Computational Complexity Theory

Lecture 2: Class NP, Reductions, NP-completeness

Indian Institute of Science

Complexity classes
P and NP

Recap: Decision Problems

- In the initial part of this course, we'll focus primarily on decision problems.

- Decision problems can be naturally identified with boolean functions, i.e. functions from $\{0,1\}^*$ to $\{0,1\}$.

- Boolean functions can be naturally identified with sets of $\{0,1\}$ strings, also called languages.

Recap: Decision Problems

Decision problems $\Leftrightarrow$ Boolean functions $\Leftrightarrow$ Languages

- Definition. We say a TM M *decides* a language $L \subseteq \{0,1\}^*$ if M computes f_L, where $f_L(x) = 1$ if and only if $x \in L$.

Recap: Complexity Class P

- Let $T: N \to N$ be some function.

- Definition: A language L is in $DTIME(T(n))$ if there's a TM that decides L in time $O(T(n))$.

- Defintion: Class $P = \bigcup_{c \geq 0} DTIME(n^c)$.

Deterministic polynomial-time

Complexity Class P : Examples

- Cycle detection
- Solvabililty of a system of linear equations
- Perfect matching
- Primality testing *(AKS test 2002)*
  - Check if a number is prime

Polynomial time Turing Machines

- Definition. A TM M is a *polynimial time* TM if there's a polynomial function $q: N \to N$ such that for every input $x \in \{0,1\}^*$, M halts within $q(|x|)$ steps.

Polynomial function. $q(n) = n^c$ for some constant c

Class (functional) P

- What if a problem is not a decision problem? Like the task of adding two integers.

Class (functional) P

- What if a problem is not a decision problem? Like the task of adding two integers.
- One way is to focus on the i-th bit of the output and make it a decision problem.

(Is the i-th bit, on input x, 1?)

Class (functional) P

- What if a problem is not a decision problem? Like the task of adding two integers.
- One way is to focus on the i-th bit of the output and make it a decision problem.
- Alternatively, we define a class called functional P.

Class (functional) P

- What if a problem is not a decision problem? Like the task of adding two integers.
- One way is to focus on the i-th bit of the output and make it a decision problem.

- We say that a problem or a function $f: \{0,1\}^* \rightarrow \{0,1\}^*$ is in FP (functional P) if there's a polynomial-time TM that computes f.

Complexity Class FP : Examples

- Greatest Common Divisor *(Euclid ~300 BC)*
  - Given two integers a and b, find their gcd.

Complexity Class FP : Examples

- Greatest Common Divisor

- Counting paths in a DAG *(homework)*
  - Find the number of paths between two vertices in a directed
    acyclic graph.

Complexity Class FP : Examples

- Greatest Common Divisor
- Counting paths in a DAG
- Maximum matching *(Edmonds 1965)*
  - Find a maximum matching in a given graph

Complexity Class FP : Examples

- Greatest Common Divisor
- Counting paths in a DAG
- Maximum matching
- Linear Programming *(Khachiyan 1979, Karmarkar 1984)*
  - Optimize a linear objective function subject to linear (in)equality constraints

Complexity Class NP

- Solving a problem is generally *harder* than verifying a given solution to the problem.

Complexity Class NP

- Solving a problem is generally *harder* than verifying a given solution to the problem.

- Class NP captures the set of decision problems whose solutions are *efficiently verifiable*.

Complexity Class NP

- Solving a problem is generally *harder* than verifying a given solution to the problem.

- Class NP captures the set of decision problems whose solutions are *efficiently verifiable*.

Nondeterministic polynomial-time

Complexity Class NP

- Definition. A language $L \subseteq \{0,1\}^*$ is in NP if there's a polynomial function $p: N \rightarrow N$ and a polynomial time TM M (called the *verifier*) such that for every x,

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

Complexity Class NP

- Definition. A language $L \subseteq \{0,1\}^*$ is in NP if there's a polynomial function $p: N \rightarrow N$ and a polynomial time TM M (called the *verifier*) such that for every x,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

u is called a *certificate or witness* for x (w.r.t L and M) if $x \in L$

Complexity Class NP

- Definition. A language $L \subseteq \{0,1\}^*$ is in NP if there's a polynomial function $p: N \rightarrow N$ and a polynomial time TM M (called the <u>*verifier*</u>) such that for every x,

$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

- It follows that verifier M <u>cannot be fooled</u>!

