Computational Complexity Theory

Lecture 3: NP-complete problems, Search versus Decision

Department of Computer Science,
Indian Institute of Science

Recap: Cook-Levin Theorem

- 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.*

- Theorem. *(Cook 1971, Levin 1973)* SAT is NP-complete.

Recap: Cook-Levin Theorem

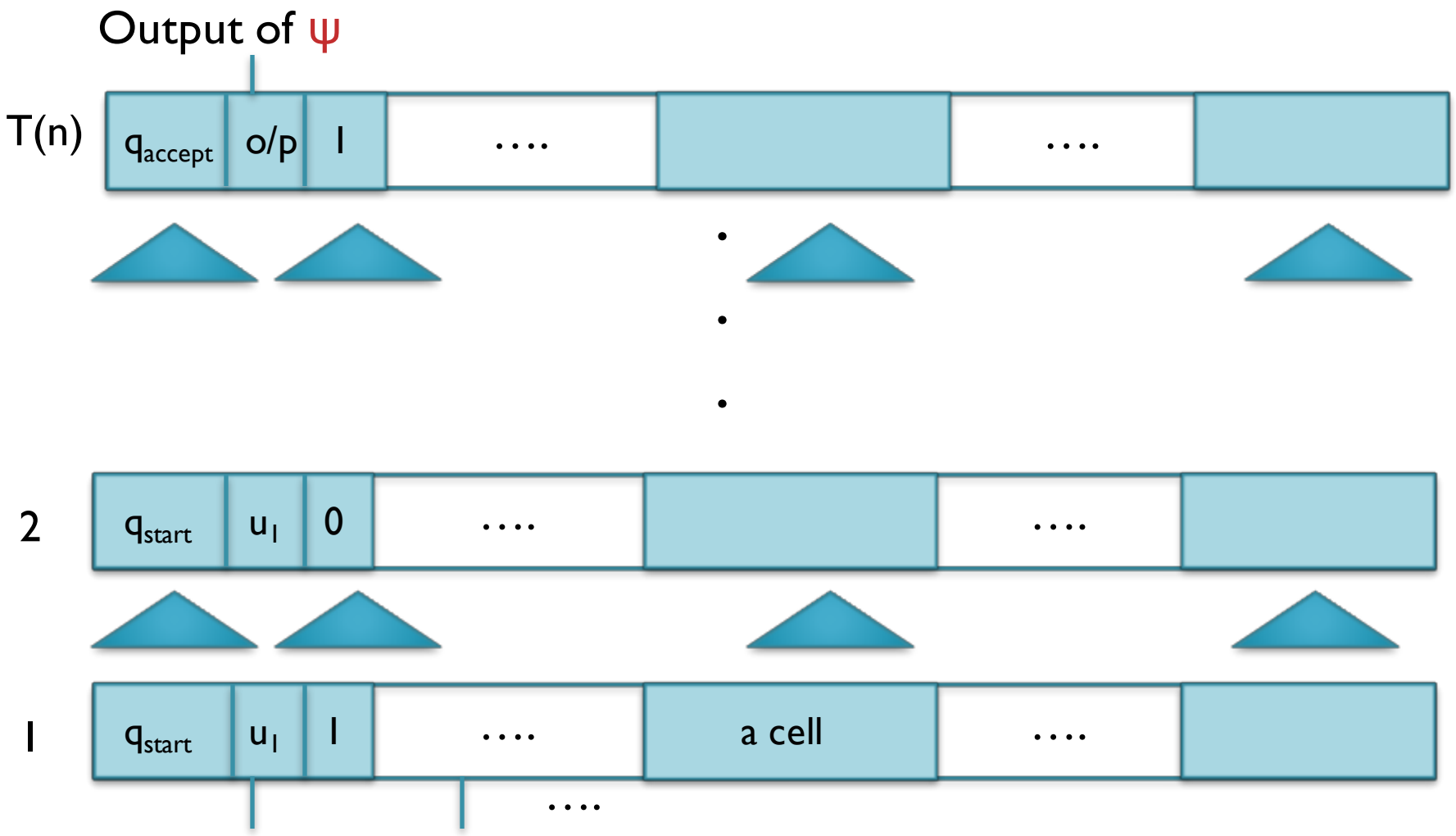
- Let $L \in NP$. We intend to come up with a polynomial-time computable function $f: x \longmapsto \phi_x$ s.t.,

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

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

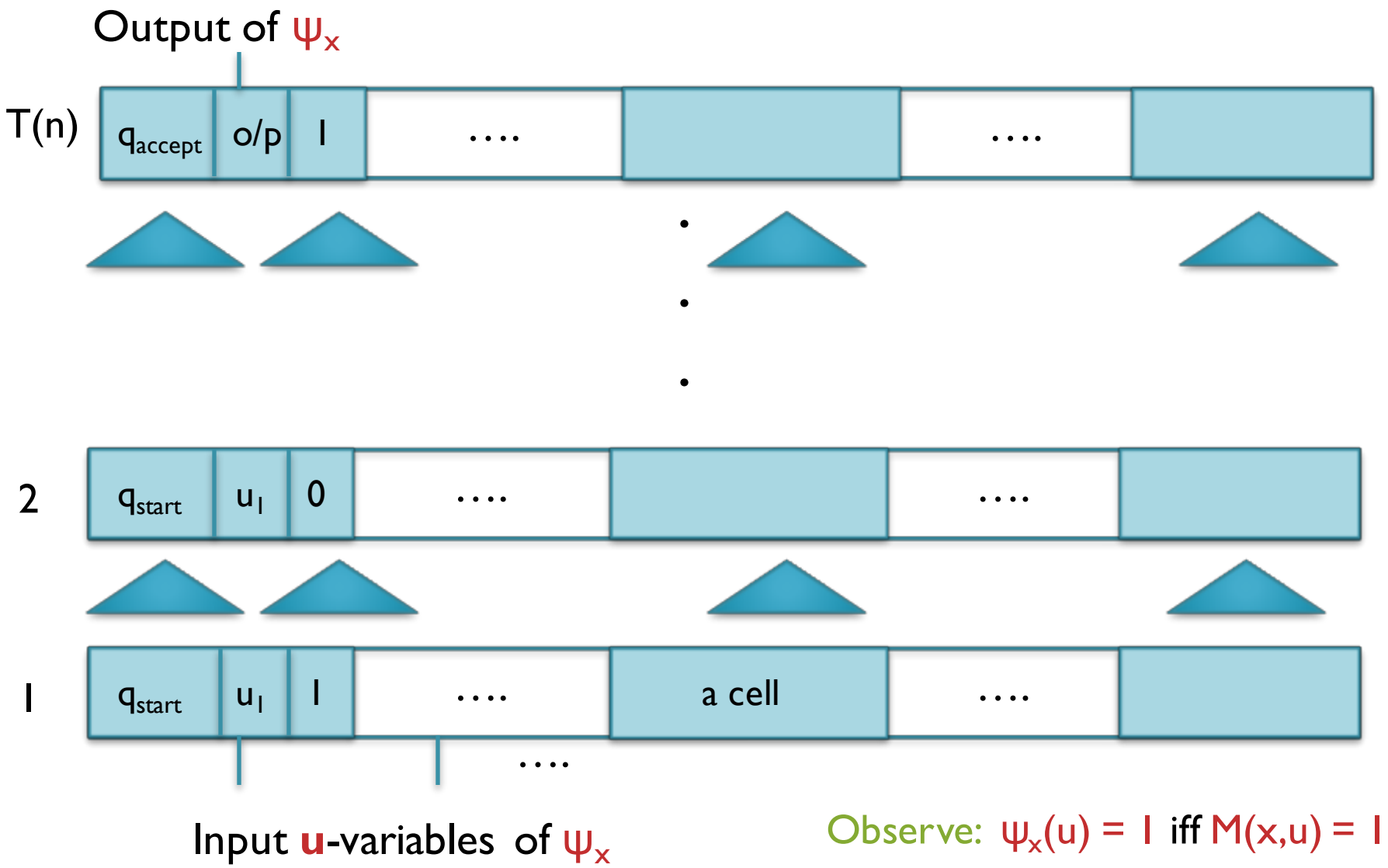
$$x \in L \iff \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

