Computational Complexity Theory

Lecture 6: Decision vs. Search; NTMs

Department of Computer Science,
Indian Institute of Science

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 $x \leq c$ 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 ∈ [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 ≤ s that computes f.

- Easy to see that MCSP is in NP.

- Is MCSP NP-complete? Not known!
- Multi-output MCSP & Partial fn. MCSP are NP-hard under poly-time randomized reductions.

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 $\Longrightarrow$ 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 $\Longrightarrow$ search) We'll prove this for L = 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 $\Rightarrow$ 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 $\longrightarrow$ search) Let L = SAT, and A be a poly-time algorithm to decide if $\phi(x_1,\ldots,x_n)$ is satisfiable.

$$\begin{array}{lll} & \phi(x_1,\ldots,x_n) & A(\phi) = Y \\ \swarrow & & \\ \phi(0,\ldots,x_n) & & \end{array}$$

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.

$$\begin{tikzcd}
 & \phi(x_1,\ldots,x_n) \arrow[dl] \arrow[dr] & A(\phi) = Y \\
A(\phi(0,..)) = N \quad \phi(0,\ldots,x_n) & & \phi(1,\ldots,x_n)
\end{tikzcd}$$

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.

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

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

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

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

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

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

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.

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

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

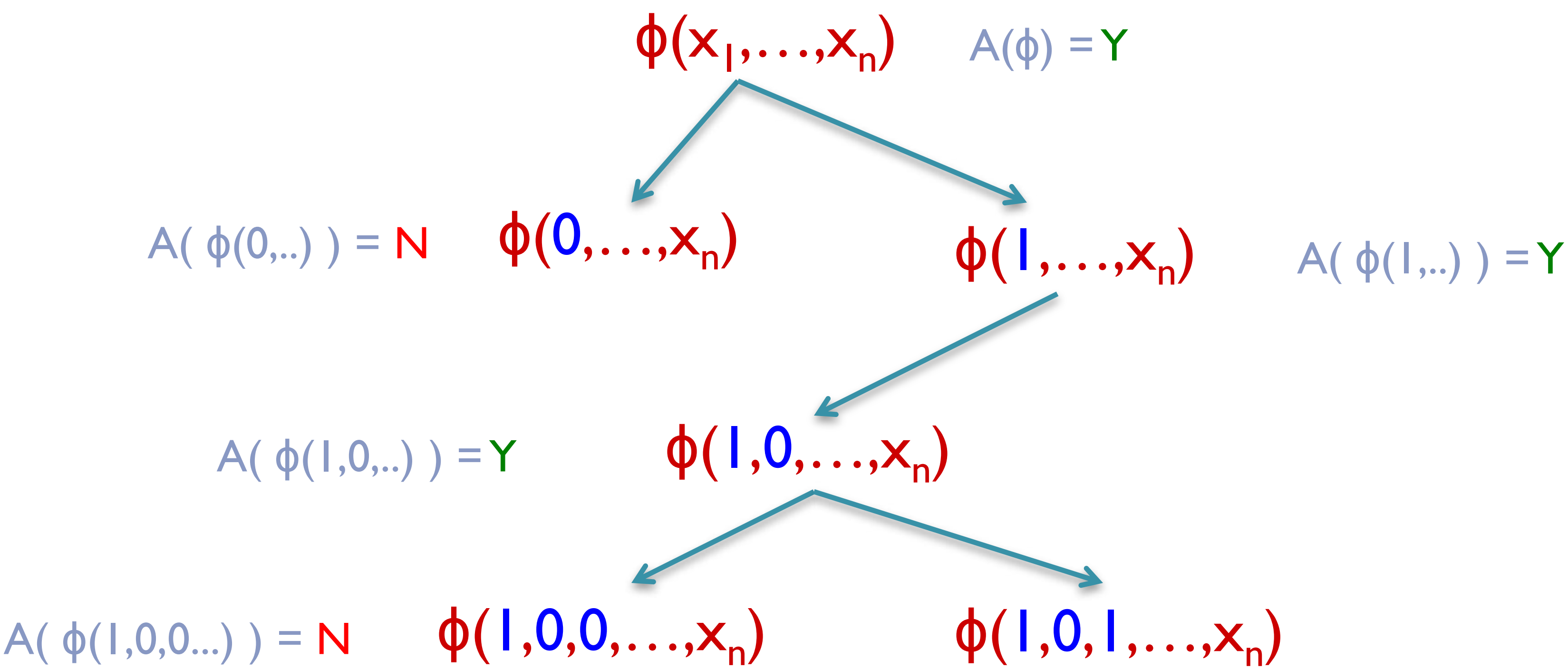
$\phi(1,\ldots,x_n)$ $\quad A(\ \phi(1,..)\ ) = Y$

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

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

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.



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.

