

# Lecture 11: NP-complete Problems

David Dill

Department of Computer Science

# Outline

---

- NP-completeness
- SAT
- CSAT
- 3SAT

## NP-complete problems

This is about *decision problems* (problems with yes/no answers). “Problem” is just another name for “language”. A “problem instance” is a string.

Solving an instance  $x$  of problem  $L$  is answering whether  $x \in L$ .

The class  $P$  – problems that are solvable in polynomial time on an DTM. (P-time, polynomial time).

The class  $NP$  – problems that are solvable in polynomial time on an NTM.

Obviously  $P \subseteq NP$ .

No one knows whether  $NP \subseteq P$  (the famous  $P = NP$  problem).

**Definition** A problem  $L$  is NP-complete if it has two properties:

1.  $L$  is in NP.
2. There is a polynomial-time reduction from each problem  $L'$  in NP to  $L$ .

## Significance of NP-completeness

NP-complete problems are the “hardest” problems in NP.

If there are *any* problems in  $NP - P$ , the NP-complete problems are all there.

Every NP-complete problem can be translated in deterministic polynomial time to every other NP-complete problem.

So, if there is a P-time to one NP-complete problem, there is a P-time solution to *every* P-time problem.

## Basic Proof Strategy

NP-completeness is a good news/bad news situation. If it's NP-hard, that means that it's pretty hard. But, it could be worse!

So, a typical NP-completeness proof consists of two parts:

- Prove that the problem is in NP (i.e., it can be solved by an NTM in polynomial time).
- Prove that the problem is at least as hard as other problems in NP.

## Guess and verify

**Prove the that problem is in NP:** The usual strategy is to show that a solution can be *verified* in polynomial time with a deterministic Turing machine.

An NTM can pre-select all its choices at the beginning, write them on its tape, and then look them up later. So, computations can be regarded as a purely nondeterministic “guessing” phase followed by a deterministic “verification” phase.

The guess finds a *certificate* if the answer is “yes” that can be verified for correctness in polynomial time. A certificate is an example that shows that the “yes” answer is “true”.

Hence, an NTM can guess the solution nondeterministically, then accept if it checks out.

A DTM can simulate an ordinary computer in polynomial time, so it is sufficient to describe a polynomial-time verification algorithm that will run on any reasonable model of computation.

## NP-hardness by reduction

Typical method: Reduce a known NP-hard problem  $P_1$  to the new problem  $P_2$ .

A reduction is a polynomial-time translation of the problem, call it  $r$ .

If  $w$  is an instance of problem  $P_1$  (e.g., a string), then  $r(w)$  is an instance of problem  $P_2$ .

$r$  must have two properties:

- it preserves the answer. So  $w \in P_1$  iff  $r(w) \in P_2$ .
- $r(w)$  can be computed in time polynomial in  $|w|$ .

In practice, there are now thousands of known NP-complete problems. A good technique is to look for one similar to the one you are trying to solve.

*Repeated warning: Make sure you are reducing the known problem to the unknown problem!*

# SAT

---

Reduction proofs require that there be a known NP-hard problem. Where does the first one come from?

The primordial NP-Complete problem is satisfiability of a propositional logic formula (SAT)

**Cook's Theorem (proved later):** SAT is NP-complete.

Propositional logic:

- Propositional variables (e.g.,  $p_{91}$ ) which can take Boolean values.
- Propositional connectives  $\neg\alpha$ ,  $\alpha \vee \beta$ ,  $\alpha \wedge \beta$ .

Example:  $p \wedge \neg q \vee r$

**Truth assignment:** A function that assigns a truth value,  $\{T, F\}$ , to each propositional variable.

A truth assignment that makes a propositional formula true is said to *satisfy* it.

A propositional formula is *satisfiable* if there exists a satisfying assignment for it.

# Satisfiability

The satisfiability problem, “SAT” for short, is “Is propositional formula,  $\phi$ , satisfiable?”

**Fact:** *SAT is in NP.*

**proof:**

A truth assignment can be represented as a list of pairs of propositional variables and truth values. The size of this is at most linear in the size of  $\phi$ .

If  $\phi$  is satisfiable, we can *guess* the assignment, and then *verify* it in polynomial time.

To check: Evaluate the propositional formula to see whether it is true.

So SAT is in NP.

# CSAT

---

## Definitions

- A *literal* is a propositional variable or its negation (e.g.,  $p$  or  $\neg q$ ).
- A *clause* is a disjunction of literals (e.g.,  $(p \vee \neg q \vee r)$ ). Since  $\vee$  is associative, we can represent clauses as lists of literals.
- A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses e.g.,  $(p \vee q \vee \neg r) \wedge (\neg p \vee s \vee t \vee \neg u)$

We can represent AND formulas as lists of clauses.

