Computational Complexity Theory

Lecture 4: NP-complete problems, NTMs, Search versus Decision

Indian Institute of Science

Recap: Cook-Levin Theorem

- 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.

Recap: Cook-Levin Theorem

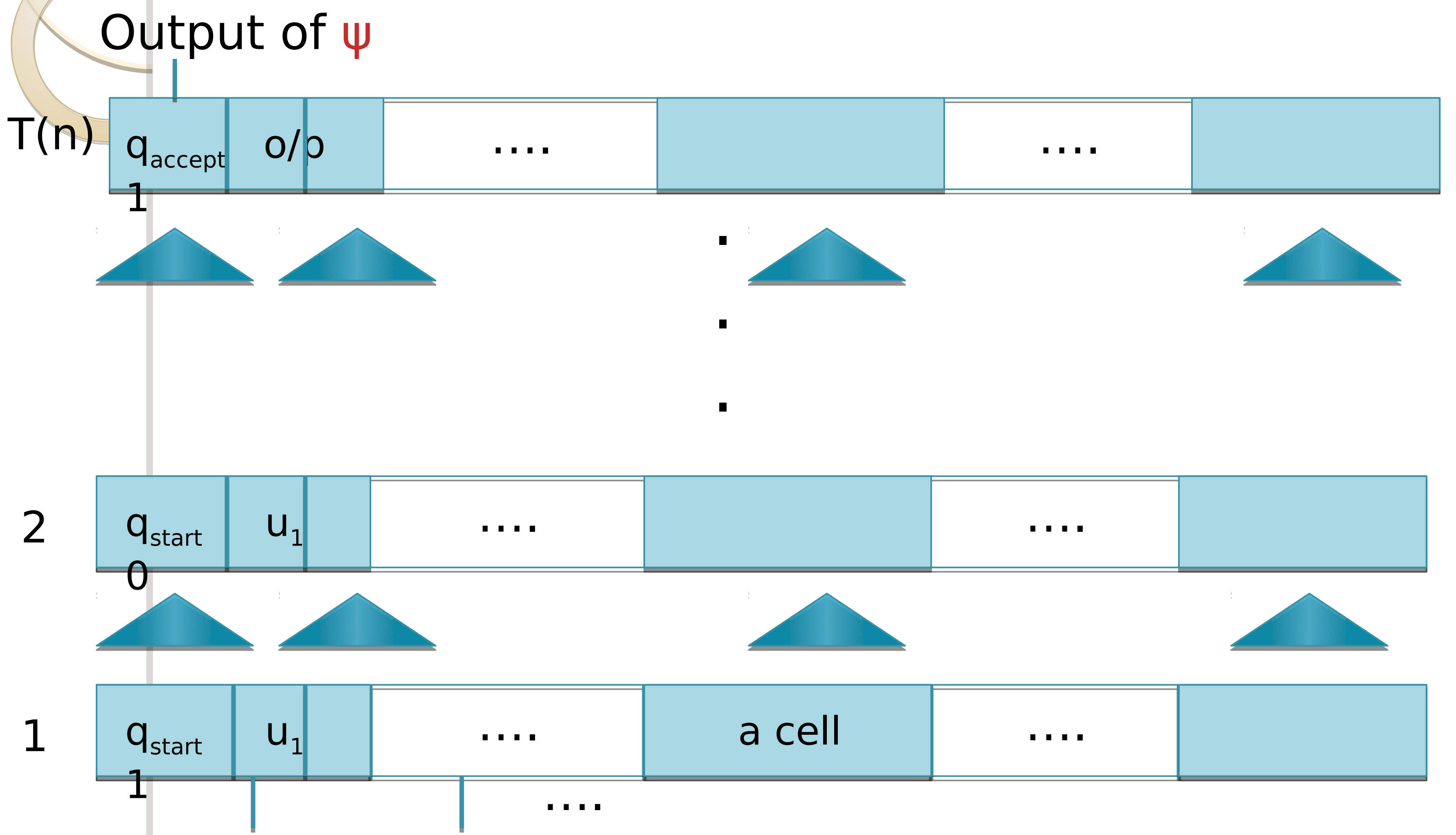
- Let $L \in$ NP. We intend to come up with a polynomial time computable function $f: x \mapsto \phi_x$ s.t.,

