CMSC5706 Topics in Theoretical Computer Science

Week 5: NP-completeness

Instructor: Shengyu Zhang

Tractable

- While we have introduced many problems with polynomial-time algorithms…
- …not all problems enjoy fast computation.

- Among those “hard” problems, an important class is **NP**.

P, NP

- **P**: Decision problems solvable in deterministic polynomial time
- **NP**: two definitions
  - Decision problems solvable in nondeterministic polynomial time.
  - Decision problems (whose valid instances are) checkable in deterministic polynomial time
- Let's use the second definition.
- Recall: A language L is just a subset of $\{0,1\}^*$, the set of all strings of bits.
  - $\{0,1\}^* = \bigcup_{n\geq 0}\{0,1\}^n$.

Formal definition of **NP**

- Def. A language $L \subseteq \{0,1\}^*$ is in **NP** if there exists a polynomial $p: \mathbb{N} \to \mathbb{N}$ and a polynomial-time Turing machine M such that for every $x \in \{0,1\}^*$,

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

- M: the verifier for L.
- For $x \in L$, the u is called a certificate for x.
- So **NP** contains those problems easy to check
  - given the certificate.

SAT and k-SAT

- SAT formula: AND of m clauses
  - n variables (taking values 0 and 1)
  - a literal: a variable x_i or its negation $\overline{x_i}$
  - m clauses, each being OR of some literals.
- SAT Problem: Is there an assignment of variables s.t. the formula evaluates to 1?
- k-SAT: same as SAT but each clause has at most k literals.
- SAT and k-SAT are in **NP**.
- Given any assignment, it's easy to check whether it satisfies all clauses.

Examples of **NP** problems

- Factoring: factor a given number n.
- Decision version: Given (n, k), decide whether n has a factor less than k.

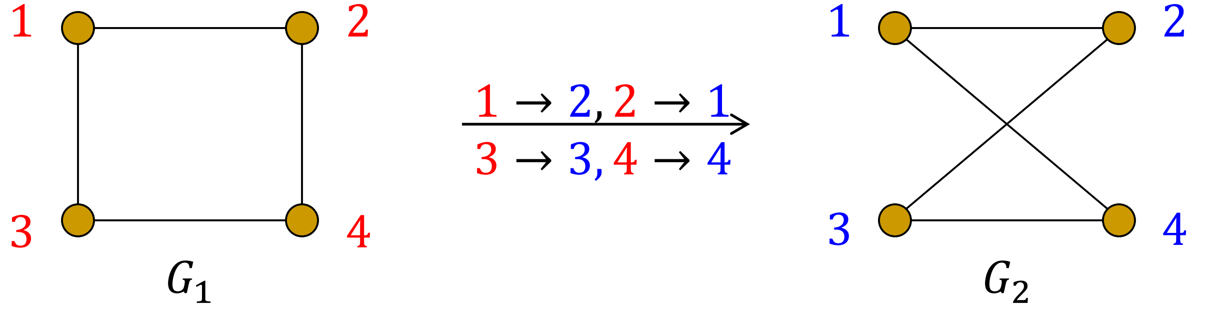
- Factoring is in **NP**: For any candidate factor $m \leq k$, it's easy to check whether $m|n$.

Examples of **NP** problems

- TSP (travelling salesperson): On a weighted graph, find a closed cycle visiting each vertex exactly once, with the total weight on the path no more than k.

- Easy to check: Given a cycle, easy to calculate the total weight.

- Graph Isomorphism: Given two graphs G_1 and G_2, decide whether we can permute vertices of G_1 to get G_2.



- Easy to check: For any given permutation, easy to permute G_1 according to it and then compare to G_2.

Question of **P** vs. **NP**

- Is **P** = **NP**?
- The most famous (and notoriously hard) question in computer science.
  - Staggering philosophical and practical implications
  - Withstood a great deal of attacks
- Clay Mathematics Institute recognized it as one of seven great mathematical challenges of the millennium. US$1M.
  - Want to get rich (and famous)? Here is a “simple” way!

