Computational Complexity Theory

Lecture 5: More NP-complete problems; Decision vs. Search

Department of Computer Science,
Indian Institute of Science

Recap: 3SAT is NP-complete

- Definition. A CNF is a called a k-CNF if every clause has at most k literals.

  e.g. a 2-CNF $\phi = (x_1 \lor x_2) \land (x_3 \lor \neg x_2)$

- Definition. k-SAT is the language consisting of all *satisfiable k-CNFs.*

- Theorem. *(Cook-Levin)* 3-SAT is NP-complete.

Recap: More NP complete problems

- Independent Set
- Clique
- Vertex cover
- 0/1 integer programming
- Max-Cut (NP-hard)

Karp 1972

- 3-coloring planar graphs *Stockmeyer 1973*
- 2-Diophantine solvability *Adleman & Manders 1975*

Ref: Garey & Johnson, "*Computers and Intractability*" 1979

Recap: NPC problems from NT

- SqRootMod: Given natural numbers a, b and c, check if there exists a natural number $x \le c$ such that

$$x^2 = a \pmod{b}.$$

- Theorem: SqRootMod is NP-complete.

Manders & Adleman 1976

Recap: NPC problems from NT

- Variant_IntFact : Given natural numbers L, U and N, check if there exists a **natural number** $d \in [L, U]$ such that d divides N.

- Claim: Variant_IntFact is NP-hard under *randomized poly-time reduction*.

- Reference: *https://cstheory.stackexchange.com/questions/4769/an-np-complete-variant-of-factoring/4785*

Recap: A peculiar NP problem

- Minimum Circuit Size Problem (MCSP): Given the <u>**truth table**</u> of a Boolean function f and an integer s, check if there is a circuit of size ≤ s that computes f.

- Easy to see that MCSP is in NP.

- Is MCSP NP-complete? Not known!
- Multi-output MCSP is NP-hard under poly-time randomized reductions. *(Ilango, Loff, Oliveira 2020)*

More NP-complete problems

Example 1: Independent Set

- INDSET := {(G, k): G has independent set of size k}

- Goal: Design a poly-time reduction f s.t.

$$x \in 3SAT \quad \Leftrightarrow \quad f(x) \in INDSET$$

- Reduction from 3SAT: Recall, a reduction is just an efficient algorithm that takes input a 3CNF ϕ and outputs a (G, k) tuple s.t

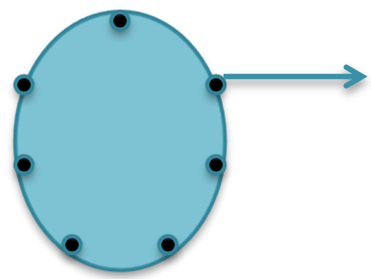
$$\phi \in 3SAT \quad \Leftrightarrow \quad (G, k) \in INDSET$$

Example 1: Independent Set

- Reduction: Let ϕ be a 3CNF with m clauses and n variables. Assume, every clause has exactly 3 literals.

Example 1: Independent Set

- Reduction: Let ϕ be a 3CNF with m clauses and n variables. Assume, every clause has exactly 3 literals.

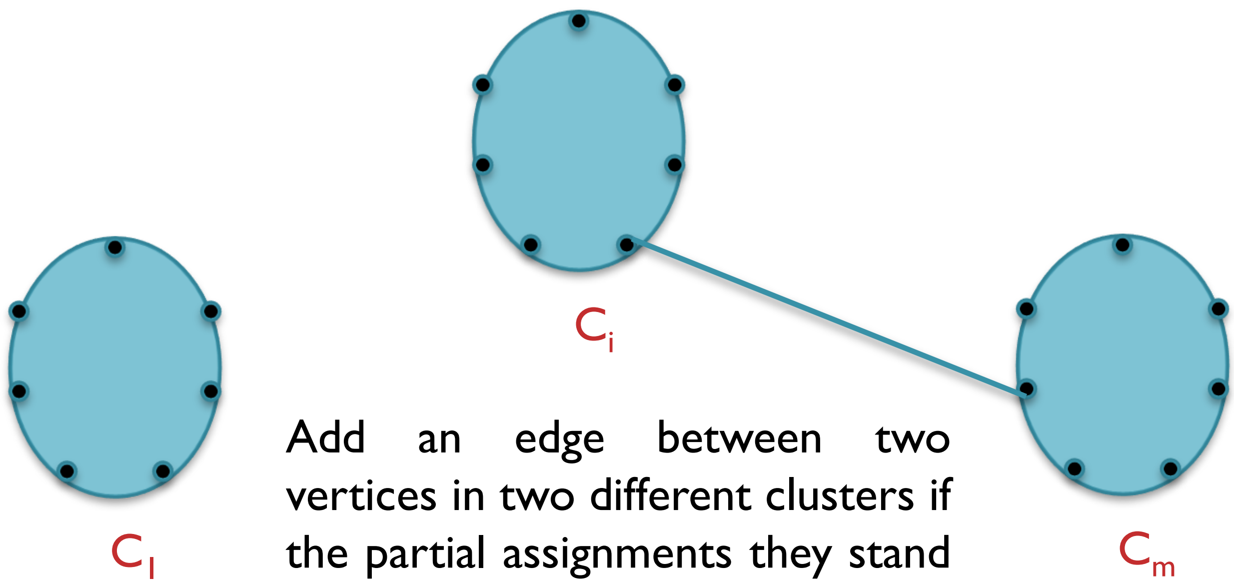


A vertex stands for a partial assignment of the variables in C_i that satisfies the clause

