

CSE200

Computability and Complexity

Daniele Micciancio

UCSD

Fall 2008

Classifying problems within NP

- Many problems within NP are equivalent to each other

Theorem

SAT, CLIQUE, VC, IS, CIRCUITSAT are all equivalent under polynomial time map reductions

Fact

All problems in P are equivalent under polynomial time reductions

- We suspect not all problems in NP are equivalent

Conjecture

P is different from NP

NP-completeness

Definition

C is NP-complete if C is in NP, and any A in NP reduces to C in polynomial time ($A \leq_P C$)

Complete problems allow to study an entire complexity class while focusing on a single problem.

Theorem

If C is NP-complete, and C is in P , then $P=NP$

Proof.

- 1 $P \subseteq NP$ is clear. Need to prove that $NP \subseteq P$
- 2 Let A be in NP. By NP-completeness, $A \leq_P C$.
- 3 Since C is in P , then A is also in P .



More on completeness

- In the context of P vs NP, NP-completeness is typically used as evidence that a problem is not in P

Corollary

If C is NP-complete, and $P \neq NP$, then C is not in P.

- Completeness is a general concept that can be instantiated differently to study different complexity classes
- Generally, studying $CLASS \subseteq CLASS'$ requires defining $CLASS'$ -complete problems under a notion of reduction that is compatible with $CLASS$:

Theorem

If $A \leq B$ and B is in $CLASS$, then A is in $CLASS$.

- We will see examples where completeness was used to show that $CLASS=CLASS'$.

Completeness: examples

- EXPTIME vs NEXPTIME: we can define NEXPTIME-completeness under deterministic exponential time reductions.
- NP vs PSPACE: we can define PSPACE-completeness under **nondeterministic** polynomial time reductions.
- P vs PSPACE: this time we need to use **deterministic** polynomial time reductions.
- L vs P: need log-space reductions, otherwise all problems in P are trivially P-complete

An NP-complete problem

Definition

WHILE-SAT = { ["P" #n]: P is a WHILE program that accepts some input within n steps }

- WHILE-SAT is in NP:

W(X,Y)

- 1 Parse X as ["P" #n]
 - 2 Run SELF on program P and input Y for n steps.
 - 3 Accept if and only if self-interpreter accept.
- W runs in time $O(n)$, polynomial in the size of first input
 - W correctly decides WHILE-SAT (non-deterministically)

Theorem

WHILE-SAT is NP-complete

Let A be an arbitrary language in NP, and $P(X, Y)$ a WHILE program deciding A nondeterministically in time $c \cdot |X|^c$.

Reduction from A to WHILE-SAT:

- 1 On input X (of size n)
- 2 Compute the size $n = |X|$ (in time $O(n)$)
- 3 Compute $T = c \cdot n^c$ (in time $O(c \cdot n^c)$)
- 4 Build the WHILE program $P_1(X') = P(X, X')$
- 5 Output [P_1 T]

Correctness of the reduction

- The reduction from A to WHILE-SAT runs in polynomial time (most time goes into computing T .)
- If X is in A , then P_1 accepts X within T steps
- If X is not in A , then P_1 does not accept X within T steps
- We can say even more:
 - If X is not in A , then P_1 does not accept X at all
 - We are going to use this later today

What about other NP problems?

- We would like to prove the NP-completeness of CLIQUE, VC, IS, SAT, and many other problems
- All we need is to prove the NP-completeness of SAT

Theorem

For any problem B in NP, if C is NP-complete and $C \leq_P B$, then B is also NP-complete

Proof.

- 1 Let A be any problem in NP. We want to prove that $A \leq_P B$.
- 2 By NP-completeness of C , we have $A \leq_P C$.
- 3 Combining $A \leq_P C \leq_P B$, we obtain $A \leq_P B$



Cook-Levin Theorem: SAT is NP-complete

- Enough to prove the NP-completeness of CIRCUIT-SAT, because $\text{CIRCUIT-SAT} \leq_P \text{SAT}$
- We need to reduce WHILE-SAT to CIRCUIT-SAT:
 - 1 On input ["P" #n], where P is a WHILE program
 - 2 Output a boolean circuit C such that C is satisfiable if and only if P accepts some input within time n .
- Intuition:
 - The inputs to C encode the possible inputs to P
 - The size of C depends on n
 - C emulates P for n steps and accepts if and only if P does

From WHILE-SAT to CIRCUIT-SAT

We give a sequence of reductions starting from WHILE-SAT to problems of type XXX-SAT, for simpler and simpler programming models XXX:

- WHILE-SAT
- WHILE-SAT with binary input:
- JUMP-SAT: lower level programming language with explicit memory, pointers and conditional jump.
- CPU-SAT: down to the actual hardware
- CIRCUIT-SAT

From lists to binary strings

- WHILE-SAT: set of all $[\text{"P"} \#n]$ such that P accept some input within time n .
- We never imposed any bound on the size of the input accepted by P !
- Notice: P can access at most n sublists of X .
- If P accepts some input, then it accepts some input of size at most n
- Modify P into a program P_1 that first parses its binary input $[001010001111]$ as a parenthesized list $[[]][[]][[]]$, and then runs P
- Output $[\text{"P}_1" \#(c n)]$, for an appropriate constant c .

Consider a programming language with instructions

$$\begin{aligned} P &::= C_1; \dots; C_n \\ C &::= V = M[V] \\ &| M[V] = V \\ &| V = n \\ &| INC(V) \\ &| JUMPZ(V, n) \end{aligned}$$

Time is over ... to be continued in the next lecture.