The **P** vs. **NP** question: intuition

- Is producing a solution essentially harder than checking a solution?
  - Coming up with a proof vs. verifying a proof.
  - Composing a song vs. appreciating a song.
  - Cooking good food vs. recognizing good food
  - …

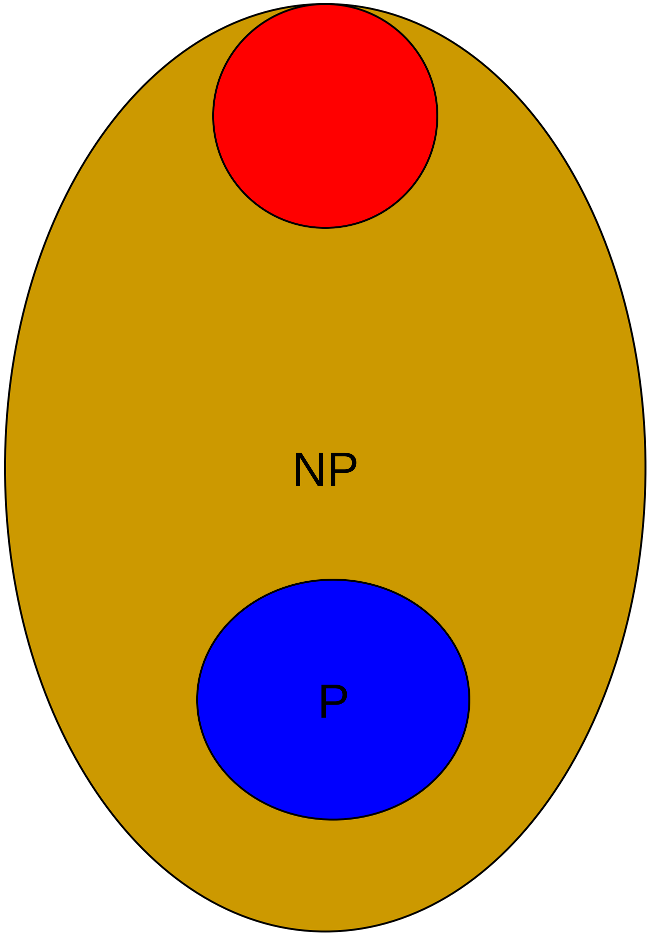
What if **P** = **NP**?

- The world becomes a Utopia.
  - Mathematicians are replaced by efficient theorem-discovering machines.
  - It becomes easy to come up with the simplest theory to explain known data
  - …
- But at the same time,
  - Many cryptosystems are insecure.

Completeness

- [Cook-Levin] There is a class of **NP** problems, such that

  solve any of them in polynomial time,
  ⇒ solve all **NP** problems in polynomial time.