For every clause C_i form a complete graph (cluster) on 7 vertices

Example 1: Independent Set

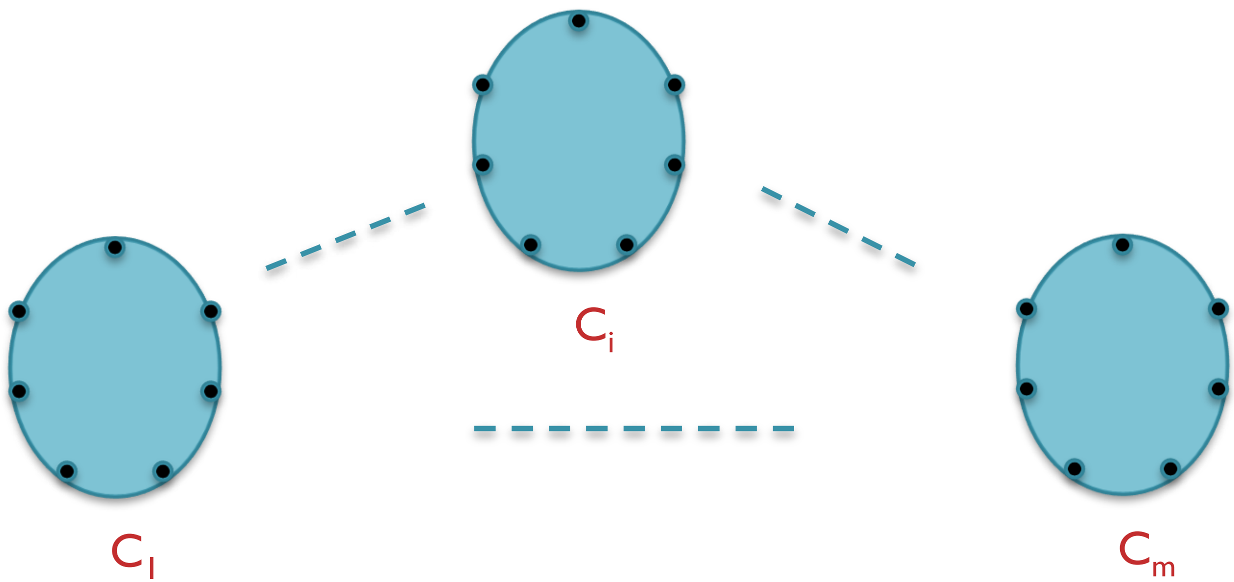
- Reduction: Let ϕ be a 3CNF with m clauses and n variables. Assume, every clause has exactly 3 literals.



Add an edge between two vertices in two different clusters if the partial assignments they stand for are <u>incompatible</u>.

Example 1: Independent Set

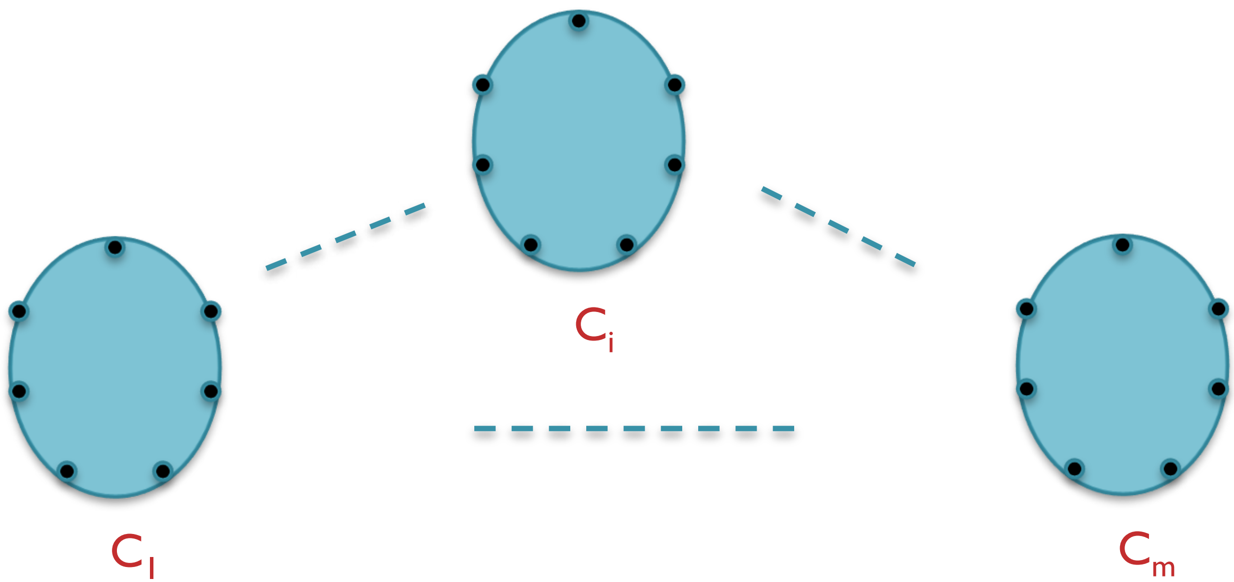
- Reduction: Let ϕ be a 3CNF with m clauses and n variables. Assume, every clause has exactly 3 literals.



Graph G on $7m$ vertices

Example 1: Independent Set

- Reduction: Let ϕ be a 3CNF with m clauses and n variables. Assume, every clause has exactly 3 literals.



- Obs: ϕ is satisfiable iff G has an ind. set of size m.

Example 2: Clique

- CLIQUE := {(H, k): H has a clique of size k}

- Goal: Design a poly-time reduction f s.t.

  $$x \in \text{INDSET} \Leftrightarrow f(x) \in \text{CLIQUE}$$

- Reduction from INDSET: The reduction algorithm computes $\overline{G}$ from G

  $$(G, k) \in \text{INDSET} \Leftrightarrow (\overline{G}, k) \in \text{CLIQUE}$$

Example 3: Vertex Cover

- VCover := {(H, k): H has a vertex cover of size k}

- Goal: Design a poly-time reduction f s.t.

  x ∈ INDSET ⟺ f(x) ∈ VCover

- Reduction from INDSET: Let n be the number of vertices in G. The reduction algorithm maps (G, k) to (G, n-k).