$$x \in L \iff \phi_x \in SAT$$

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

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

Recap: Cook-Levin Theorem



Input **u**-variables of ψ

Observe: ψ(u) = 1 iff N(u) = 1

Recap: Cook-Levin Theorem

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

$$x \in L \iff \phi_x \in SAT$$

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

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

ψ_x is a poly(|x|)-size circuit

Recap: Cook-Levin Theorem

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

$$x \in L \Leftrightarrow \phi_x \in SAT$$

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

$$x \in L \Leftrightarrow \psi_x(u) \text{ is satisfiable}$$

- **Important note:** A satisfying assignment u for ψ_x trivially gives a certificate u such that $M(x, u) = 1$.

Recap: Cook-Levin Theorem

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

$$x \in L \Leftrightarrow \phi_x \in SAT$$

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

$$x \in L \Leftrightarrow \psi_x(u) \text{ is satisfiable}$$

a poly-size circuit but not a poly-size CNF

Recap: Cook-Levin Theorem

- From circuit to CNF. From circuit ψ construct a CNF ϕ by introducing some extra variables v such that

$$\psi(u) = 1 \text{ iff } \phi(u, v) \text{ is satisfiable.}$$

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

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

Recap: Cook-Levin Theorem

- From circuit to CNF. From circuit ψ construct a CNF ϕ by introducing some <u>extra variables</u> v such that

$$\psi(u) = 1 \text{ iff } \phi(u, v) \text{ is satisfiable.}$$

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

$$x \in L \Leftrightarrow \phi_x(u, v) \text{ is satisfiable}$$

- <u>**Important note:**</u> A satisfying assignment (u, v) for ϕ_x trivially gives a certificate u such that $M(x, u) = 1$.

Recap: Cook-Levin Theorem

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

  e.g. a 2CNF $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

- Definition. kSAT is the language consisting of all *satisfiable kCNFs.*

- Cook-Levin. There's some constant k such that kSAT is NP-complete.

Recap: 3SAT is NP-complete

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

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

- Definition. kSAT is the language consisting of all *satisfiable kCNFs.*

- Theorem. 3SAT is NP-complete.

Proof sketch: $(x_1 \lor x_2 \lor x_3 \lor \neg x_4)$ is satisfiable iff $(x_1 \lor x_2 \lor z) \land (x_3 \lor \neg x_4 \lor \neg z)$ is satisfiable.

More NP-complete problems

NP-complete problems: Examples

- Independent Set
- Clique
- Vertex Cover
- 0/1 Integer Programming
- Max-Cut (NP-hard)

And many many other natural 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 ∈ 3SAT ⟺ f(x) ∈ 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

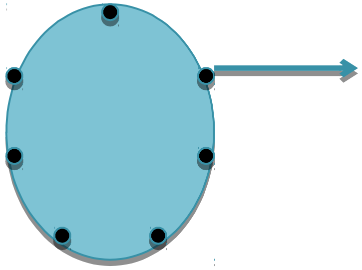
  ϕ ∈ 3SAT ⟺ (G, k) ∈ INDSET

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.