Reduction and completeness

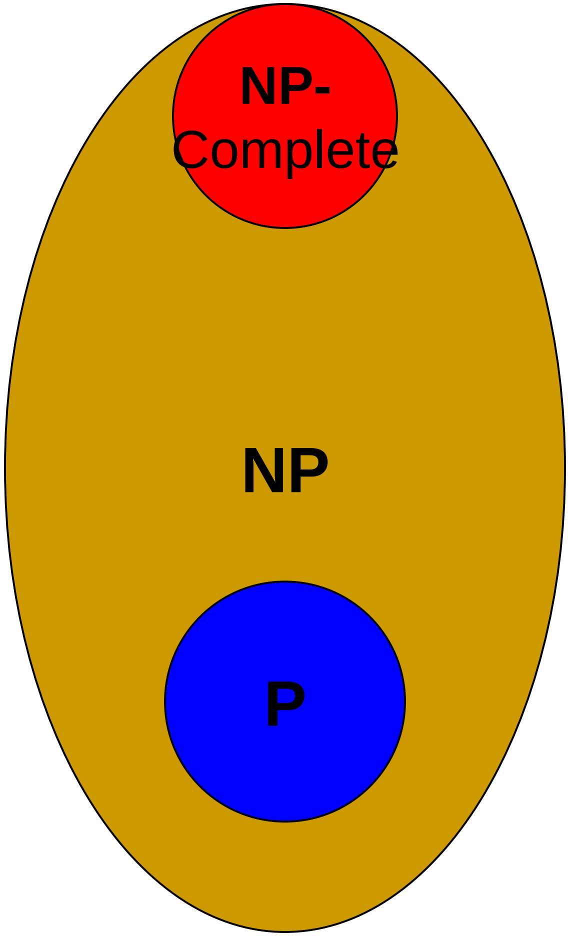
- Decision problem for language A is reducible to that for language B in time t if $\exists f: Domain(A) \rightarrow Domain(B)$ s.t. $\forall$ input instance x for A,
  1. $x \in A \Leftrightarrow f(x) \in B$, and
  2. one can compute $f(x)$ in time $t(|x|)$
- Thus to solve A, it is enough to solve B.
  - First compute $f(x)$
  - Run algorithm for B on $f(x)$.
  - If the algorithm outputs $f(x) \in B$, then output $x \in A$.

NP-completeness

- **NP**-completeness: A language L is **NP**-complete if
  - $L \in \mathbf{NP}$
  - $\forall L' \in \mathbf{NP}$, L' is reducible to L in polynomial time.
- Such problems L are the hardest in **NP**.
- Once you can solve L, you can solve any other problem in **NP**.
- **NP**-hard: any **NP** language can reduce to it.
  - i.e. satisfies 2^{nd} condition in **NP**-completeness def.

Completeness

- The hardest problems in **NP**.
- Cook-Levin: SAT.
- Karp: 21 other problems such as TSP are also **NP**-complete
- Later: thousands of **NP**-complete problems from various sciences.



Meanings of **NP**-completeness

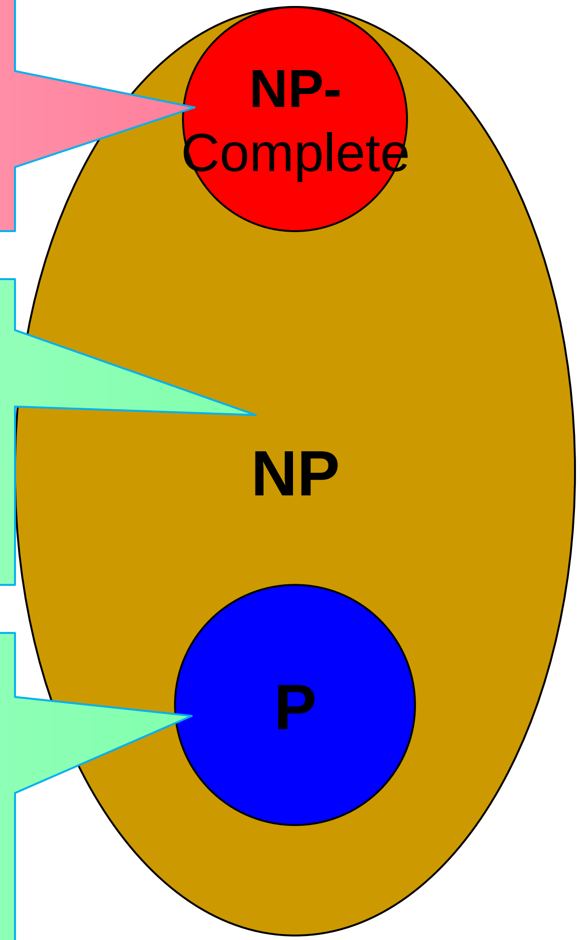
- Reduce the number of questions without increasing the number of answers.
- Huge impacts on almost all other sciences such as physics, chemistry, biology, …
  - Now given a computational problem in **NP**, the first step is usually to see whether it's in **P** or NPC.
- "The biggest export of Theoretical Computer Science."