  (G, k) ∈ INDSET ⟺ (G, n-k) ∈ VCover

Example 4: 0/1 Integer Programming

- 0/1 IProg := Set of satisfiable 0/1 integer programs
- A <u>0/1 integer program</u> is a set of linear inequalities with rational coefficients and the variables are allowed to take only 0/1 values.

- Reduction from 3SAT: A clause is mapped to a linear inequality as follows

$$x_1 \vee \bar{x}_2 \vee x_3 \longrightarrow x_1 + (1 - x_2) + x_3 \geq 1$$

Example 5: Max Cut

- MaxCut : Given a graph find a <u>cut</u> with the max size.
- A <u>*cut*</u> of $G = (V, E)$ is a tuple $(U, V\backslash U)$, $U \subseteq V$. <u>Size</u> of a cut $(U, V\backslash U)$ is the number of edges from U to $V\backslash U$.

- MinVCover: Given a graph H, find a vertex cover in H that has the min size.

- Obs: From MinVCover(H), we can readily check if $(H, k) \in$ VCover, for any k.

Example 5: Max Cut

- MaxCut : Given a graph find a cut with the max size.
- A *cut* of $G = (V, E)$ is a tuple $(U, V \setminus U)$, $U \subseteq V$. Size of a cut $(U, V \setminus U)$ is the number of edges from U to $V \setminus U$.

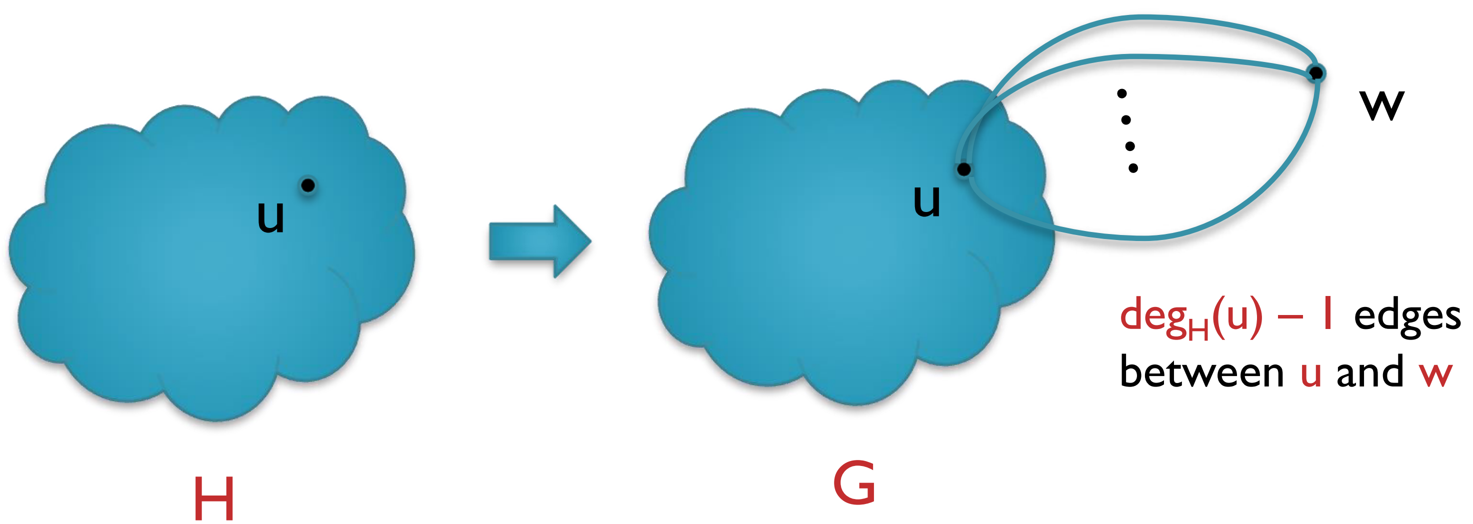
- Goal: A poly-time reduction from MinVCover to MaxCut.

$$H \xrightarrow{f} G \quad \text{s.t.}$$

$$\boxed{\text{Size of a } \mathrm{MaxCut}(G) = 2.|E(H)| - |\mathrm{MinVCover}(H)|}$$

Example 5: Max Cut

- The reduction: $H \xrightarrow{f} G$



- G is formed by adding a new vertex w and adding $\deg_H(u) - 1$ edges between every $u \in V(H)$ and w.

Example 5: Max Cut

- Claim: $|MaxCut(G)| = 2.|E(H)| - |MinVCover(H)|$

Example 5: Max Cut

- Claim: $|\text{MaxCut}(G)| = 2.|E(H)| - |\text{MinVCover}(H)|$
- Proof: Let $V(H) = V$. Then $V(G) = V + w$.

  Suppose $(U, V \setminus U + w)$ is a cut in G.

Example 5: Max Cut

- Claim: $|MaxCut(G)| = 2.|E(H)| - |MinVCover(H)|$
- Proof: Let $V(H) = V$. Then $V(G) = V + w$.

  Suppose $(U, V \setminus U + w)$ is a cut in G.

- Let $S_G(U)$:= no. of edges in G with exactly one end vertex incident on a vertex in U.

Example 5: Max Cut

- Claim: $|\text{MaxCut}(G)| = 2.|E(H)| - |\text{MinVCover}(H)|$
- Proof: Let $V(H) = V$. Then $V(G) = V + w$.

  Suppose $(U, V \setminus U + w)$ is a cut in G.

- Let $S_G(U)$ = no. of edges going out of U in G.

