



Computational Complexity Theory

Lecture 5: Max-Cut is NP-hard; Decision vs. Search

Department of Computer Science,
Indian Institute of Science

Recap: 3SAT is NP-complete

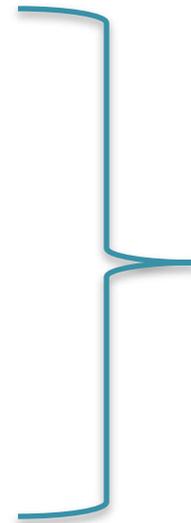
- **Definition.** A CNF is called a **k-CNF** if every clause has at most **k** literals.

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

- **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: More NP complete problems

- **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

Recap: More NP complete problems

- **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 **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 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 a graph H , find a vertex cover in H that has the min size.
- **Obs**: From **MinVCover**(H), we can readily check if $(H, k) \in \text{VCover}$, for any k .

Example 5: Max Cut

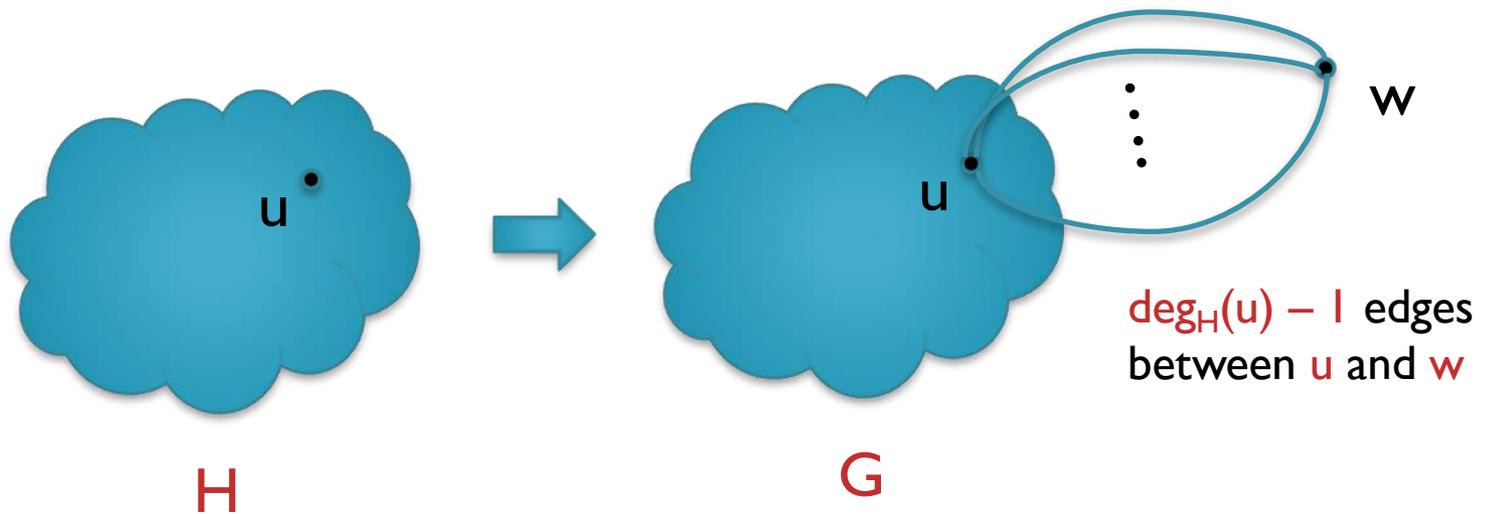
- **MaxCut** : Given a graph find a cut with the max size.
- A *cut* of $G = (V, E)$ is a tuple $(U, V \setminus U)$, $U \subseteq V$. Size of a cut $(U, V \setminus U)$ is the number of edges from U to $V \setminus U$.
- **Goal**: A poly-time reduction from **MinVCover** to **MaxCut**.



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

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: $|\text{MaxCut}(G)| = 2 \cdot |E(H)| - |\text{MinVCover}(H)|$

Example 5: Max Cut

- **Claim:** $|\text{MaxCut}(G)| = 2 \cdot |E(H)| - |\text{MinVCover}(H)|$
- **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:** $|\text{MaxCut}(G)| = 2 \cdot |E(H)| - |\text{MinVCover}(H)|$
- **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 in G with exactly one end vertex incident on a vertex in U .

Example 5: Max Cut

- **Claim:** $|\text{MaxCut}(G)| = 2 \cdot |E(H)| - |\text{MinVCover}(H)|$
- **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:** $|\text{MaxCut}(G)| = 2 \cdot |E(H)| - |\text{MinVCover}(H)|$
- **Proof:** Let $V(H) = V$. Then $V(G) = V + w$.
Suppose $(U, V \setminus U + w)$ is a cut in G .
- Let $S_G(U) = \text{size of the cut } (U, V \setminus U + w)$.

Example 5: Max Cut

- **Claim:** $|\text{MaxCut}(G)| = 2 \cdot |E(H)| - |\text{MinVCover}(H)|$
- **Proof:** Let $V(H) = V$. Then $V(G) = V + w$.
Suppose $(U, V \setminus U + w)$ is a cut in G .
- Let $S_H(U) :=$ no. of edges in H with exactly one end vertex incident on a vertex in U .