- SAT
- Clique
- Subset Sum
- TSP
- Vertex Cover
- Integer Programming

Not known to be in **P** or **NP**-complete:

- Factoring
- Graph Isomorphism
- Nash Equilibrium
- Local Search

- Shortest Path
- MST
- Maximum Flow
- Maximum Matching
- PRIMES
- Linear Programming



The 1st **NP**-complete problem: 3-SAT

- Any **NP** problem can be verified in polynomial time, by definition.
- Turn the verification algorithm into a formula which checks every step of computation.
- Note that in either circuit definition or Turing machine definition, computation is local.
  - The change of configuration is only at several adjacent locations.

- Thus the verification can be encoded into a sequence of local consistency checks.
- The number of clauses is polynomial
  - The verification algorithm is of polynomial time.
  - Polynomial time also implies polynomial space.
- This shows that SAT is **NP**-complete.
- It turns out that any SAT can be further reduced to 3-SAT problem.

NP-complete problem 1: Clique

- Clique: Given a graph G and a number k, decide whether G has a clique of size $\geq k$.
  - Clique: a complete subgraph.
- Fact: Clique is in **NP**.
- Theorem: If one can solve Clique in polynomial time, then one can also solve 3SAT in polynomial time.
  - So Clique is at least as hard as 3-SAT.
- Corollary: Clique is **NP**-complete.

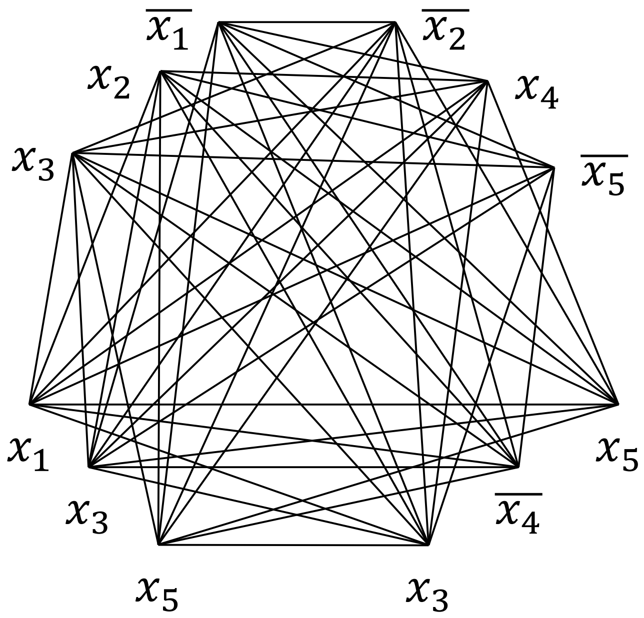
Approach: reduction

- Given a 3-SAT formula $\varphi = C_1 \wedge \cdots \wedge C_k$, we construct a graph G s.t.
  - if φ is satisfiable, then G has a clique of size k.
  - if φ is unsatisfiable, then G has no clique of size $\geq k$.
  - Note: k is the number of clauses of φ.

- If you can solve the Clique problem, then you can also solve the 3-SAT problem.

