UMB CS420

NP

Monday, April 25, 2022

Announcements

- HW 10 out
  - Due Tuesday 4/26 11:59pm EST

- Hannah Office Hours moved
  - Now Monday 2-3:30pm in-person
  - McCormack, 3rd Floor, Room 0201-33

Last Time: 3 Problems in **P**

- A Graph Problem:

  $PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

  “search” problem

- A Number Problem:

  $RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\}$

- A CFL Problem:

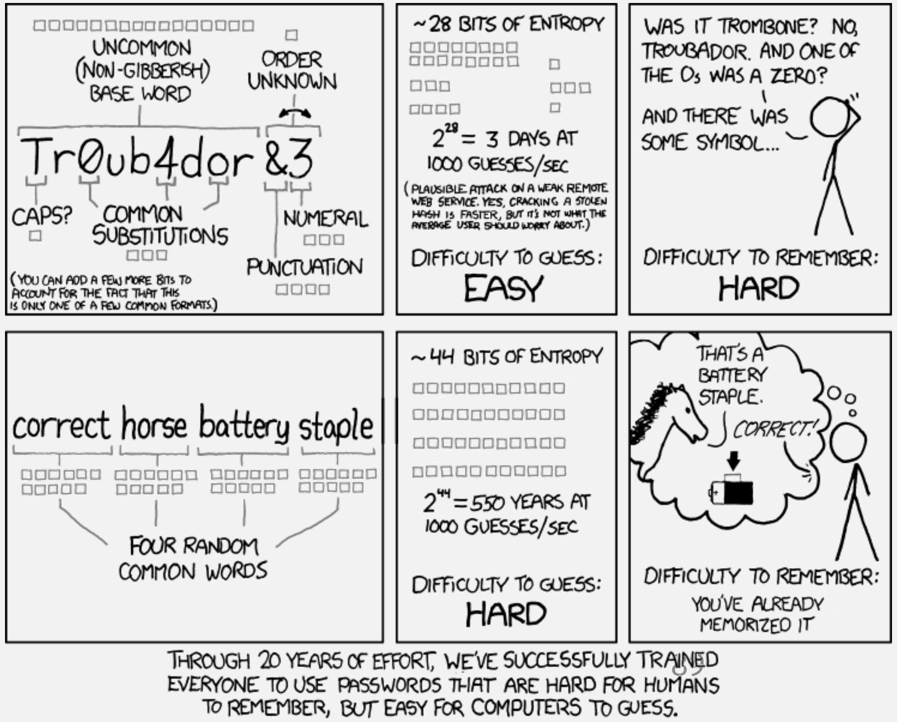
  Every context-free language is a member of P

Search vs Verification

- **<u>Search</u>** problems are often **unsolvable**
- But, **<u>verification</u>** of a search result is usually **solvable**

Examples

- Factoring
  - **Unsolvable**: <u>Find</u> factors of 8633
  - **Solvable**: <u>Verify</u> 89 and 97 are factors of 8633
- Passwords
  - **Unsolvable**: <u>Find</u> my `umb.edu` password
  - **Solvable** : <u>Verify</u> whether my `umb.edu` password is …
    - “`correct horse battery staple`”



The *PATH* Problem

$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

- It's a **search** problem:
  - Exponential time (brute force) algorithm (n^n):
    - Check all possible paths and see if any connects s and t
  - Polynomial time algorithm:
    - Do a breadth-first search (roughly), marking "seen" nodes as we go

PROOF A polynomial time algorithm M for *PATH* operates as follows.

M = "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

Verifying a *PATH*

$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

The **verification** problem:

- Given some path p in G, check that it is a path from s to t
- Let $\boldsymbol{m}$ = longest possible path = # edges in G

NOTE: extra argument p

Verifier V = On input <G, s, t, p>, where p is some set of edges:

1. Check some edge in p has "from" node s; mark and set it as "current" edge
   - Max steps = $O(\boldsymbol{m})$
2. Loop: While there remains unmarked edges in p:
   1. Find the "next" edge in p, whose "from" node is the "to" node of "current" edge
   2. If found, then mark that edge and set it as "current", else reject
   - Each loop iteration: $O(\boldsymbol{m})$
   - # loops: $O(\boldsymbol{m})$
   - Total looping time = $O(\boldsymbol{m}^2)$
3. Check "current" edge has "to" node t; if yes accept, else reject