Complexity Class NP

- Definition. A language $L \subseteq \{0,1\}^*$ is in NP if there's a polynomial function $p: N \rightarrow N$ and a polynomial time TM M (called the *verifier*) such that for every x,

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

- Class NP contains those problems (languages) which have such efficient verifiers.

Class NP : Examples

- Vertex cover
  - Given a graph G and an integer k, check if G has a vertex cover of size k.

Class NP : Examples

- Vertex cover

- 0/1 integer programming
  - Given a system of linear (in)equalities with integer coefficients, check if there's a 0-1 assignment to the variables that satisfy all the (in)equalities.

Class NP : Examples

- Vertex cover
- 0/1 integer programming
- Integer factoring
  - Given 2 numbers n and U, check if n has a nontrivial factor less than equal to U.

Class NP : Examples

- Vertex cover
- 0/1 integer programming
- Integer factoring
- Graph isomorphism
  - Given 2 graphs, check if they are isomorphic

Is P = NP ?

- Obviously, $P \subseteq NP$.

- Whether or not $P = NP$ is an outstanding open question in mathematics and TCS!

Is P = NP ?

- Obviously, $P \subseteq NP$.
- Whether or not $P = NP$ is an outstanding open question in mathematics and TCS!
- Solving a problem does seem harder than verifying its solution, so most people believe that $P \neq NP$.

Is P = NP ?

- Obviously, $P \subseteq NP$.
- Whether or not $P = NP$ is an outstanding open question in mathematics and TCS!
- $P = NP$ has many weird consequences, and if true, will pose a serious threat to secure and efficient cryptography.

Is P = NP ?

- Obviously, $P \subseteq NP$.
- Whether or not $P = NP$ is an outstanding open question in mathematics and TCS!
- Mathematics has gained much from attempts to prove such negative statements—Galois theory, Godel's incompleteness, Fermat's Last Theorem, Turing's undecidability, Continuum hypothesis etc.

Is P = NP ?

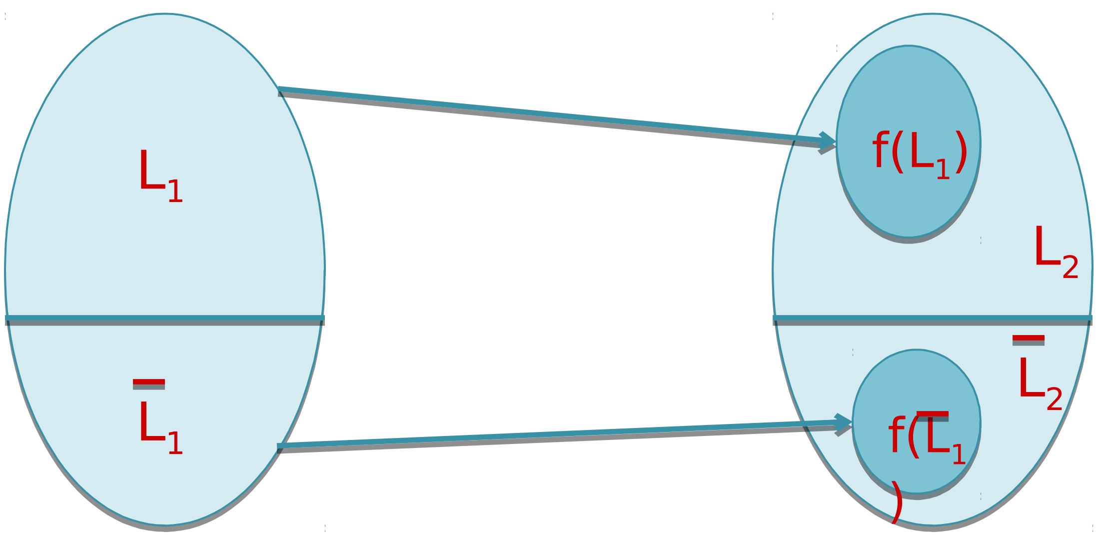
- Obviously, $P \subseteq NP$.
- Whether or not $P = NP$ is an outstanding open question in mathematics and TCS!
- Complexity theory has several of such intriguing unsolved questions.