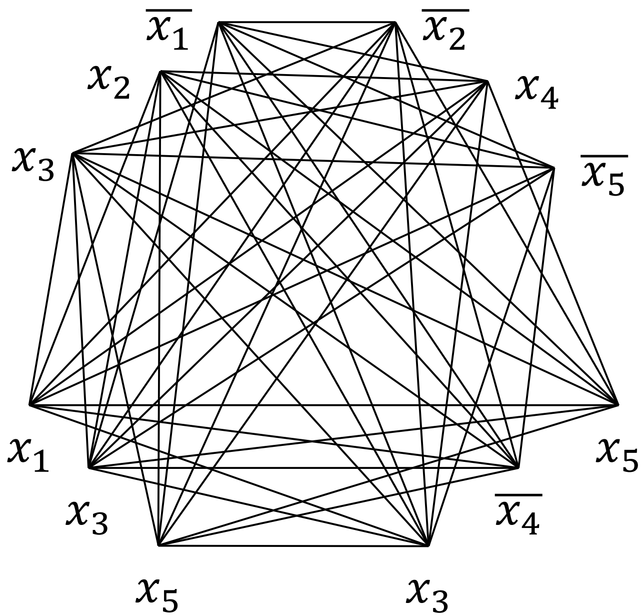
Construction

- Put each literal appearing in the formula as a vertex.
  - Literal: x_i and $\overline{x_i}$
  - e.g. $\varphi = (\overline{x_1} \lor x_2 \lor x_3) \land (\overline{x_2} \lor x_4 \lor \overline{x_5}) \land (x_1 \lor x_3 \lor x_5) \land (x_3 \lor \overline{x_4} \lor x_5)$
- Literals from the same clause are not connected.
- Two literals from different clauses are connected if they are not the negation of each other.



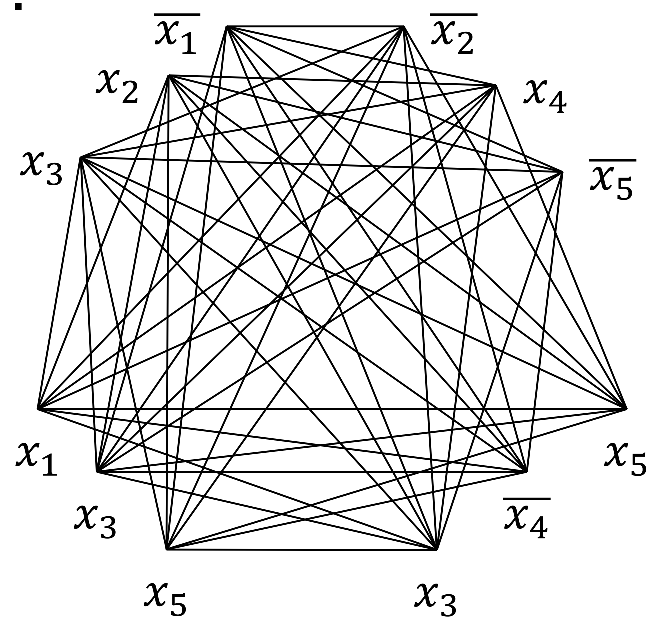
φ is satisfied $\Rightarrow G$ has a k-clique

- If φ is satisfied,
- then there is a satisfying assignment $x_1 \ldots x_n$ s.t. each clause has at least one literal being 1.
  - E.g. $x = 00111$, then pick $\overline{x_1}, x_4, x_3, x_5$
- And those literals (one from each clause) are consistent.
  - Because they all evaluate to 1
- So the subgraph with these vertices is complete. --- A clique of size k.



G has a k-clique ⇒ φ is satisfied

- If the graph has a clique of size k:
- It must be one vertex from each clause.
  - Vertices from the same clause don't connect.
- And these literals are consistent.
  - Otherwise they don't all connect.
- So we can pick the assignment by these vertices. It satisfies all clauses by satisfying at least one vertex in each clause.



NP-complete problem 2: Vertex Cover

- Vertex Cover: Given a graph G and a number k, decide whether G has a vertex cover of size $\leq k$.
  - $V' \subseteq V$ is a vertex cover if all edges in G are "touched" by vertices from V'.

- Vertex Cover is in **NP**
  - Given a candidate subset $S \subseteq V$, it is easy to check whether "$|S| \leq k$ and S touches whole E".

NP-complete

- Vertex Cover is **NP**-complete.
- Reducing Clique to Vertex Cover.
- For any graph G, the complement of G is $\overline{G}$.
  - If $G = (V, E)$, then $\overline{G} = (V, \overline{E})$.
- Theorem. G has a k-clique
  $\Leftrightarrow \overline{G}$ has a vertex cover of size $n - k$.
- Given this theorem, Clique can be reduced to Vertex Cover.
- So Vertex Cover is **NP**-complete.