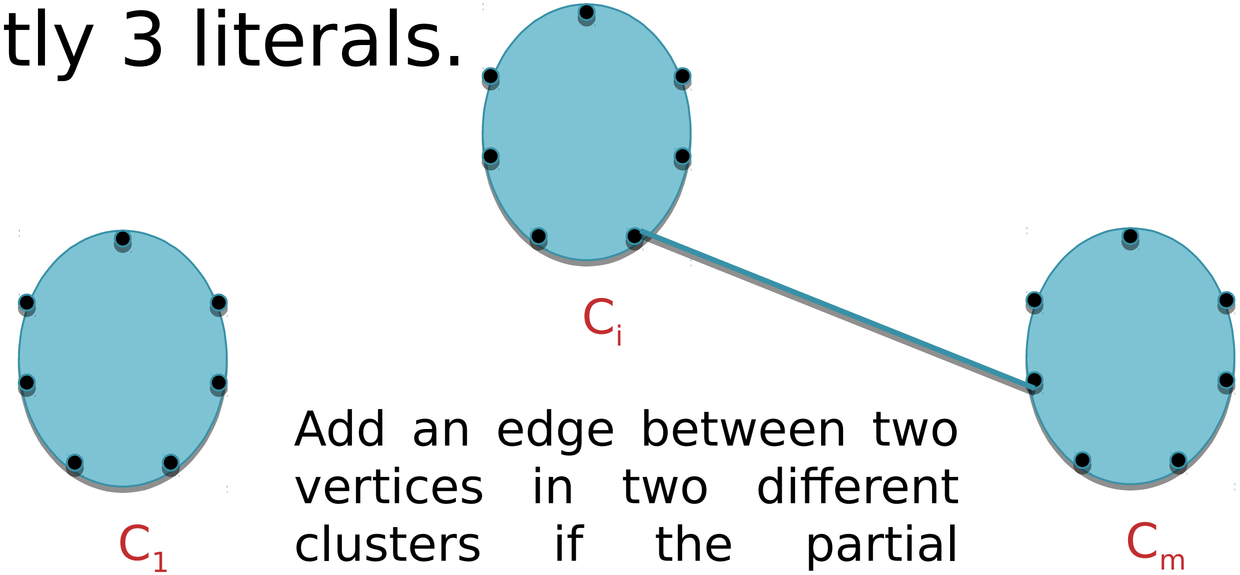


A vertex stands for a partial assignment of the variables in C_i that satisfies the clause

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

Example 1: Independent Set

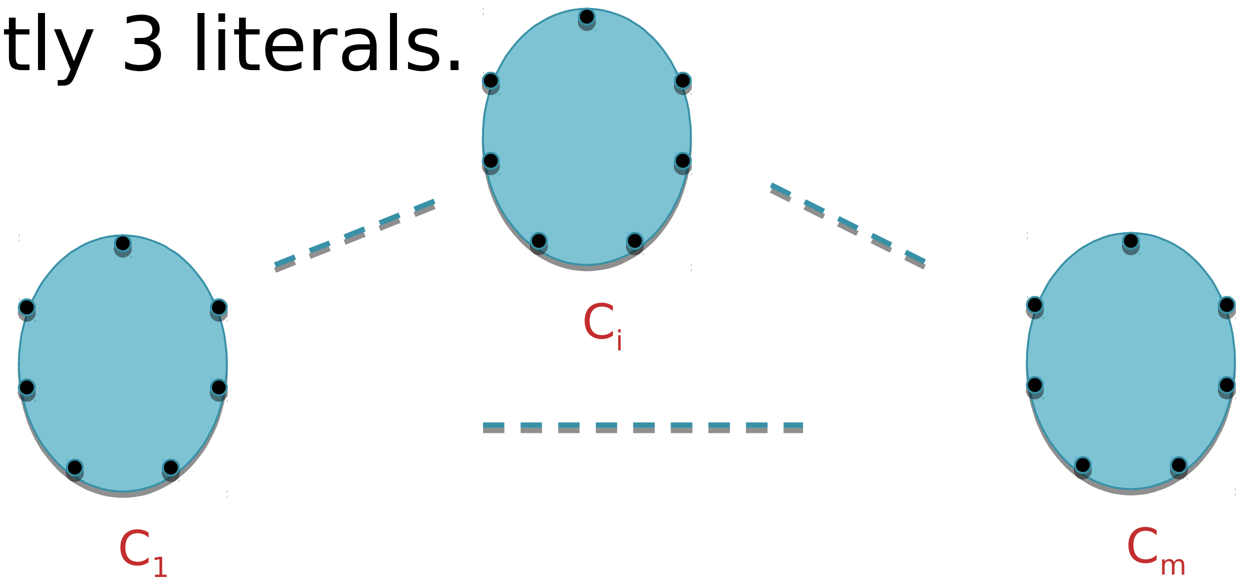
- Reduction: Let ϕ be a 3CNF with m clauses and n variables. Assume, every clause has exactly 3 literals.