Example 5: Max Cut

- **Claim:** $|\text{MaxCut}(G)| = 2 \cdot |E(H)| - |\text{MinVCover}(H)|$
- **Proof:** Let $V(H) = V$. Then $V(G) = V + w$.
Suppose $(U, V \setminus U + w)$ is a cut in G .

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

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

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

$$= S_H(U) + \sum_{u \in U} \text{deg}_H(u) - |U|$$

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

Example 5: Max Cut

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

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

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

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

$$= 2 \cdot |E_H(U)| - |U|$$

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

Example 5: Max Cut

- **Claim:** $|\text{MaxCut}(G)| = 2 \cdot |E(H)| - |\text{MinVCover}(H)|$

- **Proof:** Let $V(H) = V$. Then $V(G) = V + w$.

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

- Then $S_G(U) = 2 \cdot |E_H(U)| - |U| \quad \dots \text{Eqn (I)}$

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

Example 5: Max Cut

- **Claim:** $|\text{MaxCut}(G)| = 2 \cdot |E(H)| - |\text{MinVCover}(H)|$
- **Proof:** Let $V(H) = V$. Then $V(G) = V + w$.
Suppose $(U, V \setminus U + w)$ is a cut in G .

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

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

Example 5: Max Cut

- **Claim:** $|\text{MaxCut}(G)| = 2 \cdot |E(H)| - |\text{MinVCover}(H)|$
- **Proof:** Let $V(H) = V$. Then $V(G) = V + w$.
Suppose $(U, V \setminus U + w)$ is a cut in G .

- Then $S_G(U) = 2 \cdot |E_H(U)| - |U|$... Eqn (I)

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

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

Example 5: Max Cut

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

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

- Then $S_G(U) = 2 \cdot |E_H(U)| - |U|$... Eqn (I)

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

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

Example 5: Max Cut

- **Claim:** $|MaxCut(G)| = 2 \cdot |E(H)| - |MinVCover(H)|$

- **Proof:** Let $V(H) = V$. Then $V(G) = V + w$.

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

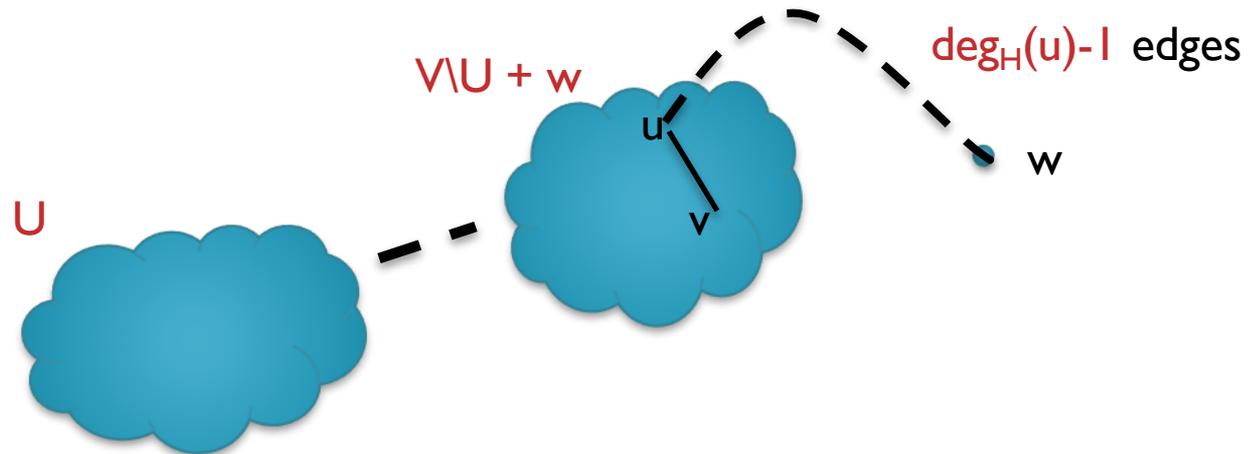
- Then $S_G(U) = 2 \cdot |E_H(U)| - |U| \quad \dots \text{Eqn (I)}$

- **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

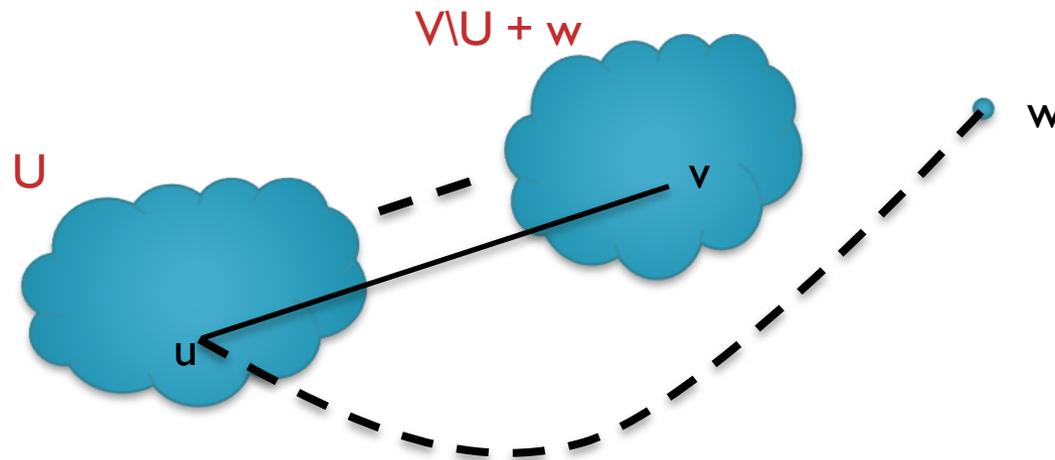
Example 5: Max Cut

- **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 search version of L can be solved in poly-time if and only if 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  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 $L = \text{SAT}$ first.