Recap: Cook-Levin Theorem



Observe: ψ(u) = 1 iff N(u) = 1

Recap: Cook-Levin Theorem



Observe: $\psi_x(u) = 1$ iff $M(x,u) = 1$

Recap: Cook-Levin Theorem

- Let $L \in NP$. We intend to come up with a polynomial time computable function $f: x \longmapsto \phi_x$ 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($|x|$)-time computable

Recap: Cook-Levin Theorem

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

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

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

  $x \in L \iff \psi_x(u)$ is satisfiable

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

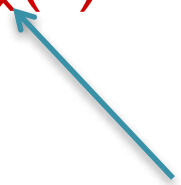
Recap: Cook-Levin Theorem

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

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

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

  $x \in L \iff \psi_x(u)$ is satisfiable



A circuit but not a CNF !

Recap: Cook-Levin Theorem

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

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

- ψ and ϕ are **not** equivalent formulas!

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

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

Recap: Cook-Levin Theorem

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

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

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

  $x \in L \iff \phi_x(u, v) \in SAT$

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

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 2CNF $\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_2)$

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

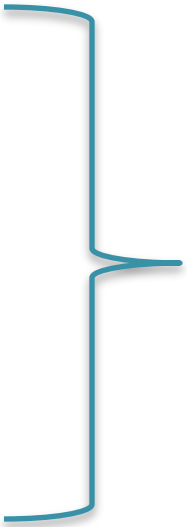
- Theorem. 3-SAT is NP-complete.

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

More NP-complete problems

NP complete problems: Examples

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



Karp 1972

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

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

NPC problems from number theory

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

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

- Theorem: SqRootMod is NP-complete.

Manders & Adleman 1976

NPC problems from number theory

- Variant_IntFact : Given natural numbers L, U and N, check if there exists a **natural number** d ∈ [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*

A peculiar NP problem

- Minimum Circuit Size Problem (MCSP): Given the truth table of a Boolean function f and an integer s, check if there is a circuit of size $\leq$ s that computes f.

- Easy to see that MCSP is in NP.

- Is MCSP NP-complete? Not known!

Example 1: Independent Set

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

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

$$x \in 3SAT \iff f(x) \in INDSET$$

- 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

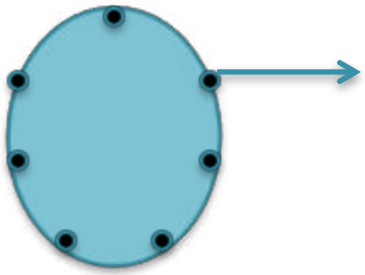
$$\phi \in 3SAT \iff (G, k) \in INDSET$$

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.

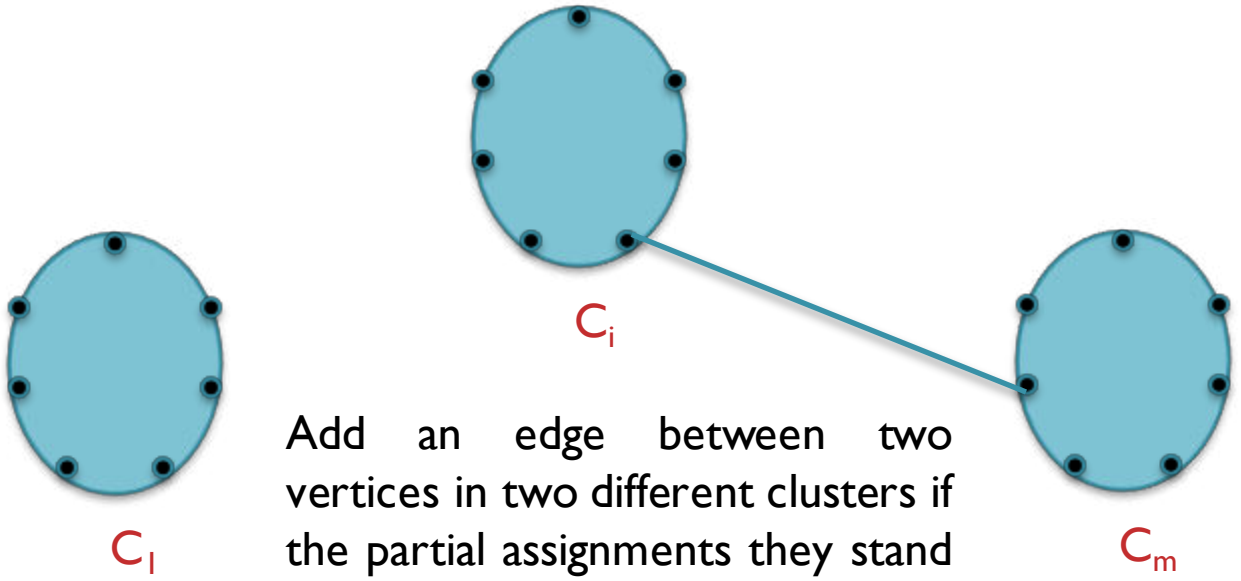


A vertex stands for a partial assignment of the variables in C_i that satisfies the clause