- Total time = $O(\boldsymbol{m}) + O(\boldsymbol{m}^2) = O(\boldsymbol{m}^2)$ = polynomial in m

PATH can be verified in polynomial time

Verifiers, Formally

$PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph that has a directed path from s to $t\}$

Decider …

… with extra argument:
can be any string that helps to find a result in poly time
(is often just a result itself)

A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

certificate, or ***proof***

We measure the time of a verifier only in terms of the length of w, so a ***polynomial time verifier*** runs in polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier.

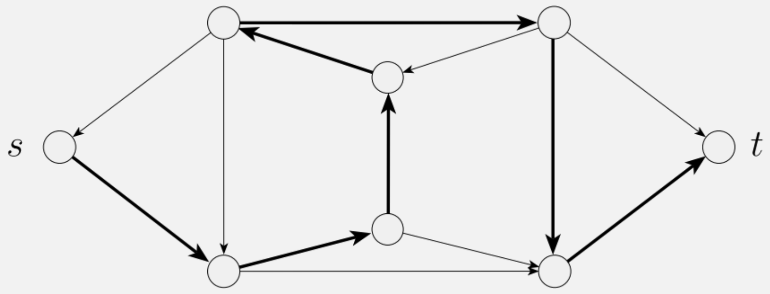
- NOTE: a cert c must be at most length n^k, where n = length of w
  - Why?

So *PATH* is polynomially verifiable

The *HAMPATH* Problem

$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$$

- A **Hamiltonian path** goes through <u>every</u> node in the graph



- The **Search** problem:
  - Exponential time (brute force) algorithm:
    - Check all possible paths and see if any connect s and t using all nodes
  - Polynomial time algorithm:
    - We don't know if there is one!!!
- The **Verification** problem:
  - Still $O(\boldsymbol{m^2})$!
  - *HAMPATH* is polynomially verifiable, but <u>not</u> polynomially decidable

The class **NP**

DEFINITION

NP is the class of languages that have polynomial time verifiers.

- *PATH* is in **NP**, and **P**
- *HAMPATH* is in **NP**, but it's unknown whether it's in **P**

NP = Nondeterministic polynomial time

NP is the class of languages that have polynomial time verifiers.

THEOREM

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

⇒ If a language is in **NP**, then it has a non-deterministic poly time decider

- We know: If a lang L is in **NP**, then it has a poly time verifier V
- Need to: create NTM deciding L:

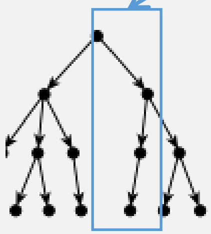
On input w =
- Nondeterministically run V with w and all possible poly length certificates c

⇐ If a language has a non-deterministic poly time decider, then it is in **NP**

- We know: L has NTM decider N,
- Need to: show L is in **NP**, i.e., create polytime verifier V:

On input <w, c> =
- Convert N to deterministic TM, and run it on w, but take only one computation path
- Let certificate c dictate which computation path to follow

Certificate c specifies a path



NP

NTIME(*t*(*n*)) $= \{L \mid L$ is a language decided by an $O(t(n))$ time nondeterministic Turing machine$\}$.

$$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k)$$

NP = Nondeterministic polynomial time

NP vs **P**

Let $t\colon \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the ***time complexity class***, **TIME(*t*(*n*))**, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

P = <u>Deterministic</u> polynomial time

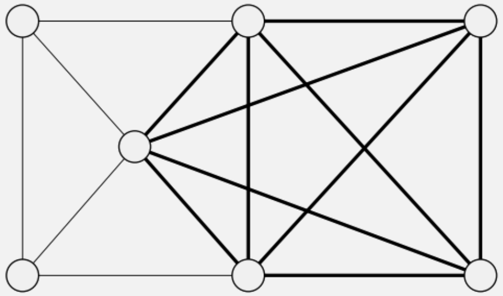
$$\textbf{NTIME}(\boldsymbol{t(n)}) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

$$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k)$$

NP = <u>Nondeterministic</u> polynomial time

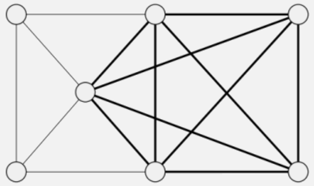
More **NP** Problems

- $CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$
  - A clique is a subgraph where every two nodes are connected
  - A k-clique contains k nodes



- $SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}$

<u>Theorem</u>: *CLIQUE* is in NP



$CLIQUE = \{\langle G, k\rangle |\ G \text{ is an undirected graph with a } k\text{-clique}\}$

PROOF IDEA The clique is the certificate.