Karp reductions

Polynomial time reduction

- Definition. We say a language $L_1 \subseteq \{0,1\}^*$ is <u>*polynomial time (Karp) reducible*</u> to a language $L_2 \subseteq \{0,1\}^*$ if there's a polynomial time computable function f s.t.

$$x \in L_1 \Leftrightarrow f(x) \in L_2$$



Polynomial time reduction

- Definition. We say a language $L_1 \subseteq \{0,1\}^*$ is <u>*polynomial time (Karp) reducible*</u> to a language $L_2 \subseteq \{0,1\}^*$ if there's a polynomial time computable function f s.t.

$$x \in L_1 \iff f(x) \in L_2$$

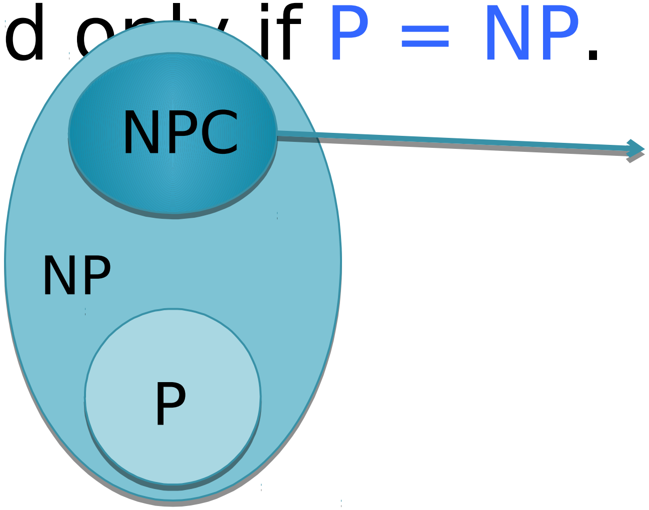
- Notation. $L_1 \leq_p L_2$

- Observe. If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ then $L_1 \leq_p L_3$.

NP-completeness

- Definition. A language L' is *NP-hard* if for every L in NP, $L \leq_p L'$. Further, L' is *NP-complete* if L' is in NP and is NP-hard.

- Observe. If L' is NP-hard and L' is in P then P = NP. If L' is NP-complete then L' in P if and only if P = NP.



Hardest problems inside NP in the sense that if one NPC problem is in P then all problems in NP is in P.

NP-completeness

- Definition. A language L' is *NP-hard* if for every L in NP, $L \le_p L'$. Further, L' is *NP-complete* if L' is in NP and is NP-hard.

- Observe. If L' is NP-hard and L' is in P then P = NP. If L' is NP-complete then L' in P if and only if P = NP.

- Exercise. Let $L_1 \subseteq \{0,1\}^*$ be any language and L_2 be a language in NP. If $L_1 \le_p L_2$ then L_1 is also in NP.

Few words on reductions

- As to how we define a reduction from one language to the other (or one function to the other) is usually guided by a *<u>question on</u> whether two <u>complexity classes</u> are different or identical*.

- For polynomial time reductions, the question is whether P equals NP.

- Reductions help us define *complete problems* (the ‘hardest’ problems in a class) which in turn help us compare the complexity classes under consideration.

Class P and NP : Examples

- Vertex cover (NP-complete)
- 0/1 integer programming (NP-complete)
- Integer factoring (unlikely to be NP-complete)
- Graph isomorphism (Quasi-P)
- Primality testing (P)
- Linear programming (P)

How to show existence of an NPC problem?

- Let L' = { $(\alpha, x, 1^m, 1^t)$: there exists a $u \in \{0,1\}^m$ s.t. M_α accepts (x, u) in t steps }

- Observation. L' is NP-complete.

- The language L' involves Turing machine in its definition. Next, we'll see an example of an NP-complete problem that is arguably more natural.

A natural NP-complete problem

- Definition. A ***boolean formula*** on variables $x_1, \ldots, x_n$ consists of AND, OR and NOT operations.

  e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- Definition. A boolean formula ϕ is ***satisfiable*** if there's a $\{0,1\}$-assignment to its variables that makes ϕ evaluate to 1.