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

Example 1: Independent Set

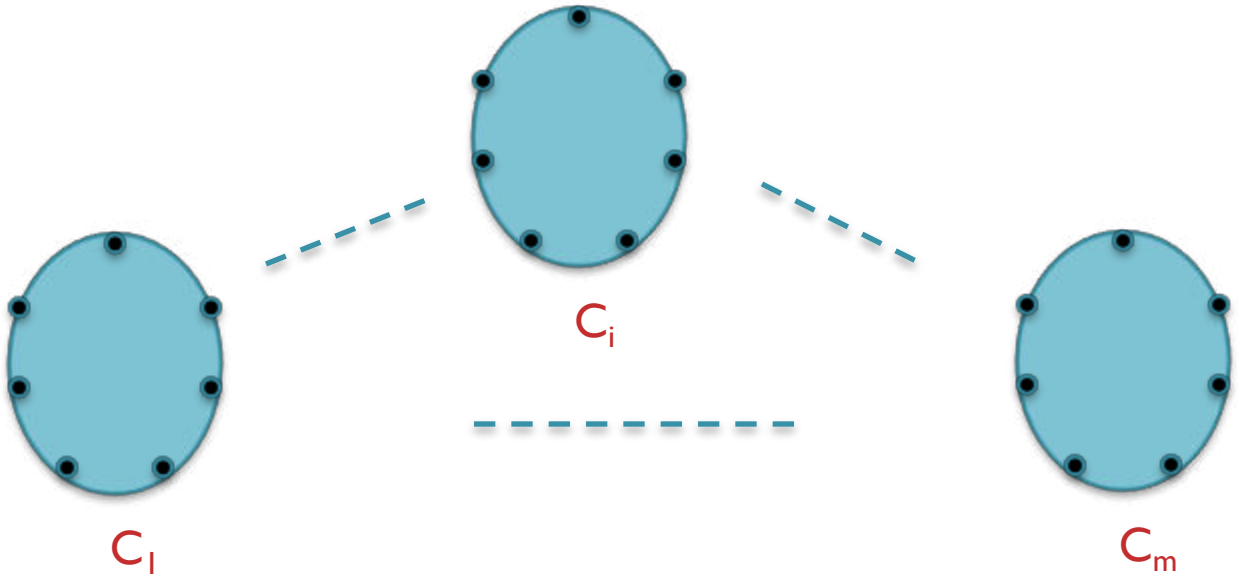
- Reduction: Let ϕ be a 3CNF with m clauses and n variables. Assume, every clause has exactly 3 literals.



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

Example 1: Independent Set

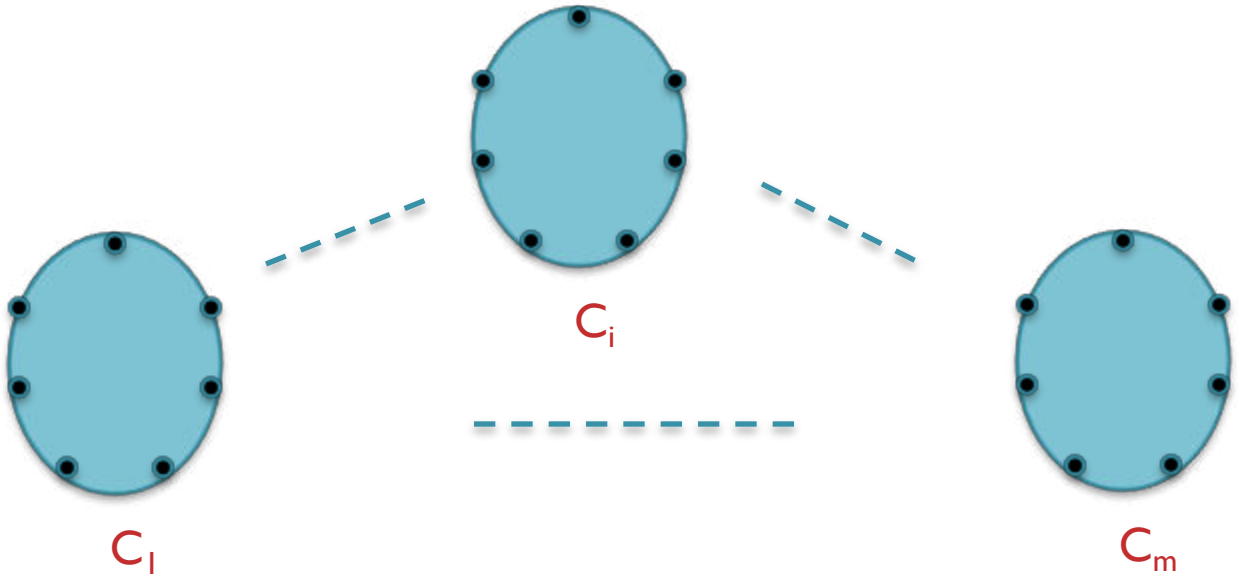
- Reduction: Let ϕ be a 3CNF with m clauses and n variables. Assume, every clause has exactly 3 literals.



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.

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

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

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

Example 3: Vertex Cover

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

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

$$x \in \text{INDSET} \iff f(x) \in \text{VCover}$$

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

$$(G, k) \in \text{INDSET} \iff (G, n\text{-}k) \in \text{VCover}$$

Example 4: 0/1 Integer Programming

- 0/1 IProg := Set of satisfiable 0/1 integer programs
- A 0/1 integer program 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\U), U ⊆ V. Size of a cut (U, V\U) is the number of edges from U to V\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) ∈ 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\U), U ⊆ V. Size of a cut (U, V\U) is the number of edges from U to V\U.

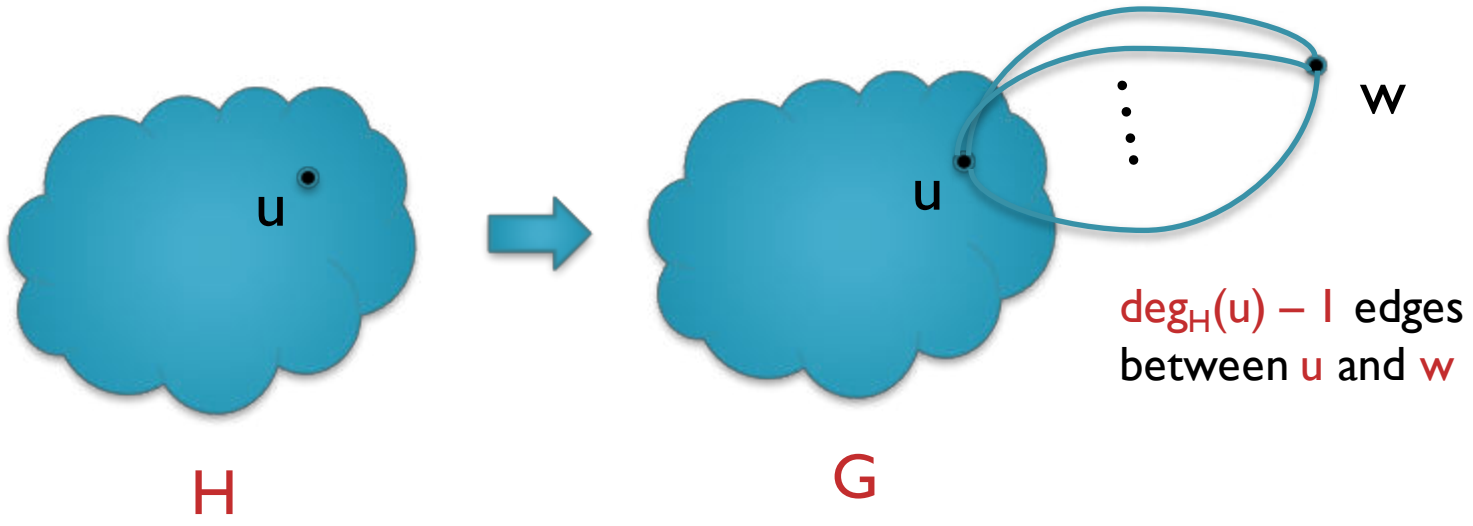
- Goal: A poly-time reduction from MinVCover to MaxCut.

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

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

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

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

- Let $S_G(U)$ = no. of edges in G with <u>exactly one</u> 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\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\U + w) is a cut in G.