Let n = # nodes in G

c is at most n

PROOF The following is a verifier V for *CLIQUE*.

V = "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a subgraph with k nodes in G.
2. Test whether G contains all edges connecting nodes in c.
3. If both pass, *accept*; otherwise, *reject*."

For each node in c, check whether it's in G: $O(n^2)$

For each pair of nodes in c, check whether there's an edge in G: $O(n^2)$

A ***verifier*** for a language A is an algorithm V, where

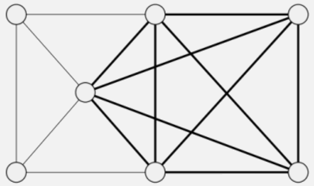
$$A = \{w |\ V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

We measure the time of a verifier only in terms of the length of w, so a ***polynomial time verifier*** runs in polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier.

How to prove a language is in **NP**:
Proof technique #1: create a verifier

NP is the class of languages that have polynomial time verifiers.

Proof 2: *CLIQUE* is in NP



$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

$N =$ "On input $\langle G, k\rangle$, where G is a graph:
1. Nondeterministically select a subset c of k nodes of G.
2. Test whether G contains all edges connecting nodes in c.
3. If yes, *accept*; otherwise, *reject*."

"try all subgraphs"

$O(n^2)$

To prove a lang L is in **NP**, create <u>either</u> a:
1. **Deterministic** poly time **verifier**
2. **Nondeterministic** poly time **decider**

How to prove a language is in **NP**:
Proof technique #2: create an NTM

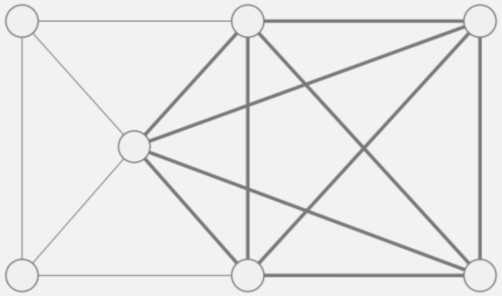
Don't forget to count the steps

THEOREM

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

More **NP** Problems

- $CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$
  - A clique is a subgraph where every two nodes are connected
  - A *k*-clique contains *k* nodes



- $SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}$
  - Some subset of a set of numbers *S* must sum to some total *t*
  - e.g., $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in SUBSET\text{-}SUM$

<u>Theorem</u>: *SUBSET-SUM* is in NP

$$SUBSET\text{-}SUM = \{\langle S,t\rangle \mid S = \{x_1,\ldots,x_k\}, \text{ and for some } \{y_1,\ldots,y_l\} \subseteq \{x_1,\ldots,x_k\}, \text{ we have } \Sigma y_i = t\}$$

PROOF IDEA The subset is the certificate.

To prove a lang is in **NP**, create <u>either</u>:
1. **Deterministic** poly time **verifier**
2. **Nondeterministic** poly time **decider**

PROOF The following is a verifier V for *SUBSET-SUM*.

V = "On input $\langle\langle S,t\rangle, c\rangle$:
1. Test whether c is a collection of numbers that sum to t.
2. Test whether S contains all the numbers in c.
3. If both pass, *accept*; otherwise, *reject*."

Runtime?

Proof 2: *SUBSET-SUM* is in NP

$$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}$$

To prove a lang is in **NP**, create either:
1. **Deterministic** poly time **verifier**
2. **Nondeterministic** poly time **decider**

ALTERNATIVE PROOF We can also prove this theorem by giving a nondeterministic polynomial time Turing machine for *SUBSET-SUM* as follows.

$N =$ "On input $\langle S, t\rangle$:
1. Nondeterministically select a subset c of the numbers in S.
2. Test whether c is a collection of numbers that sum to t.
3. If the test passes, *accept*; otherwise, *reject*."