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

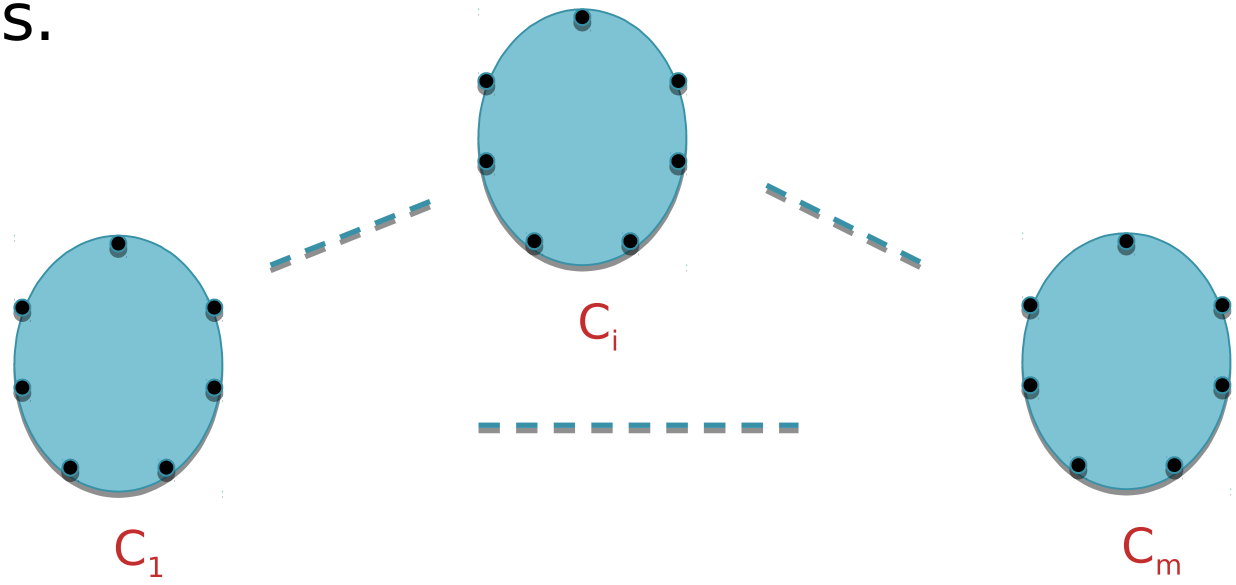
- Reduction: Let ϕ be a 3CNF with m clauses and n variables. Assume, every clause has exactly 3 literals.



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 ∈ INDSET ⟺ f(x) ∈ CLIQUE

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

  (G, k) ∈ INDSET ⟺ ($\bar{G}$, k) ∈ 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 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 $V \setminus U$.

- MinVCover: Given H, find a Vcover with 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 $(U, V \backslash U)$, $U \subseteq V$. Size of a cut $(U, V \backslash U)$ is the number of edges from U to $V \backslash U$.
- Goal: A poly-time reduction from VCover to MaxCut.

$$(H, k) \xrightarrow{f} G$$

s.t.