Proof of the theorem

- G has a k-clique

$\Leftrightarrow \exists V' \subseteq V, |V'| = k, V'$ is a clique in G

$\Leftrightarrow \exists V' \subseteq V, |V'| = k, V'$ is independent set in $\overline{G}$

  - independent set: $\forall$ two vertices $u, v \in V'$ are not connected in $\overline{G}$.

$\Leftrightarrow \exists V' \subseteq V, |V'| = k, V \backslash V'$ is a vertex cover of $\overline{G}$

$\Leftrightarrow \exists V'' \subseteq V, |V''| = n - k, V''$ is a vertex cover of $\overline{G}$

A related problem: Independent Set

- Independent Set: Decide whether a given graph has an independent set of size at least k.

- The above argument shows that the Independent Set problem is also **NP**-Complete.

Another bonus: Set Cover

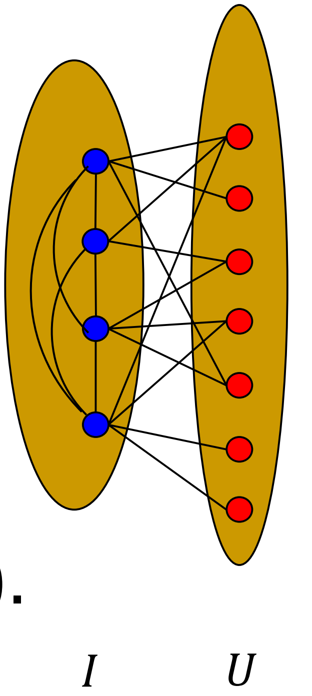
- Set Cover: Given a number k, ground set U and a collection of subsets $\{S_1, \ldots, S_m\}$ of U, decide whether $\exists$ k subsets S_i whose union covers U.
- Vertex Cover is just Set Cover with the promise that each element is covered by exactly 2 sets.
  - Ground set U: edges.
  - Sets S_v: edges incident to v for each $v \in V$.
- Thus Vertex Cover is **NP**-complete
  $\Rightarrow$ Set Cover is **NP**-complete.
  - Set Cover is clearly in **NP**.

NP-complete problem 3: Dominating Set

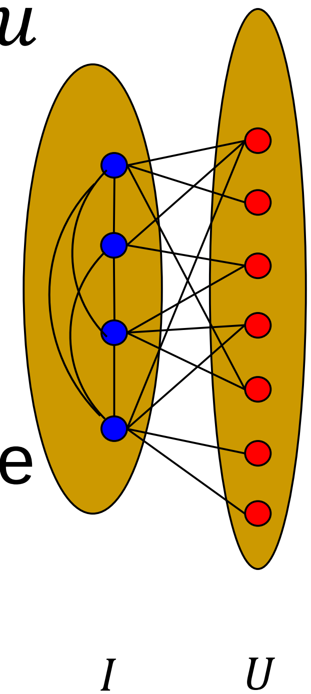
- In a graph $G = (V, E)$, a dominating set is a set $S \subseteq V$ s.t. $\forall v \in V$, either $v \in S$ or v has a neighbor in S.
  - Namely, S and S's neighbors cover the entire V.
- *Dominating Set* problem: Given a graph $G = (V, E)$ and an integer K, decide whether G contains a dominating set of size at most K.
- Theorem. Dominating Set is NP-complete.
  - Reduction from Set Cover.