Runtime?

$$COMPOSITES = \{x \mid x = pq, \text{ for integers } p, q > 1\}$$

- A composite number is <u>not</u> prime

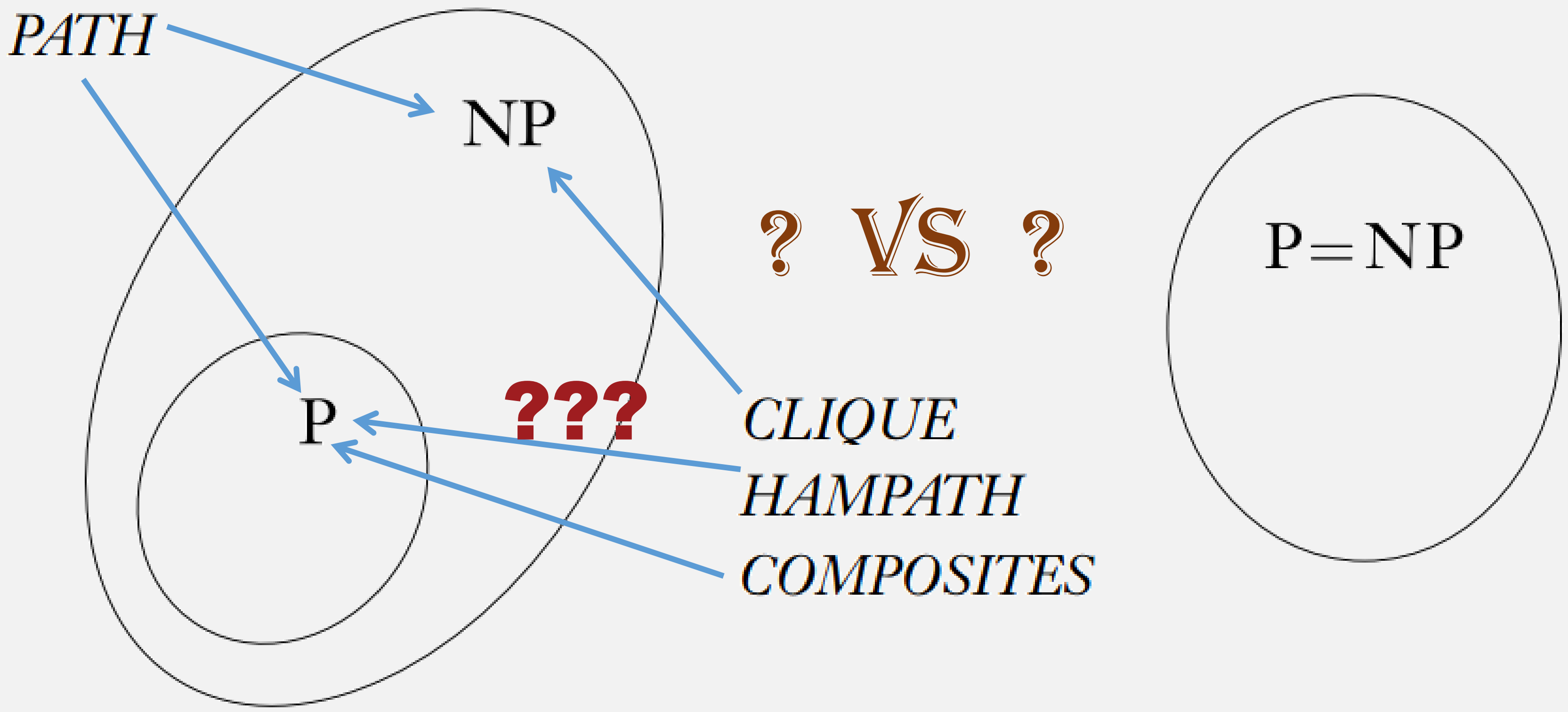
- *COMPOSITES* is polynomially verifiable
  - i.e., it's in **NP**
  - i.e., factorability is in **NP**

- A certificate could be:
  - Some factor that is not 1

- Checking existence of factors (or not, i.e., testing primality) …
  - … is also poly time
  - But only discovered <u>recently</u> (2002)!