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

Example 5: Max Cut

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

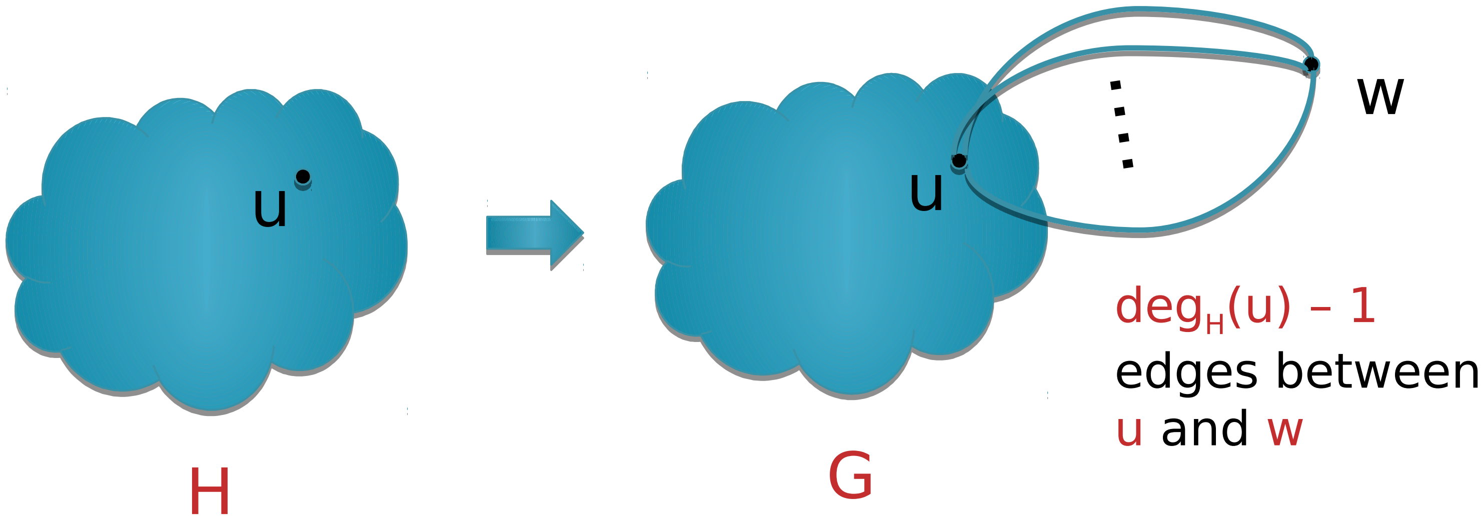
- Goal: A poly-time <u>reduction</u> from VCover to MaxCut.

$(H, k) \xrightarrow{f} G$ s.t.

Thus, checking if (H, k) ∈ VCover reduces to finding MaxCut(G).

Example 5: Max Cut

- The reduction: (H, k) $\xrightarrow{f}$ 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: |MaxCut(G)| = 2.|E(H)| - |MinVCover(H)|
- Proof: Let V(H) = V. Then V(G) = V + w.

  Suppose (U, V\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 $(U, V \backslash U + w)$ 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: $|MaxCut(G)| = 2.|E(H)| - |MinVCover(H)|$
- 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 going out of U 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 $(U, V \backslash U + w)$ is a cut in G.

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

Example 5: Max Cut

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

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

- Let $S_H(U)$ = no. of edges in H with exactly one end vertex incident on a 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 $S_G(U) = S_H(U) + \sum_{u \in U} (deg_H(u) - 1)$