Example 5: Max Cut

- Claim: $|\mathrm{MaxCut}(G)| = 2.|E(H)| - |\mathrm{MinVCover}(H)|$
- Proof: Let $V(H) = V$. Then $V(G) = V + w$.

  Suppose $(U, V \backslash U + w)$ is a cut in G.

- Let $S_G(U)$ = size of the cut $(U, V \backslash U + w)$.

Example 5: Max Cut

- Claim: $|\mathrm{MaxCut}(G)| = 2.|E(H)| - |\mathrm{MinVCover}(H)|$
- Proof: Let $V(H) = V$. Then $V(G) = V + w$.

  Suppose $(U, V \backslash U + w)$ is a cut in G.

- Let $S_H(U)$:= no. of edges in H with <u>exactly one</u> end vertex incident on a vertex in U.

Example 5: Max Cut

- Claim: $|\text{MaxCut}(G)| = 2.|E(H)| - |\text{MinVCover}(H)|$
- Proof: Let $V(H) = V$. Then $V(G) = V + w$.

  Suppose $(U, V \backslash U + w)$ is a cut in G.

- Then $S_G(U) = S_H(U) + \sum_{u \in U} (\deg_H(u) - 1)$

$$= S_H(U) + \sum_{u \in U} \deg_H(u) - |U|$$

Example 5: Max Cut

- Claim: $|\text{MaxCut}(G)| = 2.|E(H)| - |\text{MinVCover}(H)|$
- Proof: Let $V(H) = V$. Then $V(G) = V + w$.

  Suppose $(U, V\backslash U + w)$ is a cut in G.

- Then $S_G(U) = S_H(U) + \sum_{u \in U} (\deg_H(u) - 1)$

$$= S_H(U) + \sum_{u \in U} \deg_H(u) - |U|$$

Obs: Twice the number of edges in H with <u>at least one</u> end vertex in U.

Example 5: Max Cut

- Claim: $|\text{MaxCut}(G)| = 2.|E(H)| - |\text{MinVCover}(H)|$
- Proof: Let $V(H) = V$. Then $V(G) = V + w$.

  Suppose $(U, V\backslash U + w)$ is a cut in G.

- Then $S_G(U) = S_H(U) + \sum_{u \in U} (\deg_H(u) - 1)$

$$= S_H(U) + \sum_{u \in U} \deg_H(u) - |U|$$

$$= 2.|E_H(U)| - |U|$$

$E_H(U)$:= Set of edges in H with <u>at least one</u> end vertex in U.

Example 5: Max Cut

- Claim: $|MaxCut(G)| = 2.|E(H)| - |MinVCover(H)|$
- Proof: Let $V(H) = V$. Then $V(G) = V + w$.

  Suppose $(U, V \backslash U + w)$ is a cut in G.

- Then $\boxed{S_G(U) = 2.|E_H(U)| - |U|}$ … Eqn (1)

- Proposition: If $(U, V \backslash U + w)$ is a <u>max cut</u> in G then U is a <u>vertex cover</u> in H.

Example 5: Max Cut

- Claim: $|MaxCut(G)| = 2.|E(H)| - |MinVCover(H)|$
- Proof: Let $V(H) = V$. Then $V(G) = V + w$.

  Suppose $(U, V \backslash U + w)$ is a cut in G.

- Then $\boxed{S_G(U) = 2.|E_H(U)| - |U|}$ … Eqn (1)

- Proposition: If $(U, V \backslash U + w)$ is a max cut in G then U is a vertex cover in H.

  ⇒ $S_G(U) = |MaxCut(G)| = 2.|E(H)| - |U|$

Example 5: Max Cut

- Claim: $|MaxCut(G)| = 2.|E(H)| - |MinVCover(H)|$
- Proof: Let $V(H) = V$. Then $V(G) = V + w$.

  Suppose $(U, V \backslash U + w)$ is a cut in G.

- Then $\boxed{S_G(U) = 2.|E_H(U)| - |U|}$ … Eqn (1)

- Proposition: If $(U, V \backslash U + w)$ is a <u>max cut</u> in G then U is a <u>vertex cover</u> in H.

  U must be a minVCover in H

  $\Rightarrow S_G(U) = |MaxCut(G)| = 2.|E(H)| - |U|$

Example 5: Max Cut

- Claim: $|MaxCut(G)| = 2.|E(H)| - |MinVCover(H)|$
- Proof: Let $V(H) = V$. Then $V(G) = V + w$.

  Suppose $(U, V \backslash U + w)$ is a cut in G.

- Then $\boxed{S_G(U) = 2.|E_H(U)| - |U|}$ … Eqn (1)

- Proposition: If $(U, V \backslash U + w)$ is a <u>max cut</u> in G then U is a <u>vertex cover</u> in H.

  ⇒ $S_G(U) = |MaxCut(G)| = 2.|E(H)| - |MinVCover(H)|$

Example 5: Max Cut

- Claim: $|MaxCut(G)| = 2.|E(H)| - |MinVCover(H)|$
- Proof: Let $V(H) = V$. Then $V(G) = V + w$.

  Suppose $(U, V \setminus U + w)$ is a cut in G.