One of the Greatest unsolved

~~HW~~ Question: Does P = NP?



How do you prove an algorithm doesn't have a poly time algorithm?
(in general it's hard to prove that something doesn't exist)

Implications if **P = NP**

- Every problem with a "brute force" solution also has an efficient solution
- I.e., "unsolvable" problems are "solvable"
- <u>BAD</u>:
  - Cryptography needs unsolvable problems
  - Near perfect AI learning, recognition
- <u>GOOD</u>: Optimization problems are solved
  - Optimal resource allocation could fix all the world's (food, energy, space …) problems?

Who doesn't like niche NP jokes?

Progress on whether **P = NP** ?

- Some, but still not close

$P \stackrel{?}{=} NP$

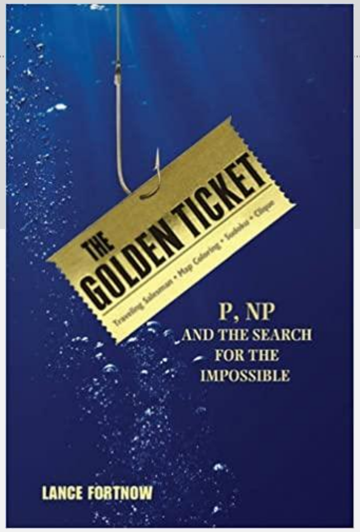
Scott Aaronson*

The Status of the P Versus NP Problem

By Lance Fortnow

Communications of the ACM, September 2009, Vol. 52 No. 9, Pages 78-86

10.1145/1562164.1562186



- One important concept discovered:
  - **NP**-Completeness

NP-Completeness

Must look at all langs, can't just look at a single lang

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and — easy
2. every A in NP is polynomial time reducible to B. — hard????

What's this?

- How does this help the **P** = **NP** problem?

THEOREM

If B is NP-complete and $B \in$ P, then P $=$ NP.

Flashback: Mapping Reducibility

Language A is ***mapping reducible*** to language B, written $A \leq_{\mathrm{m}} B$, if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \iff f(w) \in B.$$

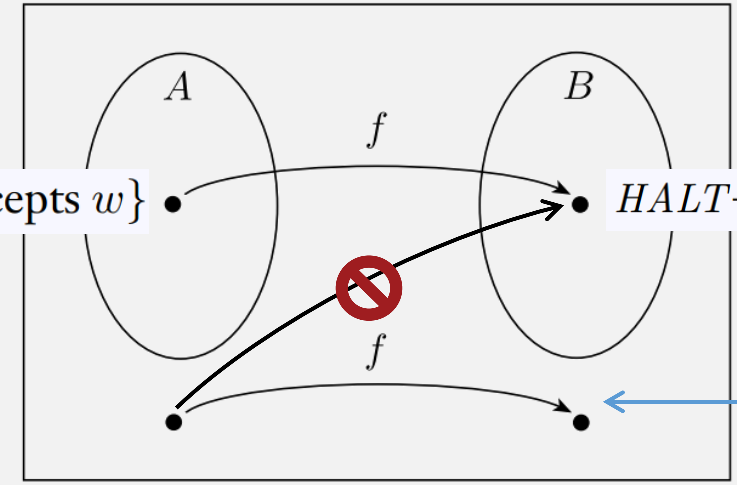
The function f is called the ***reduction*** from A to B.

<u>IMPORTANT</u>: "if and only if" …

To show <u>mapping reducibility</u>:

1. create **computable fn**
2. and then show **forward direction**
3. and **reverse direction** (or **contrapositive of forward direction**)

$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$



$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$

… means $\overline{A} \leq_{\mathrm{m}} \overline{B}$

A function $f\colon \Sigma^* \longrightarrow \Sigma^*$ is a ***computable function*** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Polynomial Time Mapping Reducibility

Language A is ***mapping reducible*** to language B, written $A \leq_m B$, if there is a computable function $f \colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***reduction*** from A to B.

Language A is ***polynomial time mapping reducible***, or simply ***polynomial time reducible***, to language B, written $A \leq_P B$, if a polynomial time computable function $f \colon \Sigma^* \longrightarrow \Sigma^*$ exists, where for every w,

$$w \in A \iff f(w) \in B.$$

Don't forget: "if and only if" …

The function f is called the ***polynomial time reduction*** of A to B.