A natural NP-complete problem

- Definition. A boolean formula is in *Conjunctive Normal Form* (*CNF*) if it is an AND of OR of literals.

e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

clauses

A natural NP-complete problem

- Definition. A boolean formula is in <u>*Conjunctive Normal Form*</u> (*CNF*) if it is an AND of OR of literals.

e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

literals

A natural NP-complete problem

- Definition. A boolean formula is in *Conjunctive Normal Form* (*CNF*) if it is an AND of OR of literals.

  e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- Definition. Let SAT be the language consisting of all *satisfiable CNF formulae.*

A natural NP-complete problem

- Definition. A boolean formula is in <u>*Conjunctive Normal Form*</u> (*CNF*) if it is an AND of OR of literals.

  e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- Definition. Let SAT be the language consisting of all *satisfiable CNF formulae.*

- Theorem. *(Cook-Levin)* SAT is NP-complete.

A natural NP-complete problem

- Definition. A boolean formula is in <u>*Conjunctive Normal Form*</u> (*CNF*) if it is an AND of OR of literals.

  e.g. $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- Definition. Let SAT be the language consisting of all *satisfiable CNF formulae.*

- Theorem. *(Cook-Levin)* SAT is NP-complete.

  Easy to see that SAT is in NP.

  Need to show that SAT is NP-hard.

Proof of Cook-Levin Theorem

Cook-Levin theorem: Proof

- Main idea: Computation is *local*; i.e. every step of computation *looks at* and *changes* only constantly many bits; and this step can be implemented by a small CNF formula.

Cook-Levin theorem: Proof

- Main idea: Computation is *local*; i.e. every step of computation *looks at* and *changes* only constantly many bits; and this step can be implemented by a small CNF formula.

- Let $L \in NP$. We intend to come up with a polynomial time computable function $f: x \mapsto \phi_x$ s.t.,
  - $x \in L \iff \phi_x \in SAT$

Cook-Levin theorem: Proof

- Main idea: Computation is *local*; i.e. every step of computation *looks at* and *changes* only constantly many bits; and this step can be implemented by a small CNF formula.

- Let L $\in$ NP. We intend to come up with a polynomial time computable function f: $x \longmapsto \phi_x$ s.t.,
  - $x \in L \iff \phi_x \in SAT$

  - <u>Notation:</u> $|\phi_x| :=$ size of ϕ_x

    $=$ number of $\vee$ or $\wedge$ in ϕ_x

Cook-Levin theorem: Proof

- Language L has a poly-time verifier M such that

$x \in L \iff \exists u \in \{0,1\}^{p(|x|)}$ s.t. $M(x, u) = 1$

Cook-Levin theorem: Proof

- Language L has a poly-time verifier M such that

  $x \in L \iff \exists u \in \{0,1\}^{p(|x|)}$ s.t. $M(x, u) = 1$

- Idea: Capture the computation of $M(x, ..)$ by a CNF ϕ_x such that

  $\exists u \in \{0,1\}^{p(|x|)}$ s.t. $M(x, u) = 1 \iff \phi_x$ is satisfiable

Cook-Levin theorem: Proof

- Language L has a poly-time verifier M such that

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

- Idea: Capture the computation of $M(x, ..)$ by a CNF ϕ_x such that

$$\exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1 \Leftrightarrow \phi_x \text{ is satisfiable}$$

- For any fixed x, $M(x, ..)$ is a deterministic TM that takes u as input and runs in time polynomial in $|u|$.

Cook-Levin theorem: Proof

- Main Theorem. Let N be a deterministic TM that runs in time T(n) on every input u of length n, and outputs 0/1. Then,

Cook-Levin theorem: Proof

- Main Theorem. Let N be a deterministic TM that runs in time $T(n)$ on every input u of length n, and outputs $0/1$. Then,
    1. There's a CNF ϕ of size *poly*$(T(n))$ such that $\phi(u,$ *"additional variables"*$)$ is satisfiable if and only if $N(u) = 1$.

Cook-Levin theorem: Proof

- Main Theorem. Let N be a deterministic TM that runs in time T(n) on every input u of length n, and outputs 0/1. Then,
  1. There's a CNF ϕ of size *poly*(T(n)) such that ϕ(u, *"additional variables"*) is satisfiable if and only if N(u) =1.
  2. ϕ is computable in time *poly(T(n))*.

Cook-Levin theorem: Proof

- Main Theorem. Let N be a deterministic TM that runs in time T(n) on every input u of length n, and outputs 0/1. Then,
  1. There's a CNF ϕ of size *poly*(T(n)) such that ϕ(u, *"additional variables"*) is satisfiable if and only if N(u) =1.
  2. ϕ is computable in time *poly(T(n))*.

- Cook-Levin theorem follows from above!