Reduction

- Given an instance of Set Cover
  - $(k, U, \{S_i : i \in I\})$
- construct an instance of Dominating Set: (k, G),
  - $G = (I \cup U, E)$
  - $E = \{(i, u) : u \in S_i\} \cup \{(i, j) : i, j \in I\}$
- If $\exists C \subseteq I$ s.t. $\bigcup_{i \in C} S_i = U$, $|C| \leq k$:
- C is a dominating set of G (with $|C| \leq k$).
  - $N(C)$ covers U since $\bigcup_{i \in C} S_i = U$,
  - $N(C)$ covers I since $(i, j) \in E$, $\forall i, j \in I$



- If $\exists D \subseteq I \cup U$ s.t. $D \cup N(D) = I \cup U$, $|D| \leq k$: For any $u \in U \cap D$, replace u by an $i \in N(u)$.
  - The resulting set $J \subseteq I$ is of size $\leq k$.
  - For each $u \in U$, if it was in D, it's now covered by i.
  - If it wasn't in D, then it's in $N(j)$ for some $j \in D$. It's still covered by $N(j)$.
- Therefore $(k, U, \{S_i : i \in I\})$ is Yes for Set Cover iff (k, G) is Yes for Dominating Set.



NP-complete problem 4: Integer Programming (IP)

- Any 3-SAT formula can be expressed by integer programming.
- Consider a clause, for example, $\overline{x_1} \vee x_2 \vee x_3$
- $\overline{x_1} \vee x_2 \vee x_3 = 1, \qquad x_1, x_2, x_3 \in \{0,1\}$
  $\Leftrightarrow (1 - x_1) + x_2 + x_3 \geq 1, \qquad x_1, x_2, x_3 \in \{0,1\}$
- Indeed, when all $x_1, x_2, x_3 \in \{0,1\}$,

$$\overline{x_1} \vee x_2 \vee x_3 = 0$$
$$\Leftrightarrow x_1 = 1, x_2 = 0, x_3 = 0$$
$$\Leftrightarrow (1 - x_1) + x_2 + x_3 = 0$$

- So the satisfiability problem on a 3SAT formula like $(\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_2} \vee x_4 \vee \overline{x_5}) \wedge (x_1 \vee x_3 \vee x_5) \wedge (x_3 \vee \overline{x_4} \vee x_5)$ is reduced to the feasibility problem of the following IP:
- $(1 - x_1) + x_2 + x_3 \geq 1,$
  $(1 - x_2) + x_4 + (1 - x_5) \geq 1,$
  $x_1 + x_3 + x_5 \geq 1,$
  $x_3 + (1 - x_4) + x_5 \geq 1,$
  $x_1, x_2, x_3, x_4, x_5 \in \{0,1\}$
- So if one can solve IP efficiently, then one can also solve 3SAT efficiently.

Summary

- **NP**: problems that can be verified in polynomial time.
- An important concept: **NP**-complete.
  - The hardest problems in **NP**.
- Whether **P**=**NP** is the biggest open question in computer science.
- Proofs of **NP**-completeness usually use reduction.

Randomized Algorithms

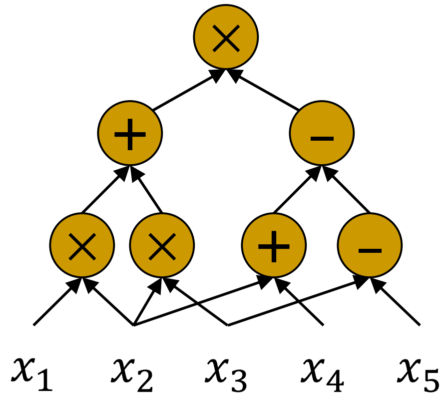
- How to deal with problems harder than **P**?
- One approach: use randomness in our algorithms.

Motivation

- Why randomness?
  - Faster.
  - Simpler.
- Price: a nonzero error probability
  - Usually can be controlled to arbitrarily small.
  - Repeating k times drops the error probability to c^{-k} for some constant $c > 1$.
    - Second part of the lecture.