SAT is downward self-reducible

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

SAT is *downward self-reducible*

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

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

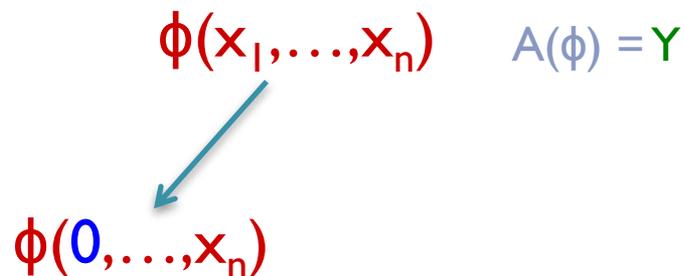
SAT is *downward self-reducible*

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

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

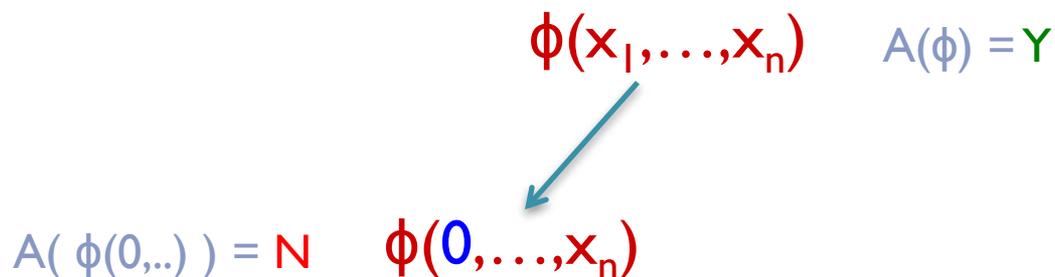
SAT is *downward self-reducible*

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



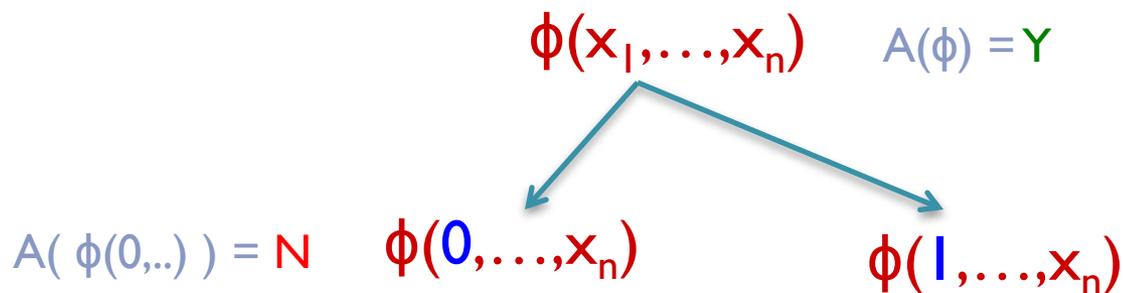
SAT is downward self-reducible

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



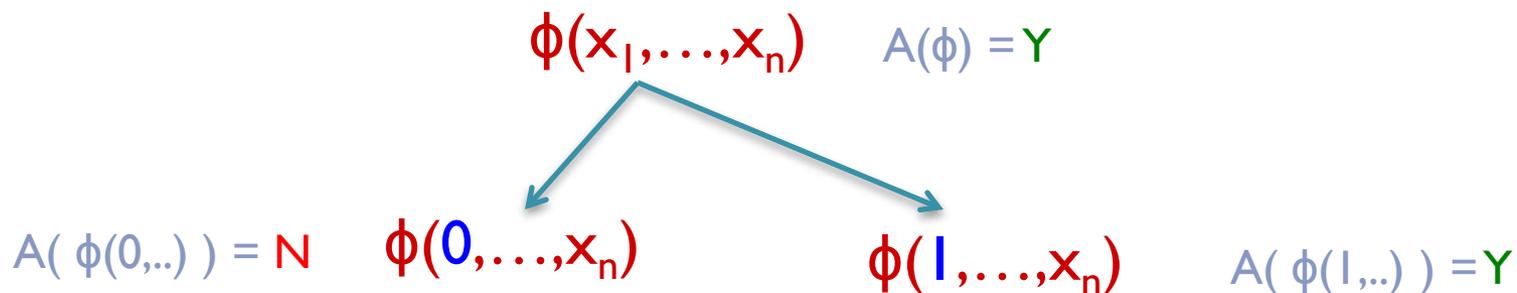