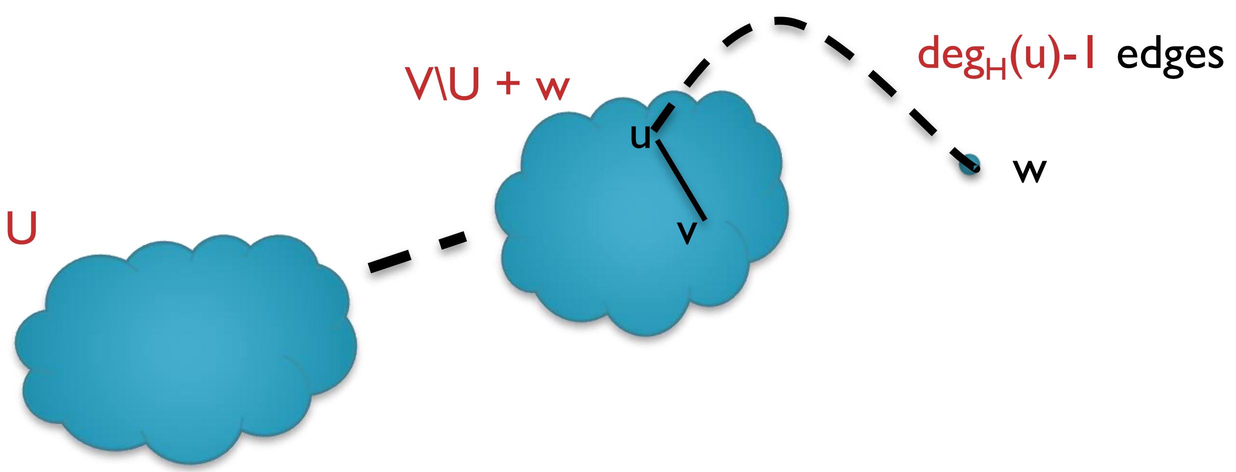
- Then $\boxed{S_G(U) = 2.|E_H(U)| - |U|}$ … Eqn (1)

- Proposition: If $(U, V \setminus U + w)$ is a max cut in G then U is a vertex cover in H.

Thus, the proof of the above claim follows from the proposition

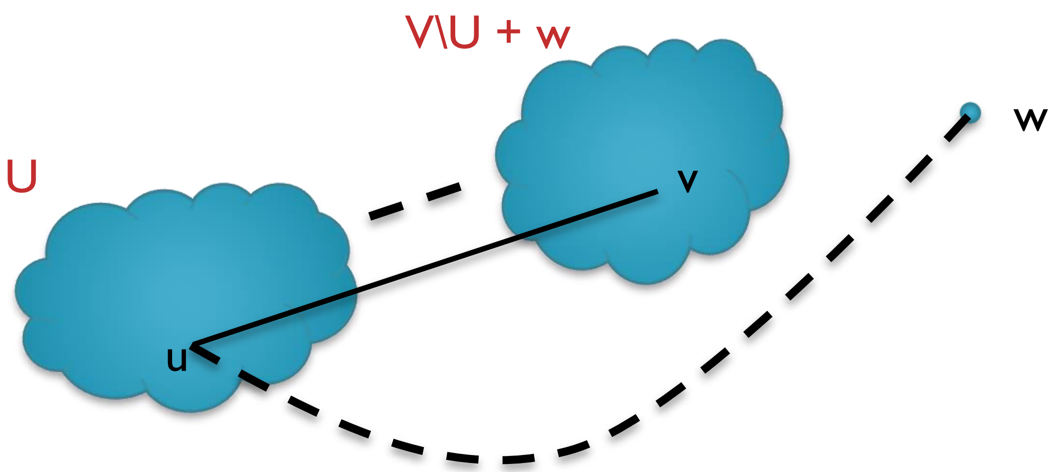
Example 5: Max Cut

- Proof of the Proposition: Suppose U is not a vertex cover



Example 5: Max Cut

- Proof of the Proposition: Suppose U is not a vertex cover



Gain: $deg_H(u)$-1 + 1 edges.
Loss: At most $deg_H(u)$-1 edges, these are the edges going from U to u.
Net gain: At least 1 edge. Hence the cut is not a max cut.

Search versus Decision

Search version of NP problems

- Recall: A language $L \subseteq \{0,1\}^*$ is in NP if
  - There's a *poly-time verifier* M and *poly. function* p s.t.
  - $x \in L$ iff there's a $u \in \{0,1\}^{p(|x|)}$ s.t $M(x, u) = 1$.

- Search version of L: Given an input $x \in \{0,1\}^*$, *find* a $u \in \{0,1\}^{p(|x|)}$ such that $M(x, u) = 1$, if such a u exists.

Search version of NP problems

- Recall: A language $L \subseteq \{0,1\}^*$ is in NP if
  - There's a *poly-time verifier* M and *poly. function* p s.t.
  - $x \in L$ iff there's a $u \in \{0,1\}^{p(|x|)}$ s.t $M(x, u) = 1$.

- Search version of L: Given an input $x \in \{0,1\}^*$, ***find*** a $u \in \{0,1\}^{p(|x|)}$ such that $M(x, u) = 1$, if such a u exists.

- Remark: Search version of L only makes sense once we have a verifier M in mind.

Search version of NP problems

- Recall: A language $L \subseteq \{0,1\}^*$ is in NP if
  - There's a *poly-time verifier* M and *poly. function* p s.t.
  - $x \in L$ iff there's a $u \in \{0,1\}^{p(|x|)}$ s.t $M(x, u) = 1$.

- Search version of L: Given an input $x \in \{0,1\}^*$, *find* a $u \in \{0,1\}^{p(|x|)}$ such that $M(x, u) = 1$, if such a u exists.

- Example: Given a 3CNF ϕ, find a satisfying assignment for ϕ if such an assignment exists.

Decision versus Search

- Is the search version of an NP-problem more difficult than the corresponding decision version?

Decision versus Search

- Is the search version of an NP-problem more difficult than the corresponding decision version?

- Theorem. Let $L \subseteq \{0,1\}^*$ be NP-complete. Then, the <u>search version of L</u> can be solved in poly-time <u>if and only if</u> the decision version can be solved in poly-time.

w.r.t any verifier M !

Decision versus Search

- Is the search version of an NP-problem more difficult than the corresponding decision version?