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

Example 5: Max Cut

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

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

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

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

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

Example 5: Max Cut

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

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

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

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

$$= 2.|E_U(H)| - |U|$$

Example 5: Max Cut

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

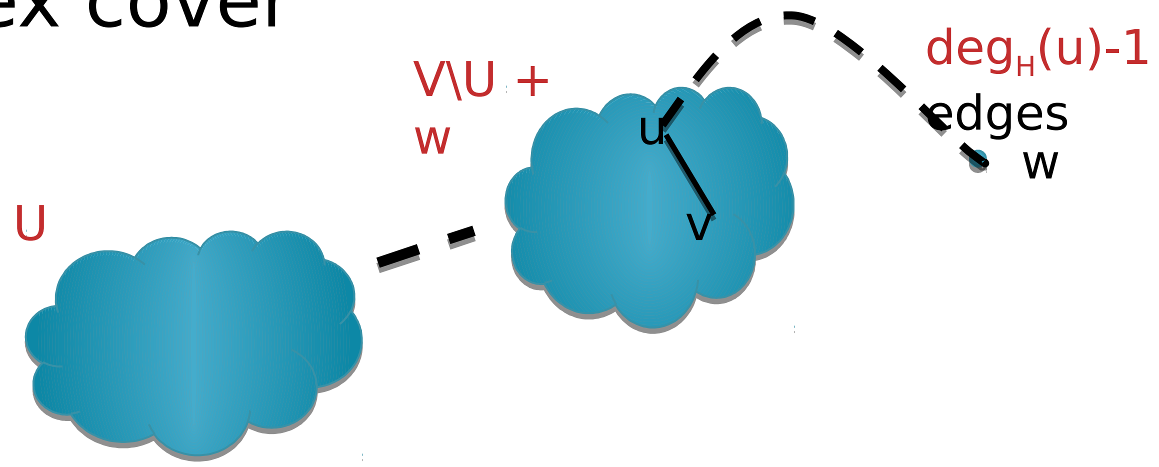
  Suppose (U, V\U + w) is a cut in G.

  ... Eqn (1)
- Then $S_G(U) = 2.|E_U(H)| - |U|$
- Proposition: If (U, V\U + w) is a max cut in G then U is a vertex cover in H.

...proof of the claim follows from the above proposition

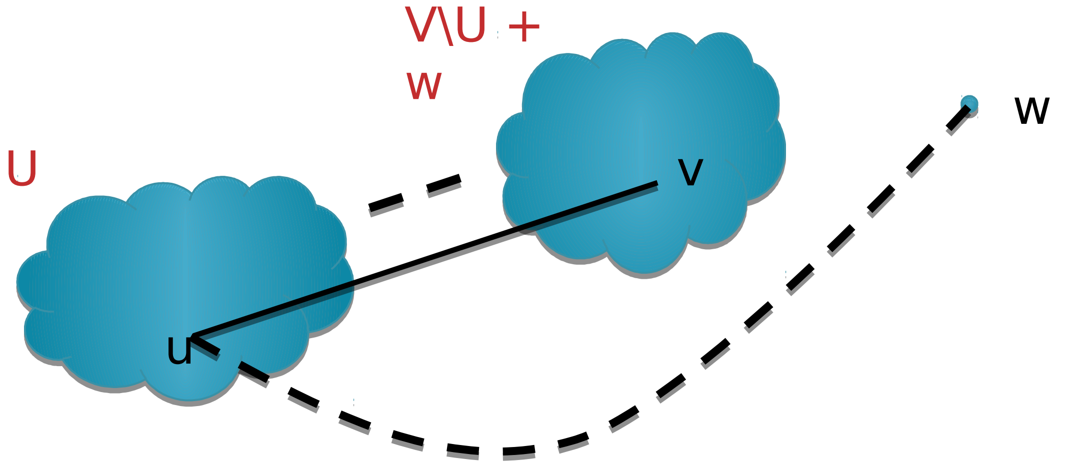
Example 5: Max Cut

- 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.

NTM: An alternate characterization of NP

Nondeterministic Turing Machines

- A *nondeterministic Turing machine* is like a deterministic Turing machines but with two transition functions.
- It is formally defined by a tuple $(\Gamma, Q, \delta_0, \delta_1)$. It has a special state q_{accept} in addition to q_{start} and q_{halt}.