Polynomial Identity Testing

- Given two polynomials p_1 and p_2 (by arithmetic circuit), decide whether they are equal.
- Arithmetic circuit:

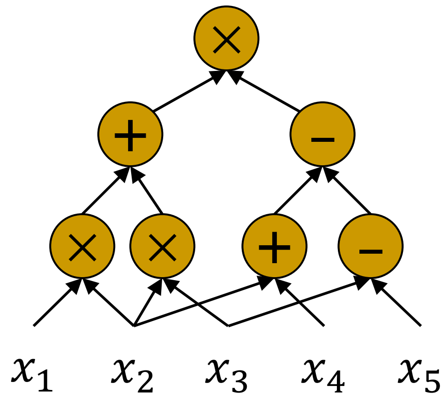


polynomial computed:

$(x_1x_2 + x_2x_3)((x_2 + x_4) - (x_3 - x_5))$

- Question: Given two such circuits, do they compute the same polynomial?

Naïve algorithm?



polynomial computed:

$$(x_1x_2 + x_2x_3)((x_2 + x_4) - (x_3 - x_5))$$

- We can expand the two polynomials and compare their coefficients
- But it takes too much time.
  - Size of the expansion can be exponential in the number of gates.
  - Can you give such an example?

Key idea

- *Schwartz-Zippel Lemma*. If $p(x_1, \ldots, x_n)$ is a polynomial of total degree d over a field $\mathbb{F}$, then $\forall S \subseteq \mathbb{F}$,

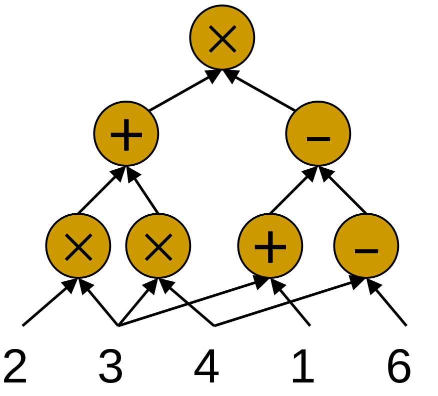
$$\Pr_{a_i \leftarrow_R S}[p(a_1, \ldots, a_n) = 0] \leq \frac{d}{|S|}.$$

  - *total degree of a monomial* $x_1^2 x_2^3 x_5^7$: $2 + 3 + 7 = 12$
  - *total degree of a polynomial*: the max total degree of its monomials.
  - $a_i \leftarrow_R S$: pick each a_i from S uniformly at random. (Different a_i's are picked independently.)

Few other observations

- A polynomial is easy to evaluate on any point by following the circuit.

- The (formal) degree of an polynomial is easy to obtain.



Randomized Algorithm

On input polynomials p_1 and p_2:

- $d = \max\{\deg(p_1), \deg(p_2)\}$
- $a_1, \ldots, a_n \leftarrow_R \{1,2, \ldots, 100d\}$
- Evaluate $p_1(a_1, \ldots, a_n)$ and $p_2(a_1, \ldots, a_n)$ by running the circuits on $(a_1, \ldots, a_n)$.
- **if** $p_1(a_1, \ldots, a_n) = p_2(a_1, \ldots, a_n)$,

  output "$p_1 = p_2$".

  else

  output "$p_1 \neq p_2$".

Correctness

- If $p_1 = p_2$, then $p_1(a_1, \ldots, a_n) = p_2(a_1, \ldots, a_n)$ is always true, so the algorithm outputs $p_1 = p_2$.
- If $p_1 \neq p_2$: Let $p = p_1 - p_2$. Recall that
  - we picked $a_1, \ldots, a_n \leftarrow_R S \stackrel{\text{def}}{=} \{1,2, \ldots, 100d\}$,
  - Lemma. $\Pr_{a_i \leftarrow_R S}[p(a_1, \ldots, a_n) = 0] \leq \frac{d}{|S|}$.
  - So $p_1(a_1, \ldots, a_n) = p_2(a_1, \ldots, a_n)$ w/ prob. only 0.01.
  - The algorithm outputs $p_1 \neq p_2$ w/ prob. ≥ 0.99.

Catch

- One catch is that if the degree d is very large, then the evaluated value can also be huge.
  - Thus unaffordable to write down.
- Fortunately, a simple trick called "fingerprint" handles this.
  - Use a little bit of algebra; omitted here.

- Questions for the algorithm?