- Theorem. Let $L \subseteq \{0,1\}^*$ be NP-complete. Then, the search version of L can be solved in poly-time if and only if the decision version can be solved in poly-time.

- Proof. (search $\Rightarrow$ decision) Obvious.

Decision versus Search

- Is the search version of an NP-problem more difficult than the corresponding decision version?

- Theorem. Let $L \subseteq \{0,1\}^*$ be NP-complete. Then, the search version of L can be solved in poly-time if and only if the decision version can be solved in poly-time.

- Proof. (decision $\Longrightarrow$ search) We'll prove this for $L = \mathrm{SAT}$ first.

SAT is *downward self-reducible*

- Proof. (decision $\longrightarrow$ search) Let L = SAT, and A be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

SAT is *downward self-reducible*

- Proof. (decision $\Longrightarrow$ search) Let L = SAT, and A be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

$$\phi(x_1,\ldots,x_n)$$

SAT is *downward self-reducible*

- Proof. (decision $\Longrightarrow$ search) Let L = SAT, and A be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

$\phi(x_1,\ldots,x_n)$ $A(\phi) = Y$

SAT is *downward self-reducible*

- Proof. (decision $\Longrightarrow$ search) Let L = SAT, and A be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

$$\phi(x_1,\ldots,x_n) \qquad A(\phi) = Y$$

$$\swarrow$$

$$\phi(0,\ldots,x_n)$$

SAT is *downward self-reducible*

- Proof. (decision $\Longrightarrow$ search) Let L = SAT, and A be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

$\phi(x_1,\ldots,x_n)$ $A(\phi) = Y$

$A(\ \phi(0,..)\ ) = N$ $\phi(0,\ldots,x_n)$

SAT is *downward self-reducible*

- Proof. (decision $\longrightarrow$ search) Let L = SAT, and A be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

$$\begin{array}{ccc} & \phi(x_1,\ldots,x_n) & A(\phi) = Y \\ & \swarrow \qquad \searrow & \\ A(\ \phi(0,..)\ ) = N \quad \phi(0,\ldots,x_n) & & \phi(1,\ldots,x_n) \end{array}$$

SAT is *downward self-reducible*

- Proof. (decision $\rightarrow$ search) Let L = SAT, and A be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

$\phi(x_1,\ldots,x_n)$ $\quad A(\phi)$ = Y

A($\phi(0,..)$) = N $\quad \phi(0,\ldots,x_n)$ $\qquad\qquad \phi(1,\ldots,x_n)$ $\quad$ A($\phi(1,..)$) = Y

SAT is *downward self-reducible*

- Proof. (decision $\longrightarrow$ search) Let L = SAT, and A be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

$\phi(x_1,\ldots,x_n)$ $A(\phi) = Y$

$A(\ \phi(0,..)\ ) = N$ $\phi(0,\ldots,x_n)$

$\phi(1,\ldots,x_n)$ $A(\ \phi(1,..)\ ) = Y$

$\phi(1,0,\ldots,x_n)$

SAT is *downward self-reducible*

- Proof. (decision $\longrightarrow$ search) Let L = SAT, and *A* be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

$\phi(x_1,\ldots,x_n)$ A(ϕ) = Y

A($\phi(0,..)$) = N $\phi(0,\ldots,x_n)$ $\phi(1,\ldots,x_n)$ A($\phi(1,..)$) = Y

A($\phi(1,0,..)$) = Y $\phi(1,0,\ldots,x_n)$

SAT is *downward self-reducible*

- Proof. (decision $\longrightarrow$ search) Let L = SAT, and A be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

$$\phi(x_1,\ldots,x_n) \qquad A(\phi) = Y$$

$$A(\ \phi(0,..)\ ) = N \qquad \phi(0,\ldots,x_n) \qquad\qquad \phi(1,\ldots,x_n) \qquad A(\ \phi(1,..)\ ) = Y$$

$$A(\ \phi(1,0,..)\ ) = Y \qquad \phi(1,0,\ldots,x_n)$$

$$\phi(1,0,0,\ldots,x_n)$$

SAT is *downward self-reducible*

- Proof. (decision $\longrightarrow$ search) Let L = SAT, and *A* be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

$\phi(x_1,\ldots,x_n)$ A(ϕ) = Y

A(ϕ(0,..)) = N $\phi(0,\ldots,x_n)$

$\phi(1,\ldots,x_n)$ A(ϕ(1,..)) = Y

A(ϕ(1,0,..)) = Y $\phi(1,0,\ldots,x_n)$

A(ϕ(1,0,0...)) = N $\phi(1,0,0,\ldots,x_n)$

SAT is *downward self-reducible*

- **Proof.** (decision $\longrightarrow$ search) Let L = SAT, and A be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

$\phi(x_1,\ldots,x_n)$ $A(\phi)$ = Y

A($\phi(0,..)$) = N $\phi(0,\ldots,x_n)$ $\phi(1,\ldots,x_n)$ A($\phi(1,..)$) = Y

A($\phi(1,0,..)$) = Y $\phi(1,0,\ldots,x_n)$

A($\phi(1,0,0...)$) = N $\phi(1,0,0,\ldots,x_n)$ $\phi(1,0,1,\ldots,x_n)$

SAT is *downward self-reducible*

- Proof. (decision $\rightarrow$ search) Let L = SAT, and A be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

$\phi(x_1,\ldots,x_n)$ A(ϕ) = Y

A(ϕ(0,..)) = N $\phi(0,\ldots,x_n)$

$\phi(1,\ldots,x_n)$ A(ϕ(1,..)) = Y

A(ϕ(1,0,..)) = Y $\phi(1,0,\ldots,x_n)$

A(ϕ(1,0,0...)) = N $\phi(1,0,0,\ldots,x_n)$

$\phi(1,0,1,\ldots,x_n)$ A(ϕ(1,0,0...)) = Y

SAT is *downward self-reducible*

- **Proof.** (decision → search) Let L = SAT, and A be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