So, solving satisfiability of CNF formulas is a restricted case of the SAT problem (it's called "CSAT").

# CSAT is NP-complete

**Theorem** CSAT is NP-complete

Who cares?

- Many practical SAT solvers use CNF now, and it is also used heavily in automatic theorem provers.
- It is often easier to do reductions from CNF than from more general propositional formulas.

Easy part of proof:

CSAT is in NP: Since CNF formulas are propositional formulas, we have already proved this. CSAT is a restricted case of SAT, so it's obviously no harder.

## NP hardness of CSAT

We can reduce SAT to CSAT in PTIME.

First, “push negations down to the variables” using de Morgan’s laws:

$$\neg(\alpha \wedge \beta) \Rightarrow (\neg\alpha) \vee (\neg\beta) \text{ and } \neg(\alpha \vee \beta) \Rightarrow (\neg\alpha) \wedge (\neg\beta).$$

Also  $\neg\neg p = p$ .

After applying this transformation everywhere we can, the formula will be in *negation normal form*: The only negations will appear in literals.  $((\neg p) \wedge q) \vee \neg r$ .

Every transformation in the reduction to NNF results in *tautologically equivalent* formula – which means that the same truth assignments give the same results for both formulas.

## From NNF to CNF

Assume formula is in NNF now. We have to change the formula so that all of the ANDs are on top of all of the ORs (which are on top of literals).

The tricky case is when there is an OR on top of an AND.

The tricky case is  $\alpha \vee \beta$  when  $\alpha$  and  $\beta$  are AND formulas.

Suppose we have  $(g_1 \wedge g_2 \wedge \dots \wedge g_p) \vee (h_1 \wedge h_2 \wedge \dots \wedge h_q)$

We could use distributive law to distribute  $\vee$  over  $\wedge$ . *But that gives exponential blowup in the formula size!*

**Trick:** add new variables to the clause formulas.

Then create a *fresh variable*  $y$  and convert the formulas to

$(y \vee g_1) \wedge (y \vee g_2) \wedge \dots \wedge (y \vee g_p) \wedge (\neg y \vee h_1) \wedge (\neg y \vee h_2) \wedge \dots \wedge (\neg y \vee h_q)$ .

Intuition: If  $y = T$ , the whole formula is true IFF the  $h$  clauses are true.

## Correctness of CNF conversion

To do a reduction proof, we need to prove two things: (1) the transformation is PTIME, and (2) the SAT formula is satisfiable iff the transformed CSAT formula is satisfiable (the formulas are *equisatisfiable*).

PTIME is obvious from the previous discussion.

For equivalence, note that The second transformation preserves satisfiability, but not logical equivalence.

The proof is by induction on each transformation used to convert the original formula to CNF.

One approach to the induction step is to show that a satisfying assignment for the pre-transform formula can be converted to a satisfying assignment to the post-transform formula, and *vice versa*.

## 3SAT

---

There is an even more restrictive satisfiability problem: CNF formulas where every clause has exactly 3 literals.

If a clause has only one literal, it can be rewritten with new variables

$$(x \vee u \vee v) \wedge (x \vee u \vee \neg v) \wedge (x \vee \neg u \vee v) \wedge (x \vee \neg u \vee \neg v)$$

If a clause has two literals  $(x_1 \vee x_2 \vee v) \wedge (x_1 \vee x_2 \vee \neg v)$

If a clause is too big, it can be broken into smaller clauses using fresh variables:

$x_1 \vee x_2 \vee \dots \vee x_n$  to

$$(x_1 \vee x_2 \vee y_1) \wedge (x_3 \vee \neg y_1 \vee y_2) \wedge \dots \wedge (x_{n-2} \vee \neg y_{n-4} \vee y_{n-3}) \wedge (x_{n-1} \vee x_n \vee \neg y_{n-3})$$

## Correctness of CSAT to 3SAT reduction

Once again, the converted formula is *equisatisfiable* to the original.

Any satisfying assignment to the original clause must have at least one true literal  $x_i$  ( $x_i$  could be a negated variable).

That truth assignment can be converted to a truth assignment for the 3CNF formulas by setting making all  $y$  literals in that clause false. Earlier  $y$  variables should be set to  $T$  and later  $y$  variables to  $F$ . The  $y$  variables will satisfy every 3CNF clause except the one that  $x_i$  satisfies.

In the other direction, note that any satisfying assignment to the 3CNF clauses must have at least one true  $x$  literal, since there are  $m - 2$  clauses and  $m - 3$   $y$  variables, and each  $y$  variable, whether true or false, can only make one clause true – so one  $x$  literal must be true to satisfy the left-over clause. Hence, the satisfying assignment for the 3CNF clauses can be converted to a satisfying assignment for the original clause by restricting it to the  $x$  variables.

What about 2CNF? The reduction doesn't work, and there's a fast algorithm that we won't present here.