- Let $S_G(U)$ = size of the cut (U, V\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\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 at least one 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 $S_G(U) = S_H(U) + \sum_{u \in U} (\deg_H(u) - 1)$

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

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

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

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

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

  ➡ $S_G(U) = |\text{MaxCut}(G)| = 2.|E(H)| - |U|$

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

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

  U must be a minVCover in H

  ➡ $S_G(U) = |\text{MaxCut}(G)| = 2.|E(H)| - |U|$

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.

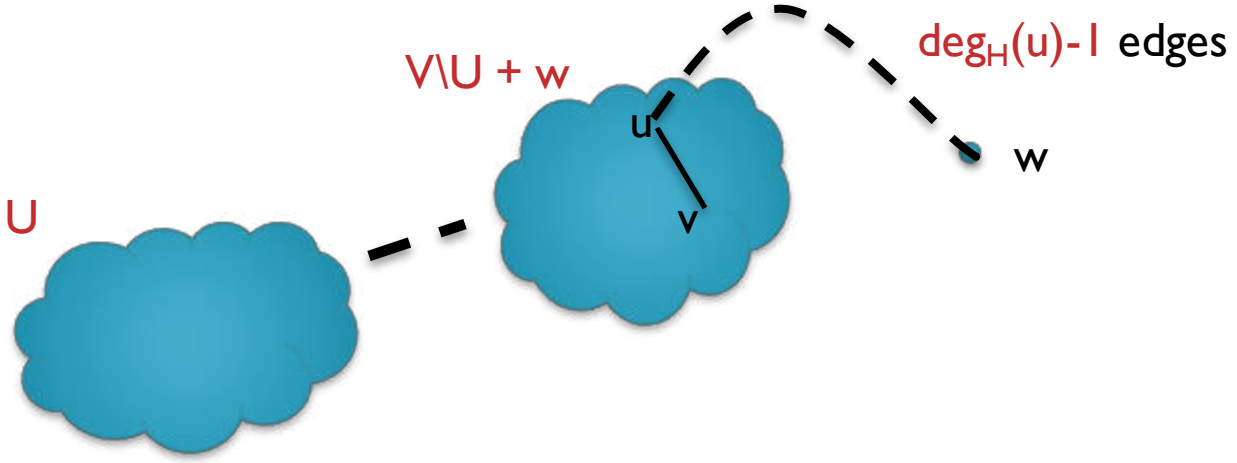
- Then $S_G(U) = 2.|E_H(U)| - |U|$ … Eqn (1)

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

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

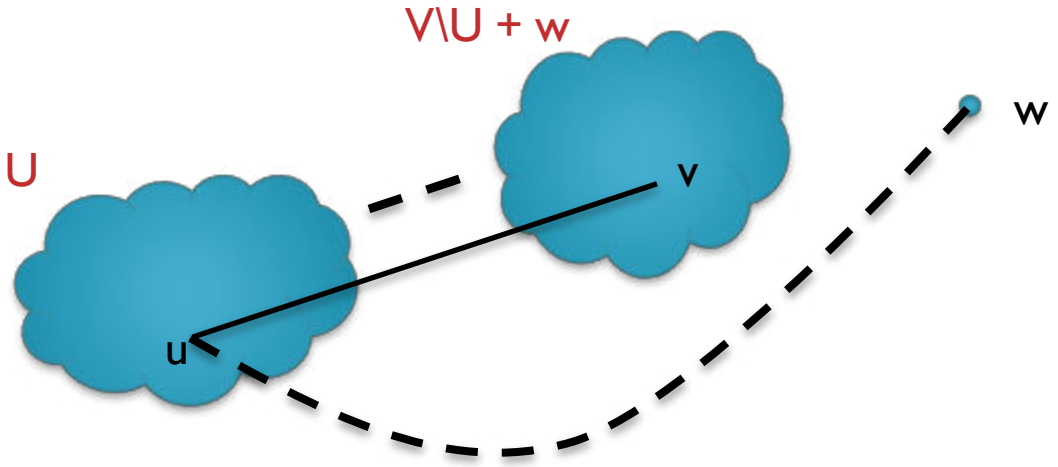
Example 5: Max Cut

- 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 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 $\Longrightarrow$ 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 ⟹ search) We'll prove this for L = SAT first.

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.