$\phi(x_1,\ldots,x_n)$ $A(\phi) = Y$

$A(\phi(0,..)) = N$ $\phi(0,\ldots,x_n)$

$\phi(1,\ldots,x_n)$ $A(\phi(1,..)) = Y$

$A(\phi(1,0,..)) = Y$ $\phi(1,0,\ldots,x_n)$

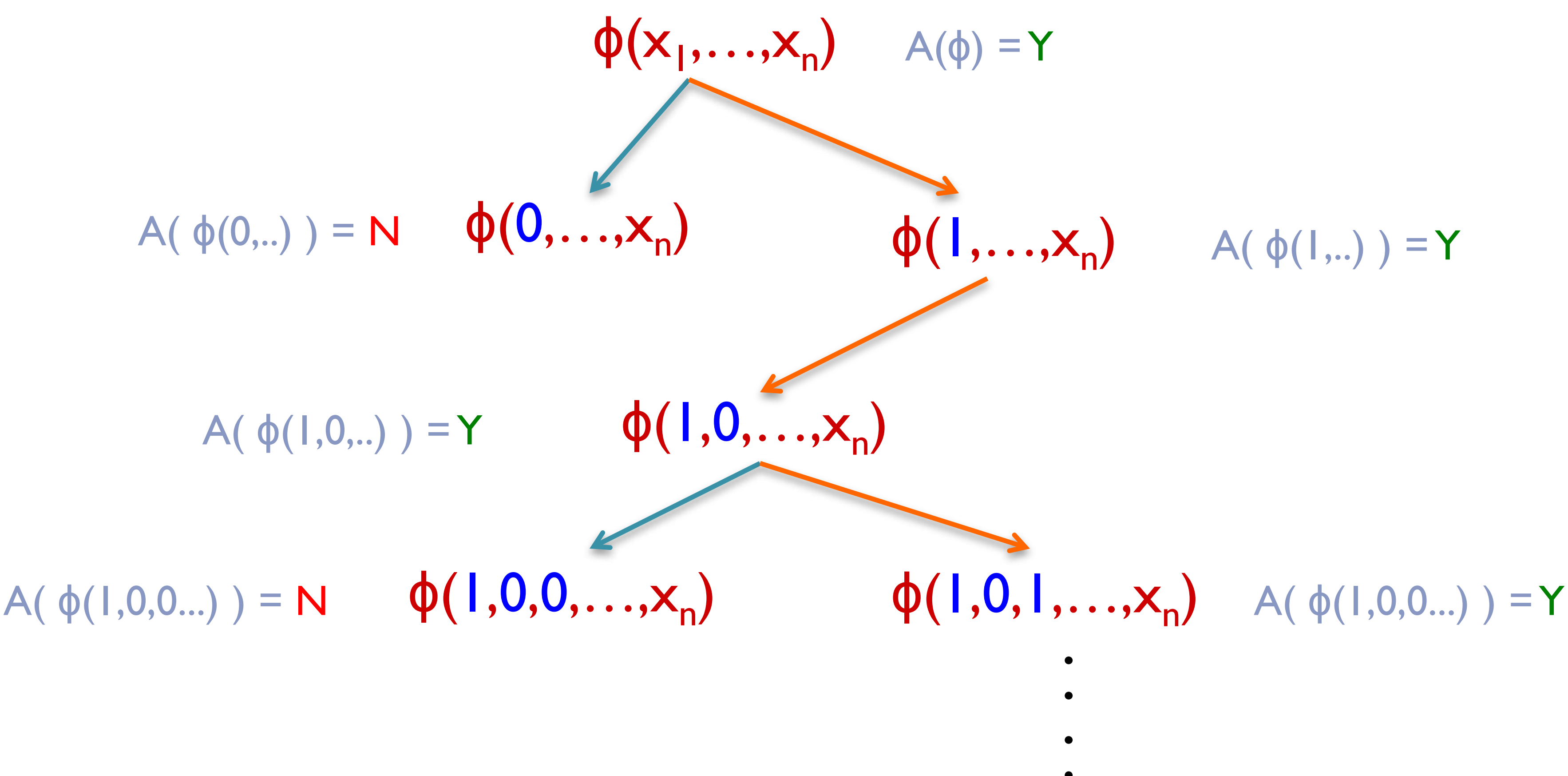
$A(\phi(1,0,0...)) = N$ $\phi(1,0,0,\ldots,x_n)$

$\phi(1,0,1,\ldots,x_n)$ $A(\phi(1,0,0...)) = Y$

$\vdots$

SAT is *downward self-reducible*

- Proof. (decision → search) Let L = SAT, and *A* be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.



SAT is *downward self-reducible*

- Proof. (decision ⟶ search) Let L = SAT, and *A* be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

- We can find a satisfying assignment of ϕ with at most $2n$ calls to A.

Decision ≡ Search for NPC problems

- Proof. (decision ⟶ search) Let L be NP-complete, M be a verifier for L, and *B* be a poly-time algorithm to decide if $x \in L$.

Decision ≡ Search for NPC problems

- Proof. (decision $\Longrightarrow$ search) Let L be NP-complete, M be a verifier for L, and *B* be a poly-time algorithm to decide if $x \in L$.

$SAT \le_p L$ $\qquad\qquad$ $L \le_p SAT$

Decision ≡ Search for NPC problems

- Proof. (decision $\Longrightarrow$ search) Let L be NP-complete, M be a verifier for L, and *B* be a poly-time algorithm to decide if $x \in L$.