SAT is downward self-reducible

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



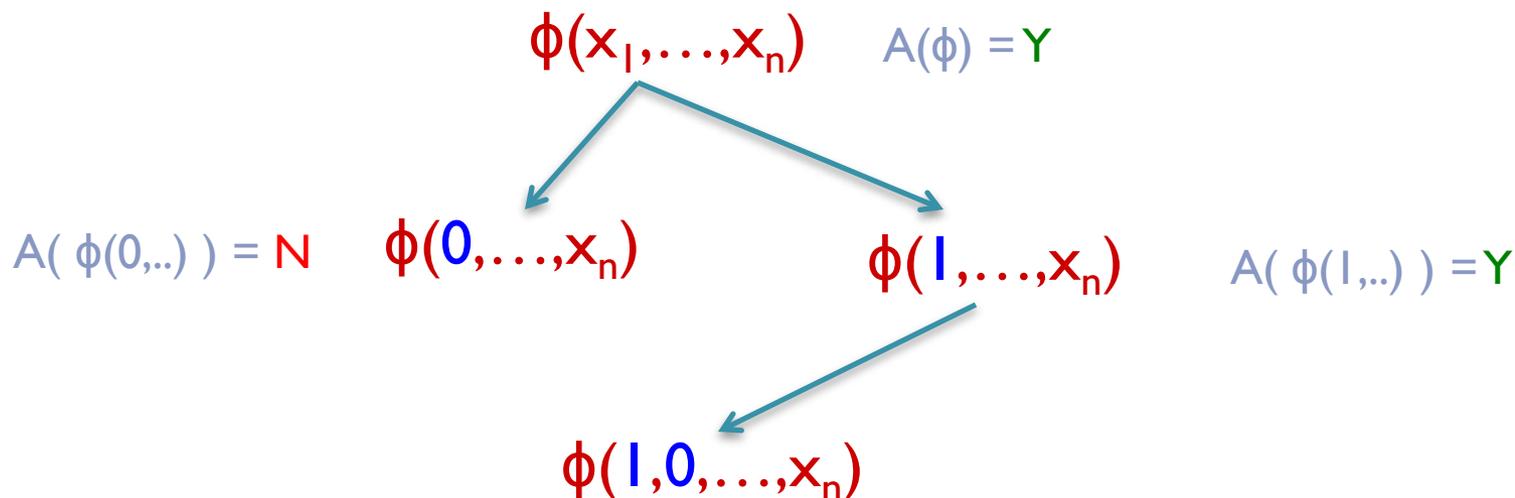
SAT is downward self-reducible

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



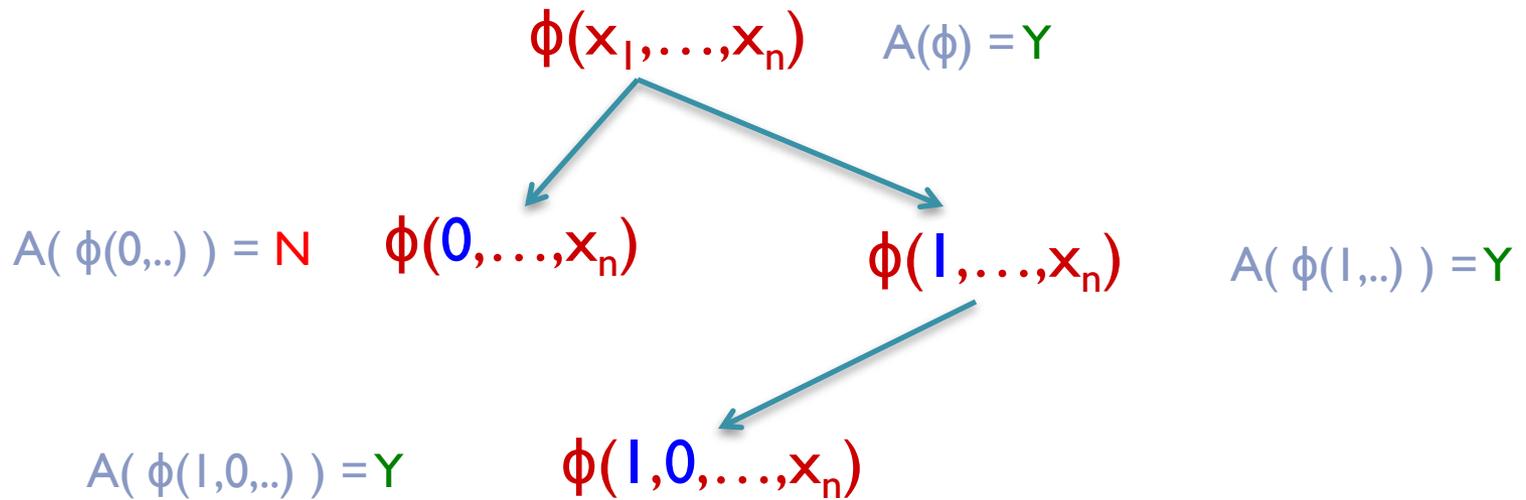
SAT is downward self-reducible

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



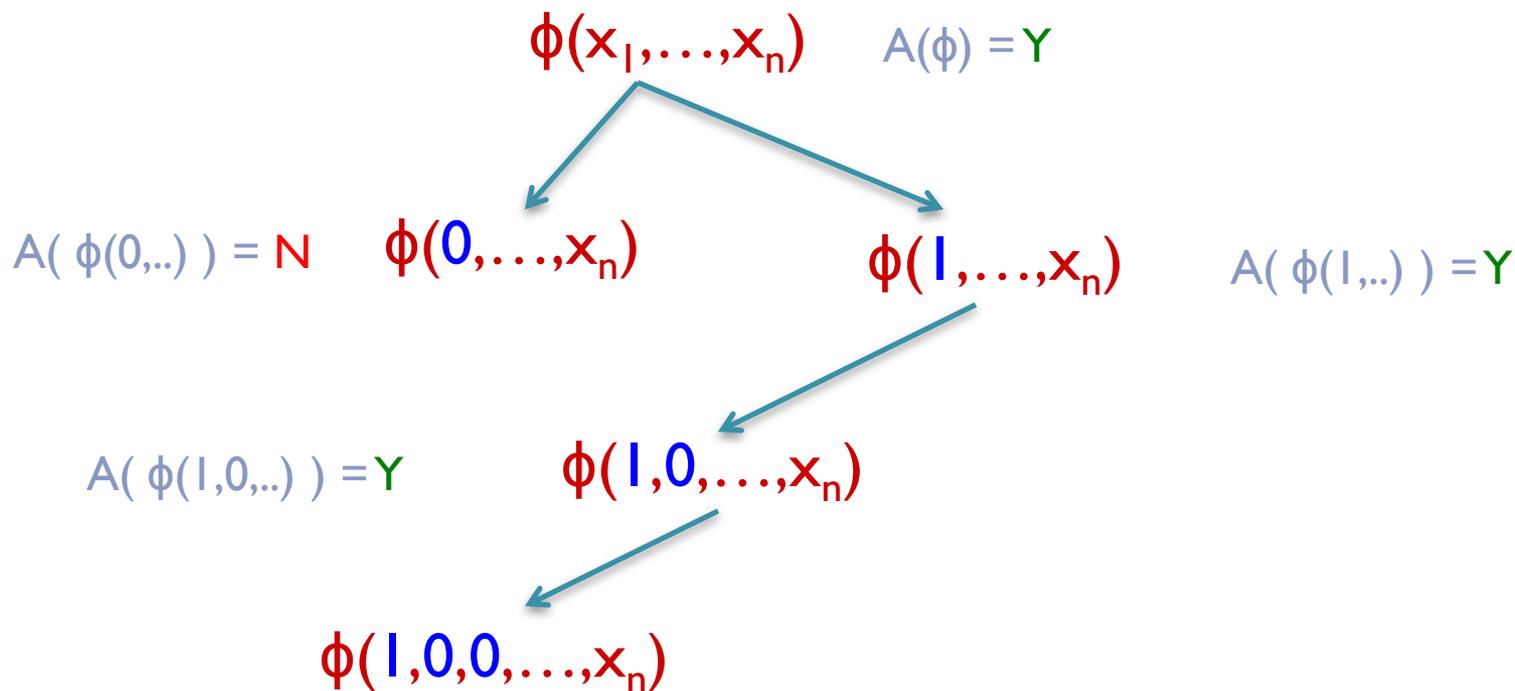
SAT is downward self-reducible

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



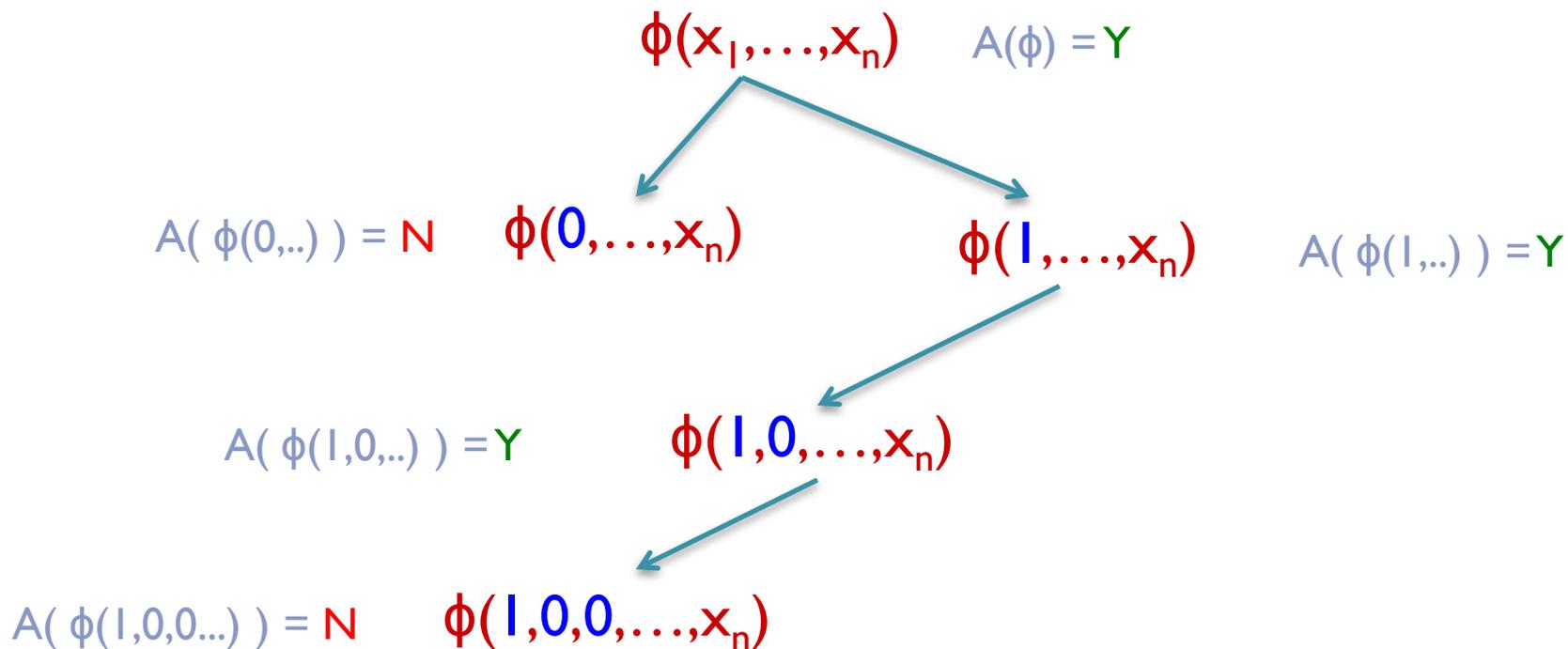
SAT is downward self-reducible

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



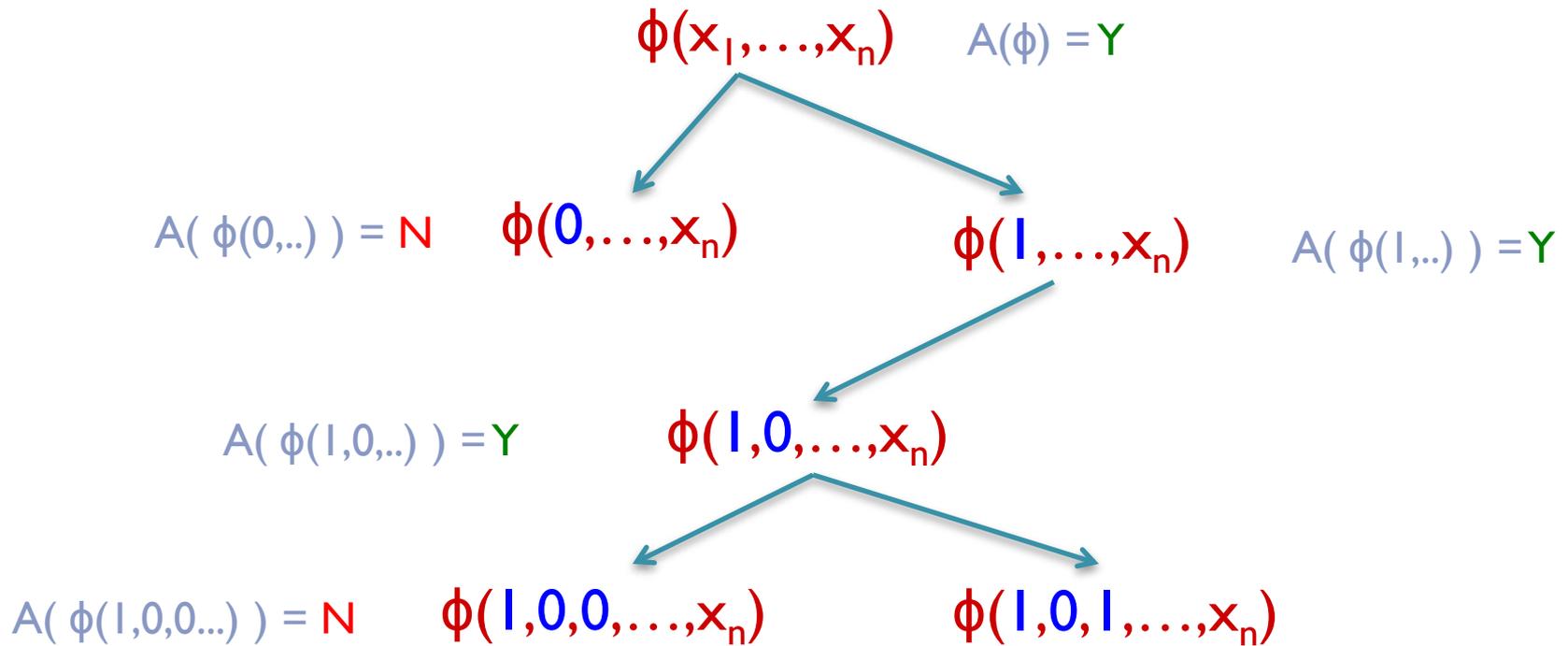
SAT is downward self-reducible

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