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

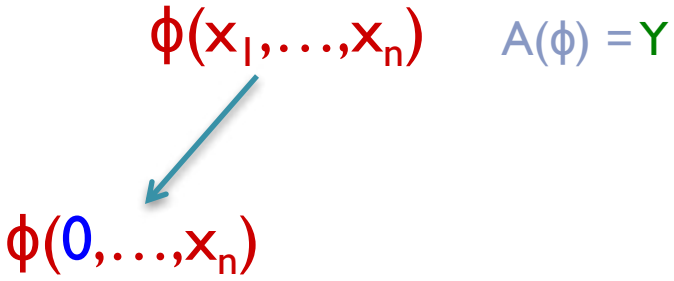
SAT is *downward self-reducible*

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

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

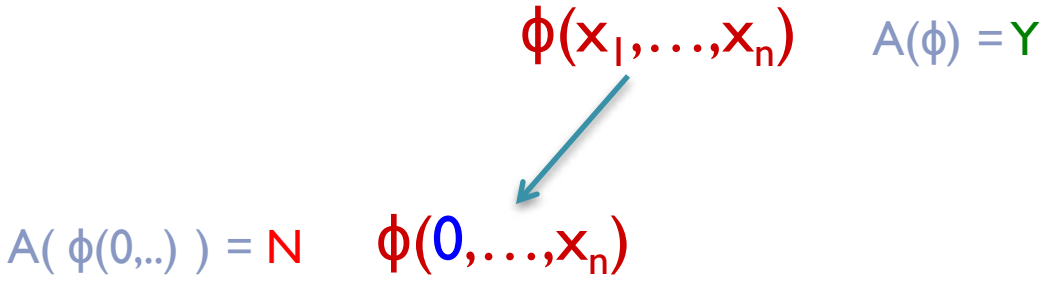
SAT is *downward self-reducible*

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



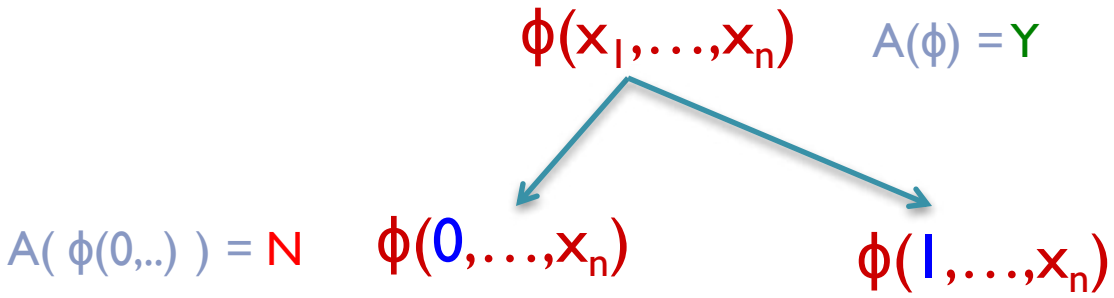
SAT is *downward self-reducible*

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



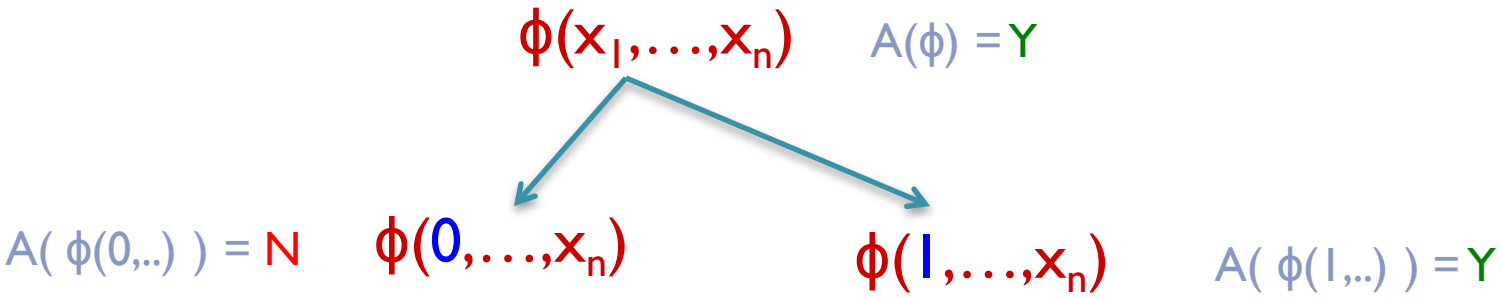
SAT is *downward self-reducible*

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



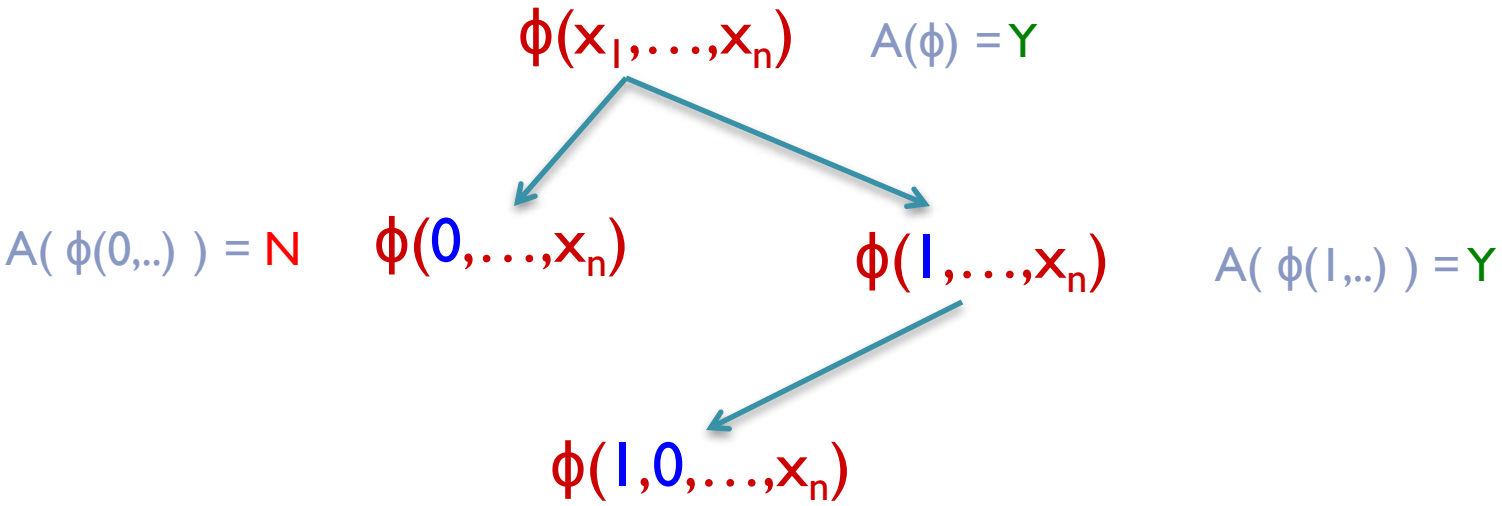
SAT is *downward self-reducible*

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



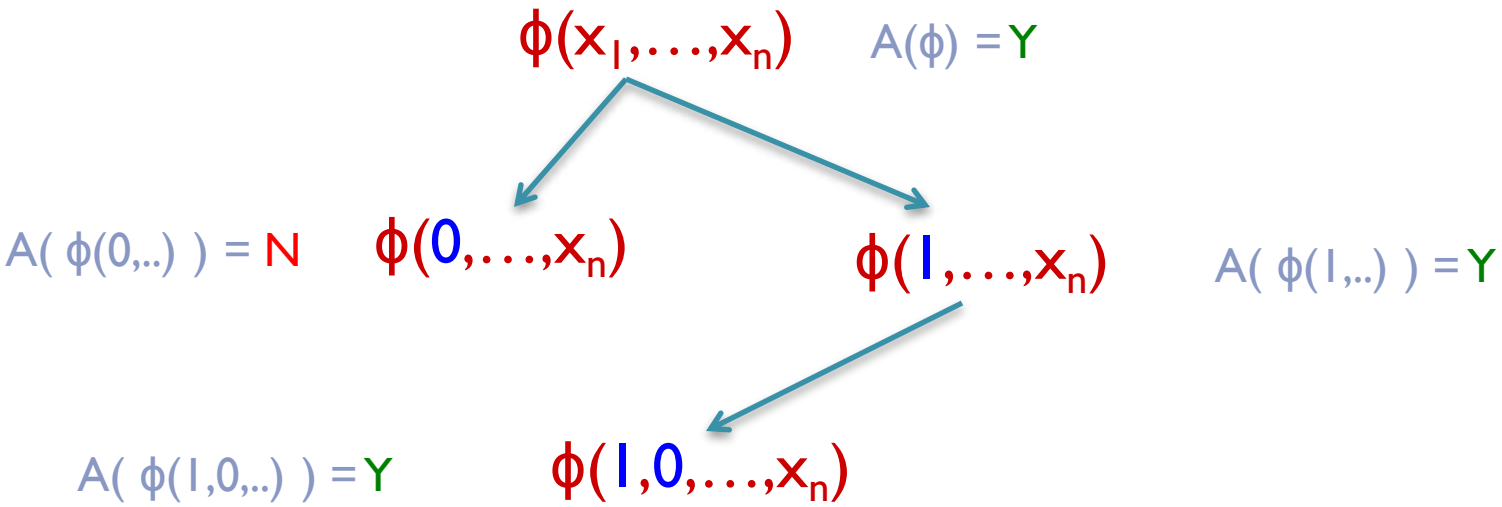
SAT is *downward self-reducible*

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



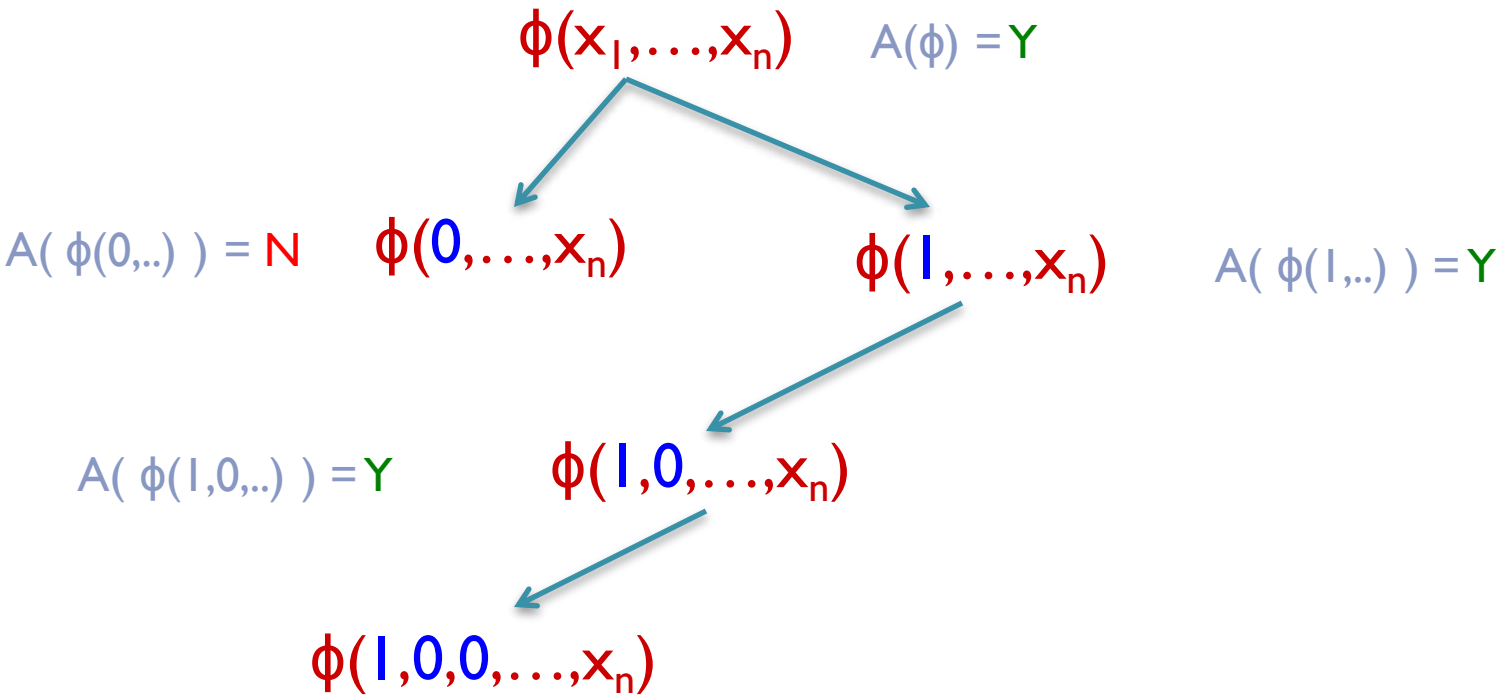
SAT is *downward self-reducible*