$\text{SAT} \le_p L$ $\qquad\qquad\qquad\qquad$ $L \le_p \text{SAT}$

$$x \longmapsto \phi_x$$

Decision ≡ Search for NPC problems

- Proof. (decision ⟶ search) Let L be NP-complete, M be a verifier for L, and *B* be a poly-time algorithm to decide if $x \in L$.

SAT $\leq_p$ L

L $\leq_p$ SAT

$x \longmapsto \phi_x$

Important note:

From Cook-Levin theorem, we can find a certificate of $x \in L$ (w.r.t. M) from a satisfying assignment of ϕ_x.

Decision ≡ Search for NPC problems

- Proof. (decision ⟶ search) Let L be NP-complete, M be a verifier for L, and *B* be a poly-time algorithm to decide if $x \in L$.

$SAT \leq_p L$ $\qquad\qquad$ $L \leq_p SAT$

$x \longmapsto \phi_x$

How to find a satisfying assignment for ϕ_x <u>using algorithm B</u> ?

Decision ≡ Search for NPC problems

- Proof. (decision $\longrightarrow$ search) Let L be NP-complete, M be a verifier for L, and *B* be a poly-time algorithm to decide if $x \in L$.

SAT $\leq_p$ L

L $\leq_p$ SAT

$x \longmapsto \phi_x$

How to find a satisfying assignment for ϕ_x using algorithm B ?

...we know how using *A*, which is a poly-time decider for SAT

Decision ≡ Search for NPC problems

- Proof. (decision ⟹ search) Let L be NP-complete, M be a verifier for L, and *B* be a poly-time algorithm to decide if $x \in L$.

$$\text{SAT} \leq_p L \qquad\qquad L \leq_p \text{SAT}$$

$$\phi \longmapsto f(\phi) \qquad\qquad x \longmapsto \phi_x$$

How to find a satisfying assignment for ϕ_x using algorithm B ?

...we know how using *A*, which is a poly-time decider for SAT

Take $A(\phi) = B(\, f(\phi)\,)$.

Decision versus Search

- Is *search* equivalent to *decision* for every NP problem?

- Graph Isomorphism (GI) is in NP and (we'll see later that) it is unlikely to be NP-complete.
- Yet, the natural search version of GI reduces in polynomial-time to the decision version *(homework)*.

Decision versus Search

- Is *search* equivalent to *decision* for every NP problem?

Probably not!

Decision versus Search

- Is *search* equivalent to *decision* for every NP problem?

- Let $EE = \bigcup_{c \geq 0} DTIME\,(2^{c.2^n})$ and

  $NEE = \bigcup_{c \geq 0} NTIME\,(2^{c.2^n})$

  Doubly exponential analogues of P and NP

- Class NTIME(T(n)) will be defined formally in the next lecture.

Decision versus Search

- Is *search* equivalent to *decision* for every NP problem?

- Theorem. *(Bellare & Goldwasser 1994)* If EE $\neq$ NEE then there's a language in NP for which search does not reduce to decision.

Decision versus Search

- Is *search* equivalent to *decision* for every NP problem?

- Theorem. *(Bellare & Goldwasser 1994)* If EE $\neq$ NEE then there's a language in NP for which search does not reduce to decision.

- Checking if a number n is **composite** can be done in polynomial-time, but finding a factor of n is not known to be solvable in polynomial-time.
- We'll show that Intfact is unlikely to be NP-complete.

Decision versus Search

- Is *search* equivalent to *decision* for every NP problem?

- Theorem. *(Bellare & Goldwasser 1994)* If EE $\neq$ NEE then there's a language in NP for which search does not reduce to decision.

- Sometimes, the decision version of a problem can be trivial but the search version is possibly hard. E.g., Computing Nash Equilibrium (see class PPAD).

Homework: Read about **total NP functions**

Two types of poly-time reductions

- Definition. A language $L_1 \subseteq \{0,1\}^*$ is *polynomial-time (Karp or many-one) reducible* to a language $L_2 \subseteq \{0,1\}^*$ if there's a polynomial time computable function f s.t.

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

- Definition. A language $L_1 \subseteq \{0,1\}^*$ is *polynomial-time (Cook or Turing) reducible* to a language $L_2 \subseteq \{0,1\}^*$ if there's a TM that decides L_1 in poly-time using poly-many calls to a "subroutine" for deciding L_2 .

Two types of poly-time reductions

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

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

- Definition. A language $L_1 \subseteq \{0,1\}^*$ is <u>*polynomial-time (Cook or Turing) reducible*</u> to a language $L_2 \subseteq \{0,1\}^*$ if there's a TM that decides L_1 in poly-time using poly-many calls to a "subroutine" for deciding L_2 .

Will be called an <u>Oracle</u> later

Two types of poly-time reductions

- Definition. A language $L_1 \subseteq \{0,1\}^*$ is *polynomial-time (Karp or many-one) reducible* to a language $L_2 \subseteq \{0,1\}^*$ if there's a polynomial time computable function f s.t.

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

- Definition. A language $L_1 \subseteq \{0,1\}^*$ is *polynomial-time (Cook or Turing) reducible* to a language $L_2 \subseteq \{0,1\}^*$ if there's a TM that decides L_1 in poly-time using poly-many calls to a "subroutine" for deciding L_2 .

Karp reduction implies Cook reduction

Two types of poly-time reductions

- Definition. A language $L_1 \subseteq \{0,1\}^*$ is *polynomial-time (Karp or many-one) reducible* to a language $L_2 \subseteq \{0,1\}^*$ if there's a polynomial time computable function f s.t.

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

- Definition. A language $L_1 \subseteq \{0,1\}^*$ is *polynomial-time (Cook or Turing) reducible* to a language $L_2 \subseteq \{0,1\}^*$ if there's a TM that decides L_1 in poly-time using poly-many calls to a "subroutine" for deciding L_2 .

Homework: Read about **Levin reduction**