Nondeterministic Turing Machines

- A *nondeterministic Turing machine* is like a deterministic Turing machines but with two transition functions.
- It is formally defined by a tuple $(\Gamma, Q, \delta_0, \delta_1)$. It has a special state q_{accept} in addition to q_{start} and q_{halt}.
- At every step of computation, the machine applies one of two functions δ_0 and δ_1 *arbitrarily*.

this is different from *randomly*

Nondeterministic Turing Machines

- A *nondeterministic Turing machine* is like a deterministic Turing machines but with two transition functions.
- It is formally defined by a tuple $(\Gamma, Q, \delta_0, \delta_1)$. It has a special state q_{accept} in addition to q_{start} and q_{halt}.
- At every step of computation, the machine applies one of two functions δ_0 and δ_1 <u>*arbitrarily*</u>.

also called <u>*nondeterministically*</u>

Nondeterministic Turing Machines

- A *nondeterministic Turing machine* is like a deterministic Turing machines but with two transition functions.
- It is formally defined by a tuple $(\Gamma, Q, \delta_0, \delta_1)$. It has a special state q_{accept} in addition to q_{start} and q_{halt}.
- At every step of computation, the machine applies one of two functions δ_0 and δ_1 *arbitrarily*.
- Unlike DTMs, NTMs are not intended to be physically realizable (because of the

Nondeterministic Turing Machines

- Definition. An NTM M *accepts* a string $x \in \{0,1\}^*$ iff on input x there ***exists*** a sequence of applications of the transition functions δ_0 and δ_1 (beginning from the start configuration) that makes M reach q_{accept}.

- Defintion. An NTM M *decides* a language $L \subseteq \{0,1\}^*$ if
  - M accepts x $\quad\quad$ $x \in L$
  - On every sequence of applications of the transition functions on input x, M either

Nondeterministic Turing Machines

- Definition. An NTM M *accepts* a string $x \in \{0,1\}^*$ iff on input x there ***exists*** a sequence of applications of the transition functions δ_0 and δ_1 (beginning from the start configuration) that makes M reach q_{accept}.

  ⟺

- Defintion. An NTM M *decides* a language $L \subseteq \{0,1\}^*$ if
  - M accepts x ⟺ $x \in L$
  - On every sequence of applications of the transition functions on input x, M either

remember in this course we'll always be dealing with TMs that halt on every input.

Nondeterministic Turing Machines

- Definition. An NTM M *accepts* a string $x \in \{0,1\}^*$ iff on input x there ***exists*** a sequence of applications of the transition functions δ_0 and δ_1 (beginning from the start configuration) that makes M reach q_{accept}.

- Defintion. An NTM M *decides* L in $T(|x|)$ time if
  - M accepts x $x \in L$
  - On every sequence of applications of the transition functions on input x, M either

Class NTIME

- Definition. A language L is in $NTIME(T(n))$ if there's an NTM M that decides L in $c.\ T(n)$ time on inputs of length n, where c is a constant.

Alternate characterization of NP

- Definition. A language L is in $NTIME(T(n))$ if there's an NTM M that decides L in $c.\,T(n)$ time on inputs of length n, where c is a constant.

- Theorem. $NP = \bigcup_{c \geq 0} NTIME(n^c)$.

Proof sketch: Let L be a language in NP. Then, there's a poly-time verifier M s.t,

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

Alternate characterization of NP

- **Definition.** A language L is in $\mathrm{NTIME}(T(n))$ if there's an NTM M that decides L in $c.\ T(n)$ time on inputs of length n, where c is a constant.

- **Theorem.** $\mathrm{NP} = \bigcup_{c>0} \mathrm{NTIME}(n^c)$.

Proof sketch: Let L be a language in NP. Then, there's a poly-time verifier M s.t,

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

Think of an NTM M' that on input x, at first <u>*guesses*</u> a $u \in \{0,1\}^{p(|x|)}$ by applying δ_0 and δ_1 nondeterministically

Alternate characterization of NP

- **Definition.** A language L is in NTIME($T(n)$) if there's an NTM M that decides L in $c.\ T(n)$ time on inputs of length n, where c is a constant.

- **Theorem.** $NP = \cup_{c>0}\, \mathrm{NTIME}(n^c)$.