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



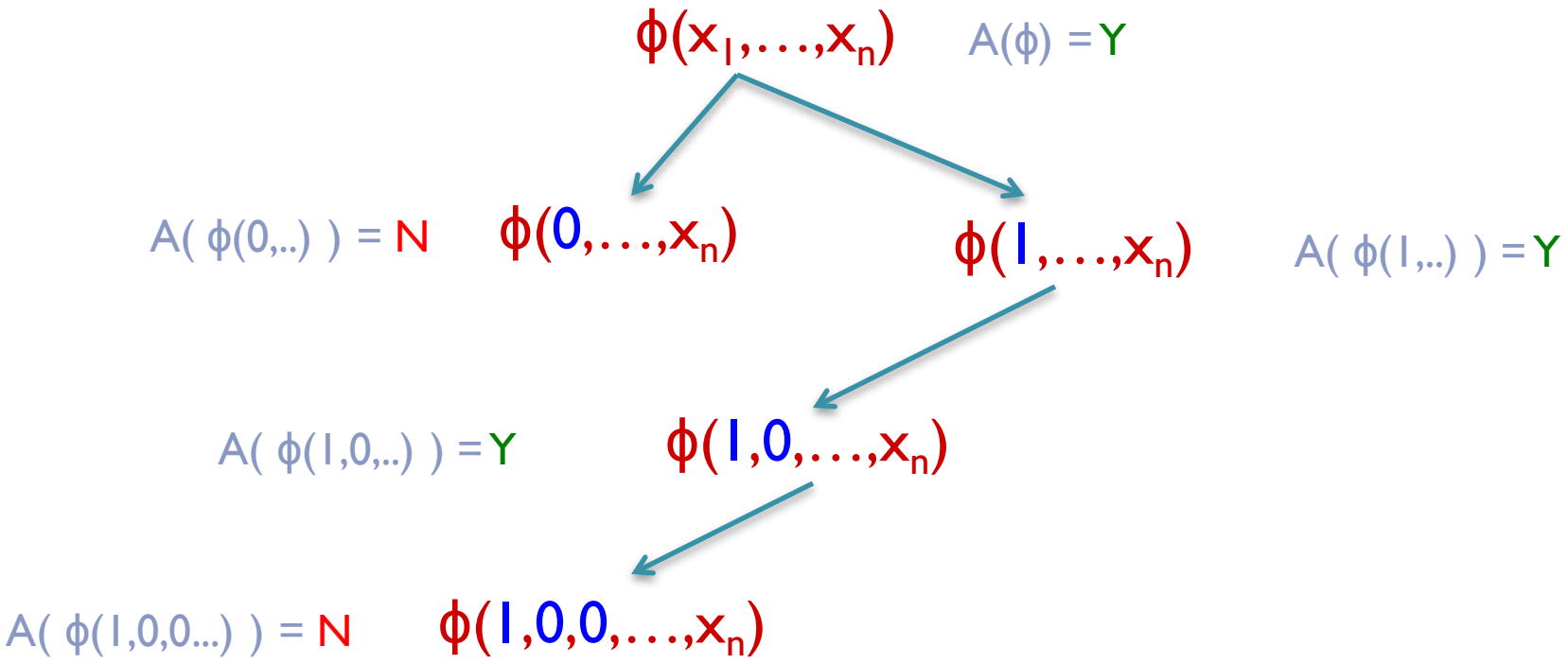
SAT is *downward self-reducible*

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



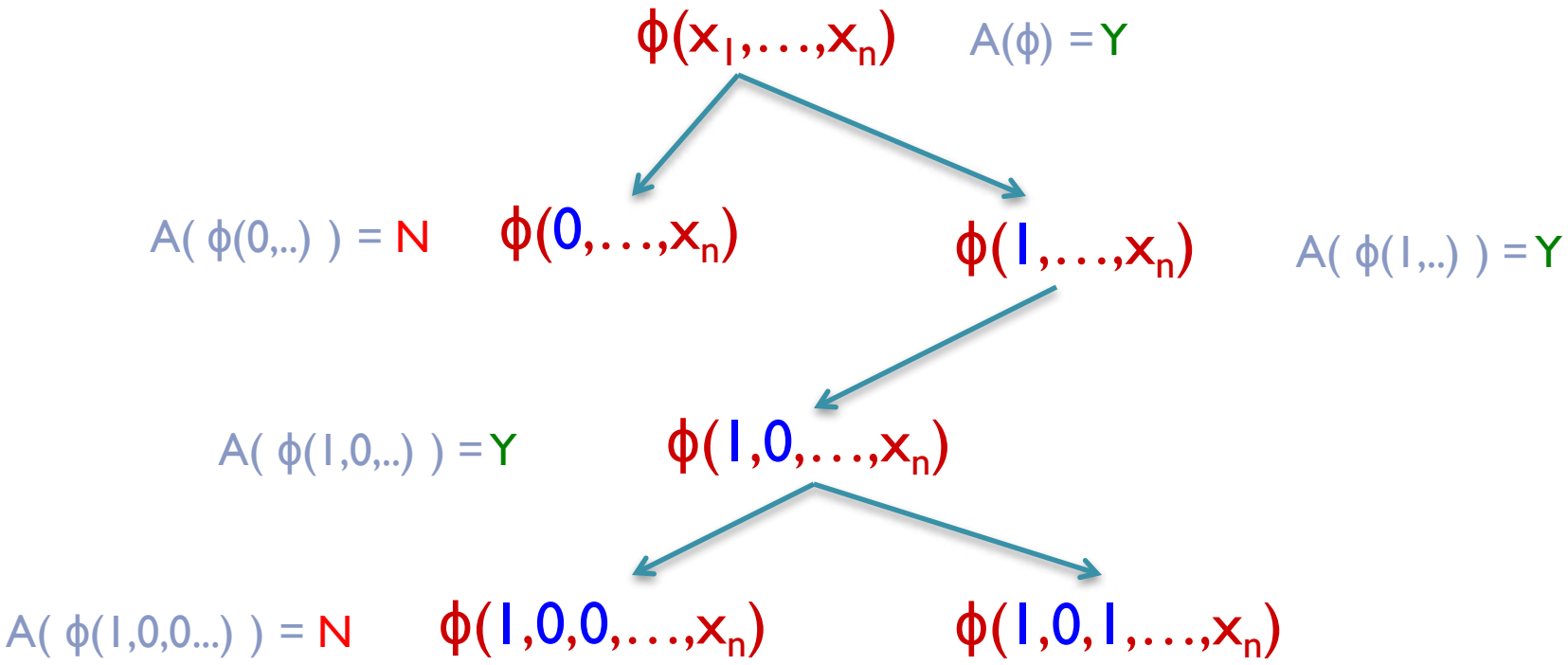
SAT is *downward self-reducible*

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



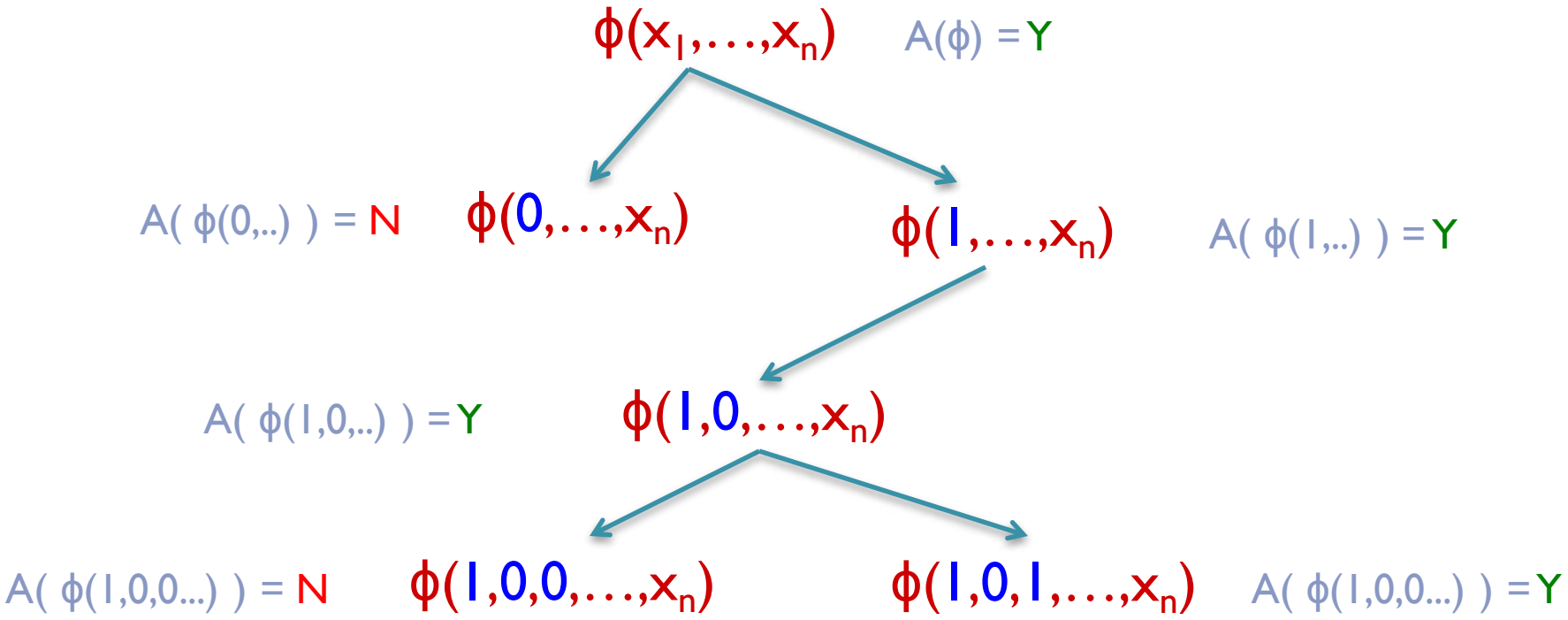
SAT is *downward self-reducible*

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



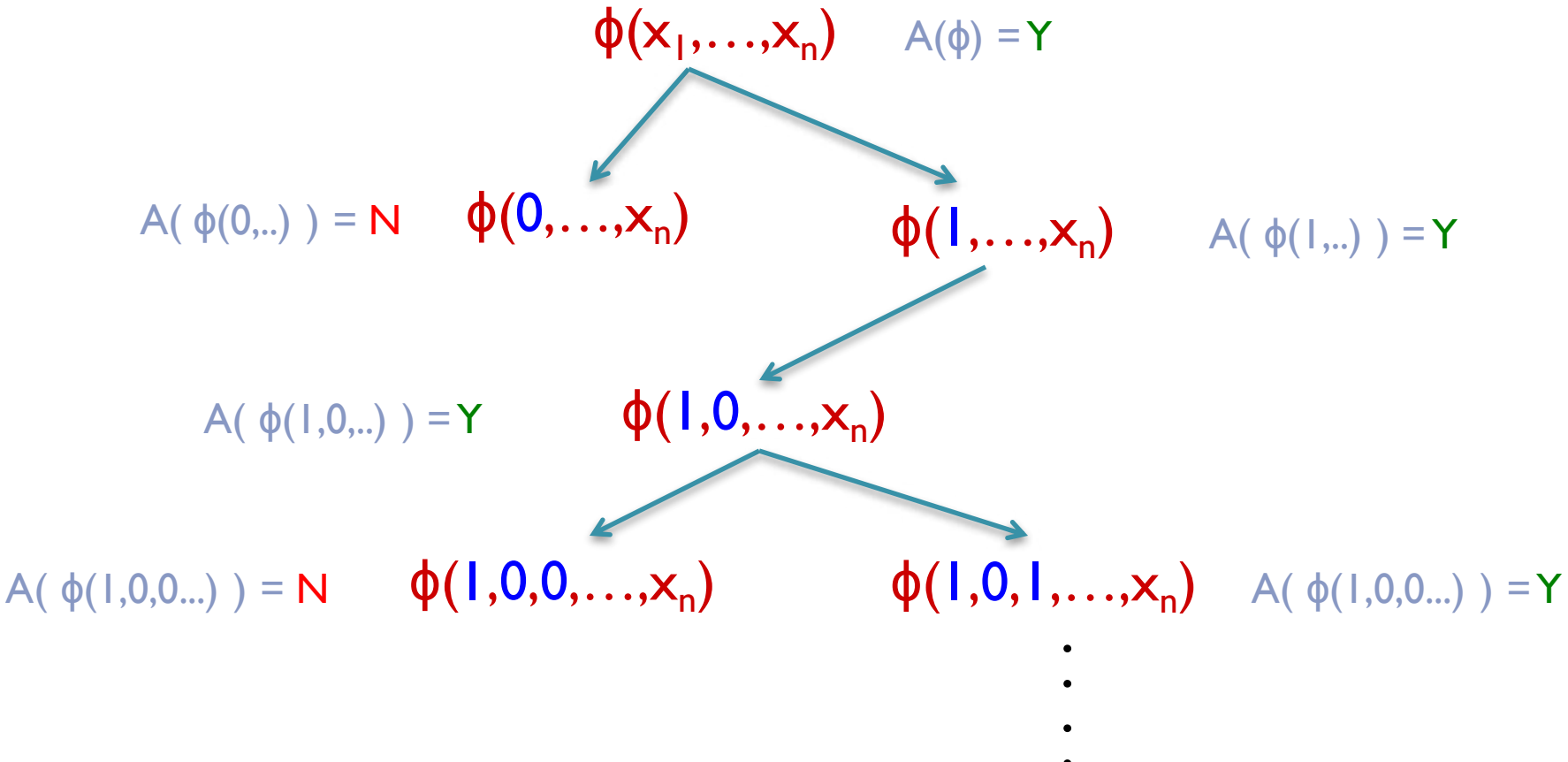
SAT is *downward self-reducible*

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



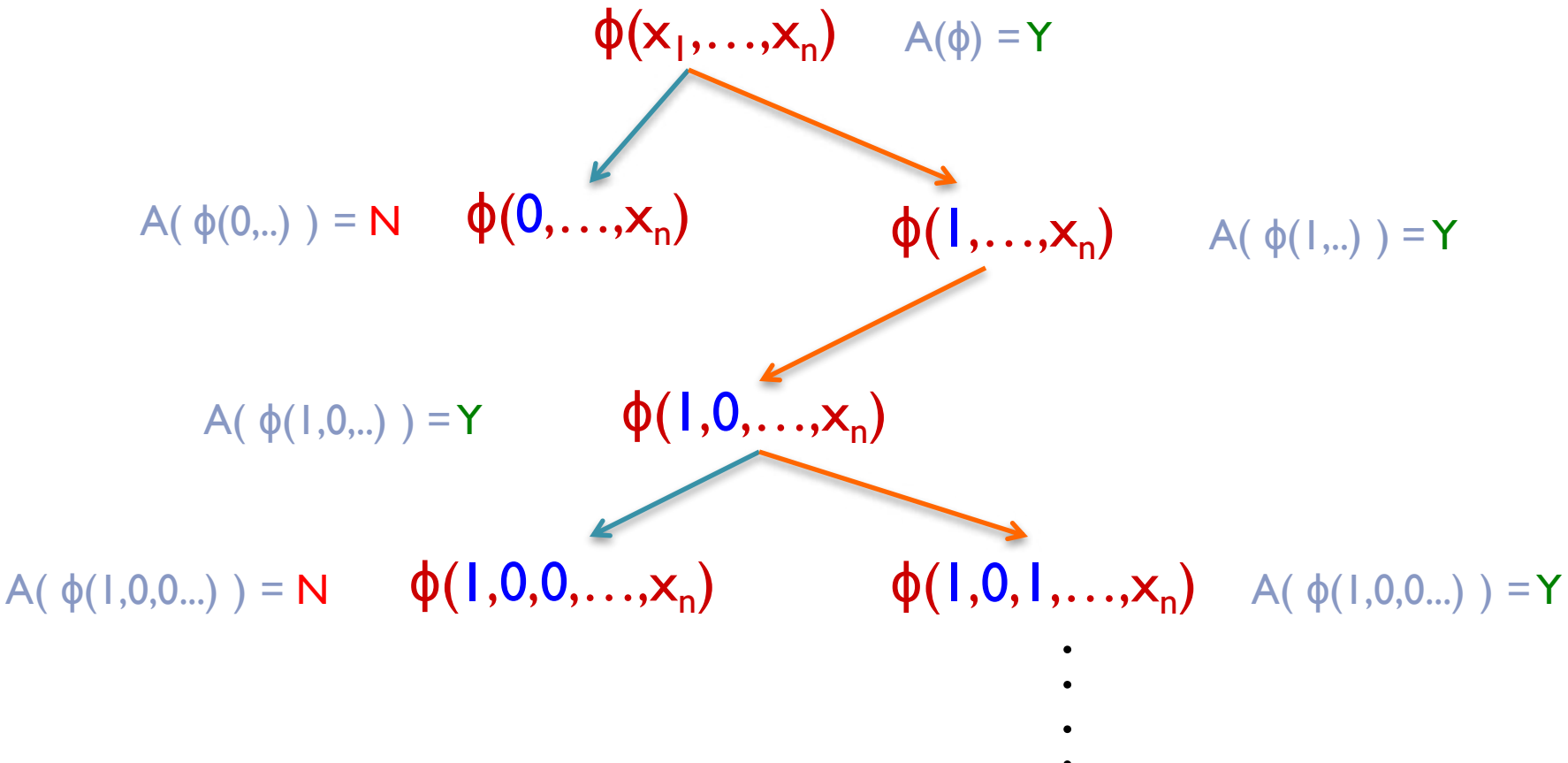
SAT is *downward self-reducible*

- 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.



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, 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 $x \in L$.

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

Decision ≡ Search for NPC problems

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

$SAT \leq_p L$

$L \leq_p SAT$

$x \longmapsto \phi_x$

Decision ≡ Search for NPC problems

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

$SAT \leq_p L$

$L \leq_p SAT$

$x \longmapsto \phi_x$

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

Decision ≡ Search for NPC problems

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

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

$x \longmapsto \phi_x$

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

Decision ≡ Search for NPC problems

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

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

$x \longmapsto \phi_x$

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

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

Decision ≡ Search for NPC problems

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

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

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

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

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

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?

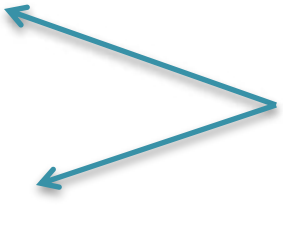
Probably not!

Decision versus Search

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

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

  $NEE = \bigcup_{c \geq 0} NTIME(2^{2^{cn}})$



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 ≠ 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 ≠ 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 *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 .

Will be called an Oracle 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.

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

- 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 .

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.

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

- 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 .

Homework: Read about **Levin reduction**