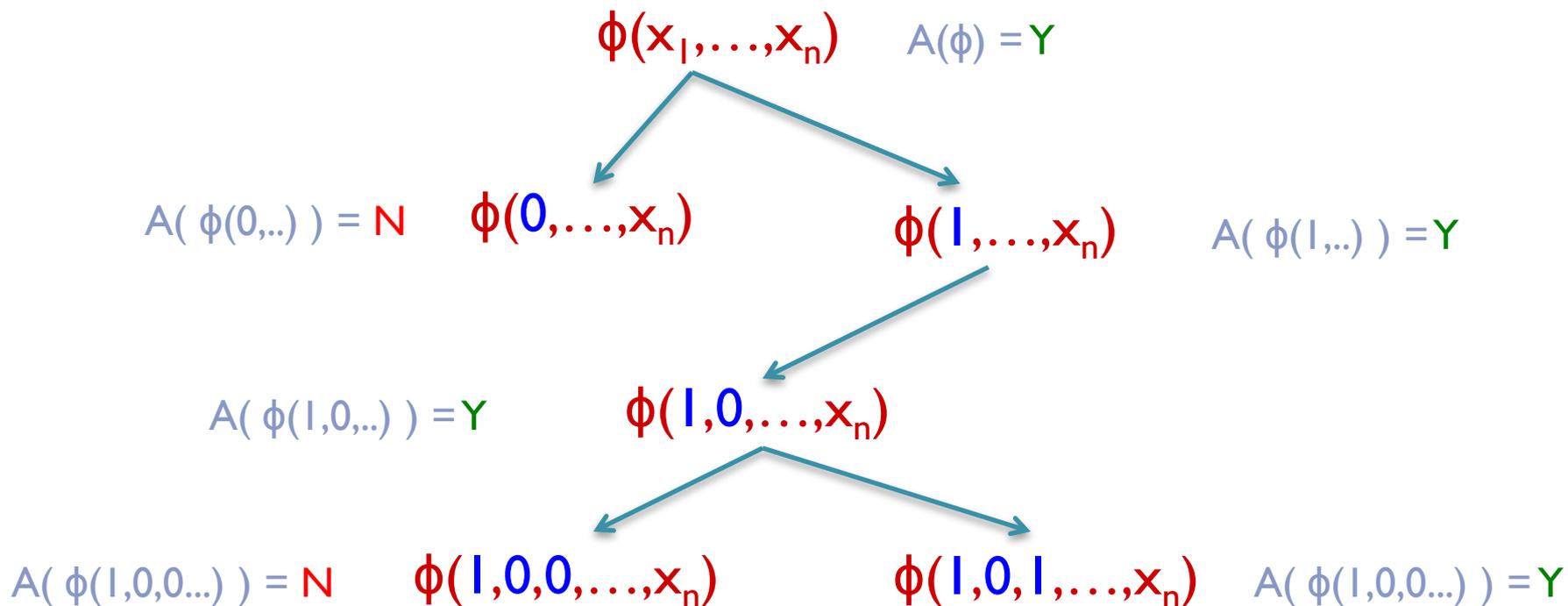
SAT is downward self-reducible

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



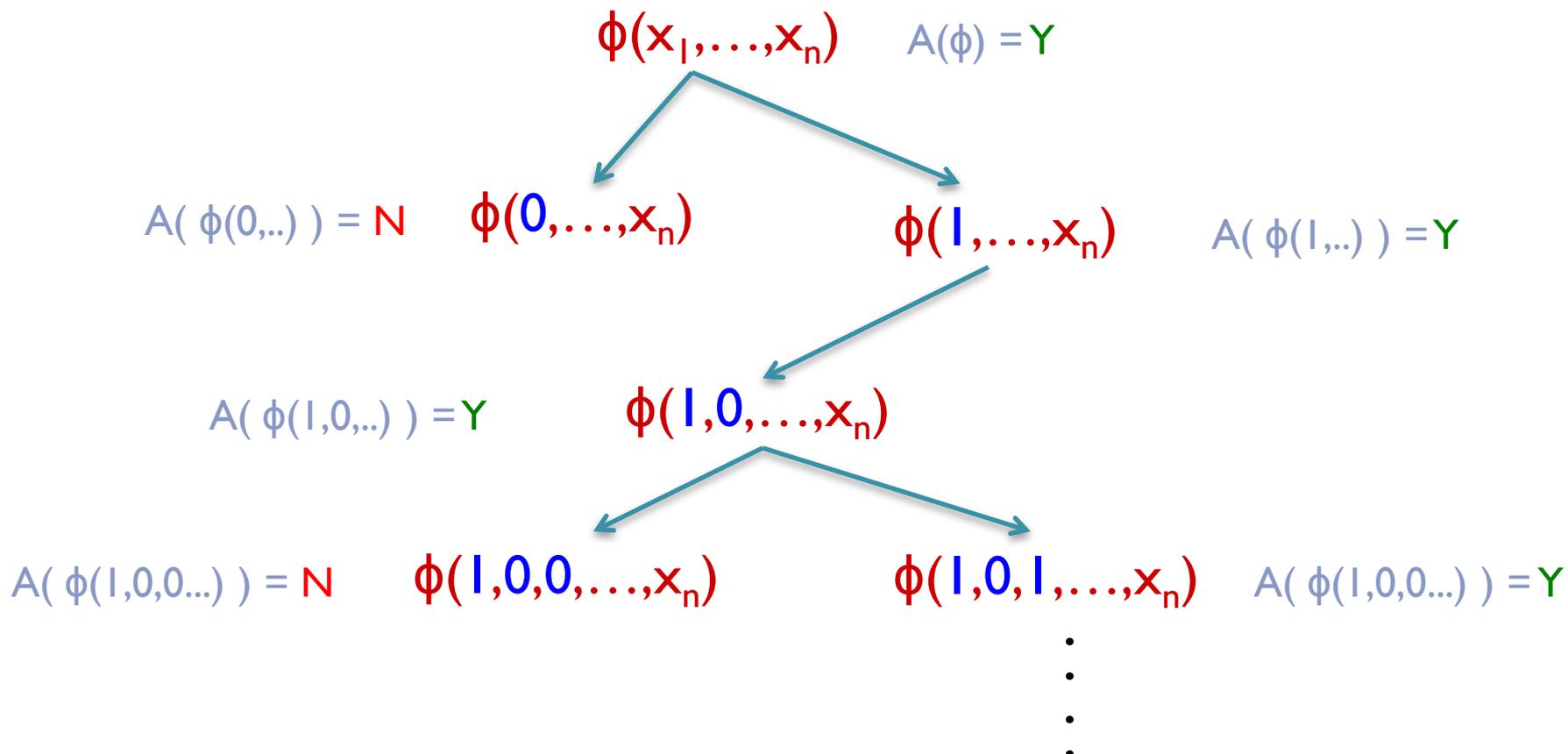
SAT is downward self-reducible

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



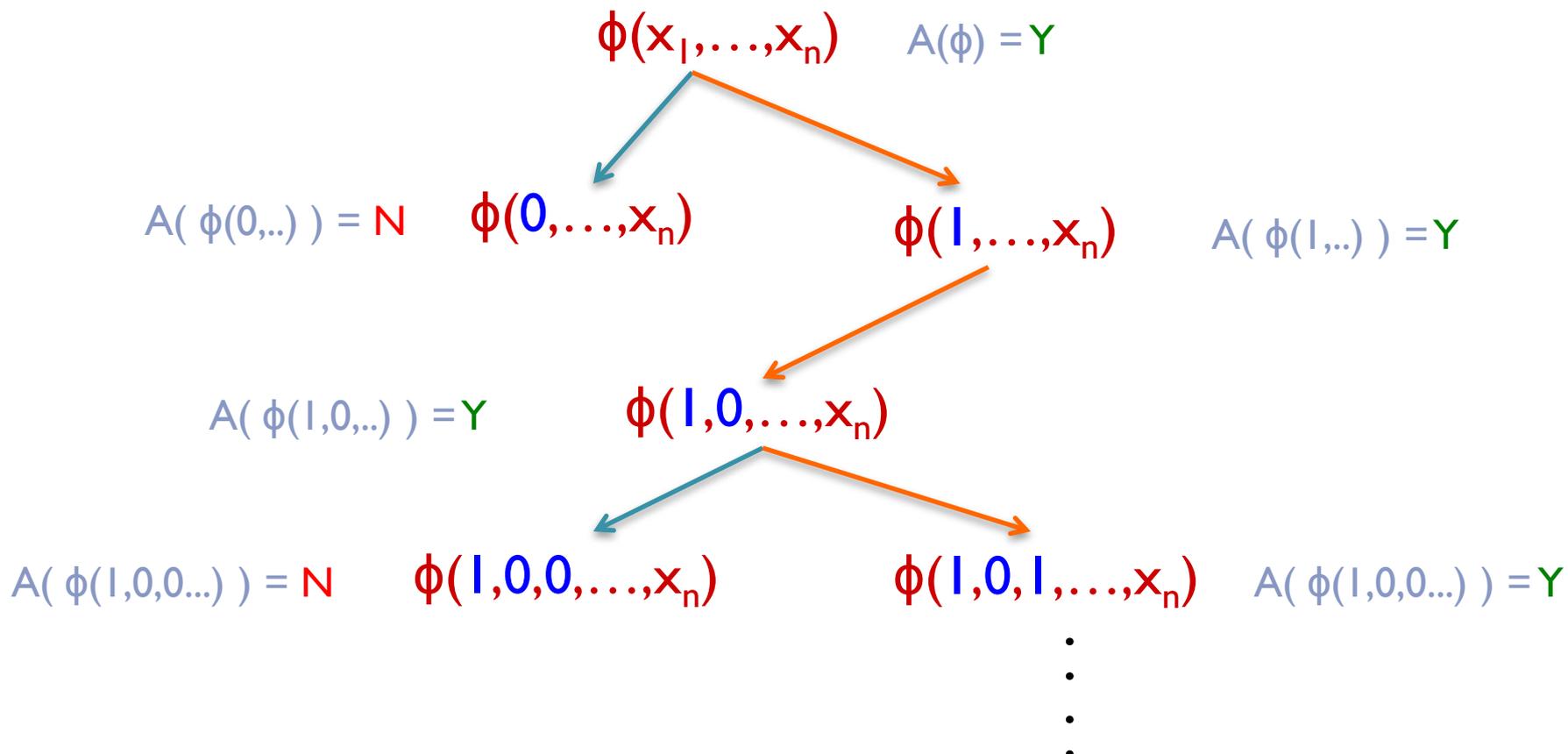
SAT is downward self-reducible

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



SAT is downward self-reducible

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



SAT is *downward self-reducible*

- **Proof.** (decision  search) Let $L = \text{SAT}$, and A be a poly-time algorithm to decide if $\phi(x_1, \dots, x_n)$ is satisfiable.
- We can find a satisfying assignment of ϕ with at most $2n$ calls to A .

Decision \equiv Search for NPC problems

- **Proof.** (decision \rightarrow 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 \equiv Search for NPC problems

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

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

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

Decision \equiv 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 $x \in L$.

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

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

$$x \longmapsto \phi_x$$

Decision \equiv 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 $x \in L$.

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

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

$$x \longmapsto \phi_x$$

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 \equiv 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 $x \in L$.

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

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

$$x \longmapsto \phi_x$$

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

Decision \equiv 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 $x \in L$.

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

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

$$x \longmapsto \phi_x$$

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 \equiv 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 $x \in L$.

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

$$\phi \longmapsto f(\phi)$$

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

$$x \longmapsto \phi_x$$

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

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!

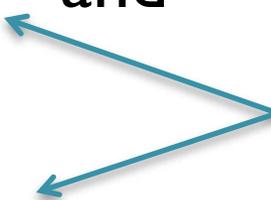
Decision versus Search

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

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

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

Doubly exponential analogues of P and NP



- Class $\text{NTIME}(T(n))$ 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.
- 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**