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

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

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

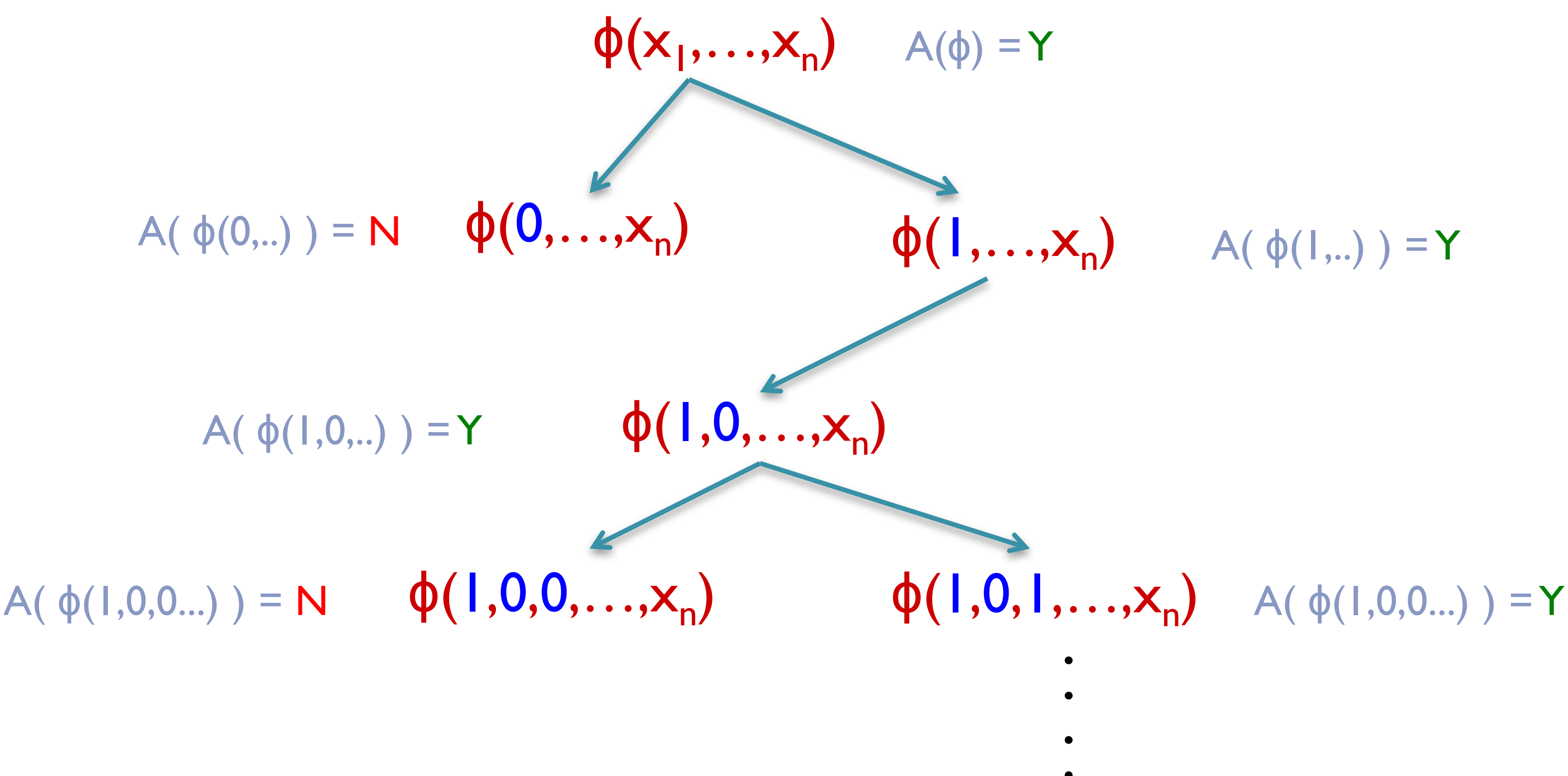
$A(\ \phi(1,0,..)\ ) = Y$ $\phi(1,0,\ldots,x_n)$

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

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

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.



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

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

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

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 $x \in L$.

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

Decision ≡ 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 $x \in L$.

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

$x \longmapsto \phi_x$

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

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

$$x \longmapsto \phi_x$$

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, 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 ≡ 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 <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.

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

- 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 .

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 .

Will be called an Oracle later

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.

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

- 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 .

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**

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*.

also called *nondeterministically*

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 *arbitrarily*.
- Unlike DTMs, NTMs are **not intended to be physically realizable** (because of the arbitrary nature of application of the transition functions).

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 $x \Longleftrightarrow x \in L$
  - On every sequence of applications of the transition functions on input x, M either reaches q_{accept} or q_{halt}.

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 $x \iff x \in L$
  - On every sequence of applications of the transition functions on input x, M either reaches q_{accept} or q_{halt}.

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 $\Longleftrightarrow$ $x \in L$
  - On every sequence of applications of the transition functions on input x, M either reaches q_{accept} or q_{halt} within $T(|x|)$ steps of computation.

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 $\mathrm{NTIME}(T(n))$ 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,

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

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

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

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

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

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

  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. $(|x| = n)$

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

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

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 .