show existence of a single $\mathcal{NP}$-hard problem!

The first $\mathcal{NP}$-hard problem

Cook-Levin Theorem (1971/1973):
CNF-Satisfiability (**SAT**) is $\mathcal{NP}$-hard

How do you show a problem, say X, is $\mathcal{NP}$-hard?

- A polytime reduction from any of the known $\mathcal{NP}$-hard problems, say SAT, to X
- That is, show how you can solve SAT using an oracle for X
- Plus a polynomial overhead for the translation

Tens of thousands of problems known to be $\mathcal{NP}$-hard

Significance of SAT

Many practical problems can be encoded into SAT
(e.g., formal verification, planning/scheduling, etc.)

A possible solution (valuation) can be verified "efficiently"

No known algorithm to solve the problem "efficiently" in all cases

In practice, SAT solvers are very efficient
($\mathcal{NP}$-hardness is the worst case)

Special cases

Let n-**SAT** be the SAT problem restricted to n-CNFs, i.e., where clauses are disjunctions of n literals

- 1-SAT is in $\mathcal{P}$
- 2-SAT is in $\mathcal{P}$
- 3-SAT is $\mathcal{NP}$-hard

Why not consider DNF instead of CNF?

Theorem: Any propositional formula can be expressed in CNF

Theorem: Any propositional formula can be expressed in DNF

Theorem: CNF satisfiability is $\mathcal{NP}$-hard

How hard is DNF satisfiability?

- Example of a DNF formula:
  $(A \wedge \neg B \wedge C) \vee (\neg X \wedge Y) \vee (Z)$
- Is it satisfiable?
- **Trivial** to check in polytime!
- Just pick any clause, and set variables to **T** or **F**.

Why not use DNFs then?

Because changing a formula from CNF to DNF can cause exponential blowup!

Why not consider DNF instead of CNF?

Because changing a formula from CNF to DNF can cause exponential blowup!

Convert $(A \vee B) \wedge (C \vee D)$ into DNF

Remember: $P \wedge (Q \vee R) \leftrightarrow (P \wedge Q) \vee (P \wedge R)$

$$
\begin{array}{ll}
 & (A \vee B) \wedge (C \vee D) \\
\leftrightarrow & ((A \vee B) \wedge C) \vee ((A \vee B) \wedge D) \\
\leftrightarrow & (C \wedge (A \vee B)) \vee (D \wedge (A \vee B)) \\
\leftrightarrow & (C \wedge A) \vee (C \wedge B) \vee (D \wedge A) \vee (D \wedge B)
\end{array}
$$

Consider the CNF formula: $(P_1 \vee Q_1) \wedge \cdots \wedge (P_n \vee Q_n)$

Expressing this formula in DNF requires 2^n clauses

Algorithms for SAT?

Brute force for SAT with N variables and M clauses needs $2^N \cdot N \cdot M$ time

- There are 2^N truth assignments
- For each truth assignment and each clause, verify if it is satisfied in N time

Can we solve SAT faster than 2^N? Say 1.999999999^N?

Conjecture (Strong Exponential Time Hypothesis (SETH)): SAT cannot be solved in $(2-\alpha)^N \cdot \text{poly}(N+M)$ time for any constant $\alpha > 0$

SAT solvers

Many state-of-the-art SAT solvers are based on the **Davis-Putman-Logemann-Loveland** algorithm (DPLL)

Basic idea (does a lot of pruning instead of brute force):

1. **Easy** cases
   - Atom p only appears as either p or $\neg p$ (but not both): assign truth value accordingly
2. **Branch** on choosing a variable p and set a truth value to it
   - This choice needs to be done **cleverly**
   - If $p =$ **T**: remove all clauses containing p and remove all literals $\neg p$ from clauses
   - If $p =$ **F**: remove all clauses containing $\neg p$ and remove all literals p from clauses
3. Keep running the above steps **until**
   - All clauses have been removed (all true): return SAT
   - One clause is empty (one is false): **backtrack** in Step 2 and choose a different truth value for p; if it is not possible to backtrack, return UNSAT

SAT solvers

Apply the DPLL algorithm to
$(\neg p \vee q \vee r) \wedge (p \vee q \vee r) \wedge (p \vee q \vee \neg r) \wedge (\neg p \vee \neg q \vee r)$

Here is a possible run of the algorithm:

$(\neg p \vee q \vee r) \wedge (p \vee q \vee r) \wedge (p \vee q \vee \neg r) \wedge (\neg p \vee \neg q \vee r)$

$\quad p = \mathbf{T}$

$(q \vee r) \wedge (\neg q \vee r)$

$\quad q = \mathbf{T}$

(r)

$\quad r = \mathbf{T}$

SAT

SAT Solvers

Let us use this SAT solver: https://jfmc.github.io/z3-play/

two variables, two clauses:

$(p \vee q) \wedge (\neg q)$

```
(declare-const p Bool)
(declare-const q Bool)
(define-fun conjecture () Bool
  (and (or p q) (not q))
)
(assert conjecture)
(check-sat)
(get-model)
```

SAT Solvers

Let us use this SAT solver: https://jfmc.github.io/z3-play/

three variables, three clauses:

$(p \lor q \lor r) \land (\neg p \lor \neg q) \land (q \lor \neg r)$

```
(declare-const p Bool)
(declare-const q Bool)
(declare-const r Bool)
(define-fun conjecture () Bool
  (and (or p q r) (or (not p) (not q)) (or q (not r)))
)
(assert conjecture)
(check-sat)
(get-model)
```

SAT Solvers

Let us use this SAT solver: https://jfmc.github.io/z3-play/

four variables, five clauses:

$(p \vee q \vee \neg r) \wedge (q \vee r \vee \neg s) \wedge (\neg p \vee q \vee r) \wedge (\neg p) \wedge (\neg r \vee s)$

```
(declare-const p Bool)
(declare-const q Bool)
(declare-const r Bool)
(declare-const s Bool)
(define-fun conjecture () Bool
  (and (or p q (not r)) (or q r (not s)) (or (not p) q r)
        (not p) (or (not r) s)
  )
)
(assert conjecture)
(check-sat)
(get-model)
```

SAT Solvers

Let us use this SAT solver: `https://jfmc.github.io/z3-play/`

five variables, eight clauses:

$$(p \vee t \vee s) \wedge (q \vee r \vee \neg s \vee \neg t) \wedge (\neg t \vee r) \wedge (p \vee \neg q \vee s)$$
$$\wedge (p \vee q \vee r \vee \neg t) \wedge (q \vee r \vee \neg s) \wedge (p \vee \neg s) \wedge (\neg p \vee q \vee s \vee t)$$

```
(declare-const p Bool)
(declare-const q Bool)
(declare-const r Bool)
(declare-const s Bool)
(declare-const t Bool)
(define-fun conjecture () Bool
  (and (or p t s) (or q r (not s) (not t)) (or (not t) r) (or p (not q) s)
       (or p q r (not t)) (or q r (not s)) (or p (not s)) (or (not p) q s t)
  )
)
(assert conjecture)
(check-sat)
(get-model)
```

Conclusion

What did we cover today?

- History of Computing
- SAT (first $\mathcal{NP}$-hard problem)
- Algorithms for SAT

Next time?

- Propositional logic (wrap-up)