Proof sketch: Let L be a language in NP. Then, there's a poly-time verifier M s.t,

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

…. and then simulates M on (x, u) to verify $M(x,u) = 1$.

Alternate characterization of NP

- Definition. A language L is in NTIME(T(n)) if there's an NTM M that decides L in c. T(n) time on inputs of length n, where c is a constant.

- Theorem. $NP = \cup_{c \geq 0} NTIME(n^c)$.

Proof sketch: Let L be in $NTIME(n^c)$. Then, there's an NTM M' that decides L in $p(n) = O(n^c)$ time. $(|x| = n)$

Alternate characterization of NP

- Definition. A language L is in NTIME(T(n)) if there's an NTM M that decides L in c. T(n) time on inputs of length n, where c is a constant.

- Theorem. NP = $\cup_{c>0}$ NTIME (n^c).

  Proof sketch: Let L be in NTIME (n^c). Then, there's an NTM M' that decides L in $p(n) = O(n^c)$ time. ($|x| = n$)

Think of a verifier M that takes x and u $\in\{0,1\}^{p(n)}$ as input,

Alternate characterization of NP

- **Definition.** A language L is in $NTIME(T(n))$ if there's an NTM M that decides L in $c.\ T(n)$ time on inputs of length n, where c is a constant.

- **Theorem.** $NP = \cup_{c > 0} NTIME(n^c)$.

Proof sketch: Let L be in $NTIME(n^c)$. Then, there's an NTM M' that decides L in $p(n) = O(n^c)$ time. $(|x| = n)$

Think of a verifier M that takes x and $u \in \{0,1\}^{p(n)}$ as input, and simulates M' on x with u as the sequence of choices for applying δ_0 and δ_1 .

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 such that
  - $x \in L$ iff there's a *poly-size certificate* u 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 such that
  - $x \in L$ iff there's a *poly-size certificate* u 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 search version of L can be solved in poly-time if and only if the decision version can be solved in poly-time.

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 $\Rightarrow$ search) We'll prove this for $L = SAT$ first.

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.

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.

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

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.

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

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.

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

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

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.

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

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

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.

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

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

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

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.

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

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

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

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.

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

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

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

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

SAT is *downward self-reducible*

- Proof. (decision $\Rightarrow$ 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 $\phi(1,0,\ldots,x_n)$

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

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

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(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 → 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 $\phi(1,0,0,\ldots,x_n)$

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

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.

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

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

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

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

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

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

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.

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

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

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

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

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

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

$\vdots$

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

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

⋮

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.

- 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, and B be a poly-time algorithm to decide if $x \in L$.

Decision ≡ Search for NPC problems

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

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

Decision ≡ Search for NPC problems

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

$\mathrm{SAT} \leq_p L$

$L \leq_p \mathrm{SAT}$

$x \longmapsto \phi_x$

Decision ≡ Search for NPC problems

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

$SAT \leq_p L$

$L \leq_p SAT$

$x \longmapsto \phi_x$

From Cook-Levin theorem, we can find a certificate of x from a satisfying assignment of ϕ_x.

Decision ≡ Search for NPC problems

- Proof. (decision $\Rightarrow$ search) Let L be NP-complete, 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 certificate of ϕ_x using algorithm B ?

Decision ≡ Search for NPC problems

- Proof. (decision $\Rightarrow$ search) Let L be NP-complete, 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 certificate of ϕ_x using algorithm B ?

… we know how to using *A*, a poly-time decider for SAT

Decision ≡ Search for NPC problems

- Proof. (decision $\Rightarrow$ search) Let L be NP-complete, 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 certificate of ϕ_x using algorithm B ?

… we know how to using A, a poly-time decider for SAT

Decision ≡ Search for NPC problems

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

$\mathrm{SAT} \le_p L$

$\phi \longmapsto f(\phi)$

$L \le_p \mathrm{SAT}$

$x \longmapsto \phi_x$

How to find a certificate of ϕ_x using algorithm B ?

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

Decision versus Search

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

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

Decision versus Search

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

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