A function $f \colon \Sigma^* \longrightarrow \Sigma^*$ is a **poly time** ***computable function*** if some **poly time** Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Flashback: If $A \leq_m B$ and B is decidable, then A is decidable.

Has a decider

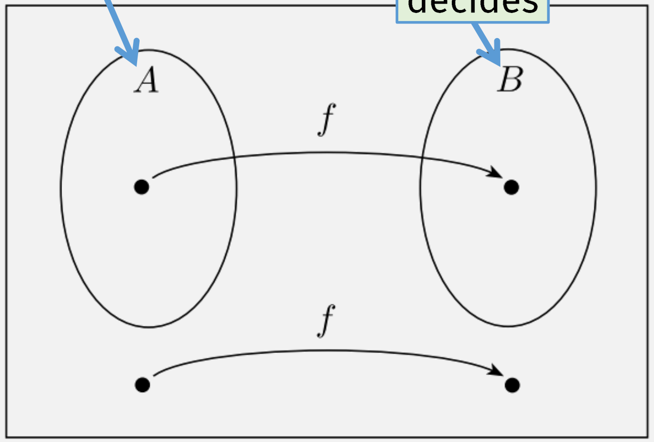
PROOF We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows.

N = "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

decides

decides



This proof only works because of the if-and-only-if requirement

Language A is ***mapping reducible*** to language B, written $A \leq_m B$, if there is a computable function $f: \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \iff f(w) \in B.$$

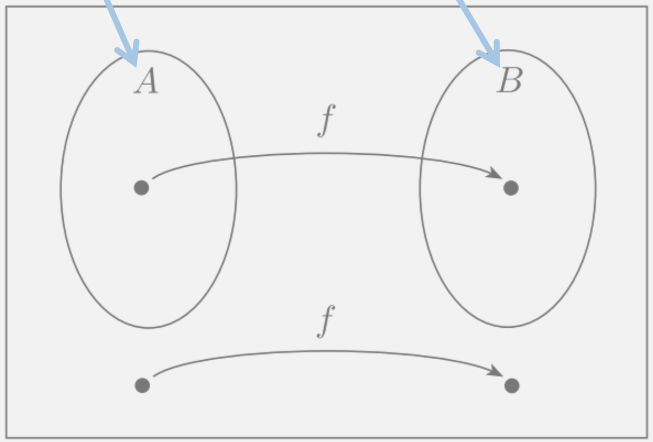
The function f is called the ***reduction*** from A to B.

Thm: If $A \le_{\mathrm{m}\ \mathbf{P}} B$ and B ~~is decidable,~~ $\in \mathrm{P}$ then A ~~is decidable.~~ $\in \mathrm{P}$

PROOF We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows.

N = "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."



Language A is ***mapping reducible*** to language B, written $A \le_{\mathrm{m}} B$, if there is a computable function $f : \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \iff f(w) \in B.$$

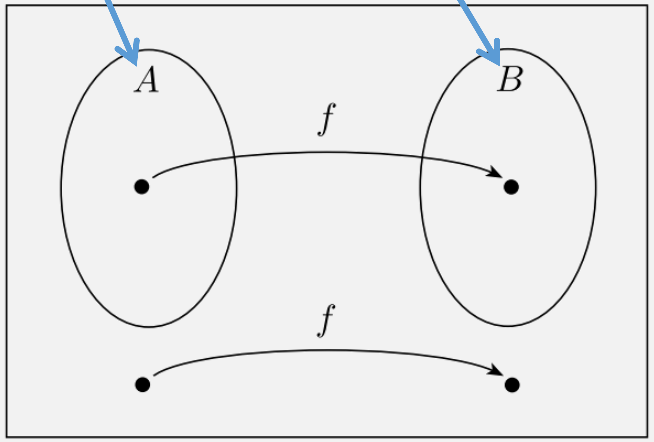
The function f is called the ***reduction*** from A to B.

Thm: If $A \leq_{\text{P}} B$ (~~m~~) and B ~~is decidable,~~ $\in$ P, then A ~~is decidable.~~ $\in$ P

PROOF We let M be the poly time decider for B and f be the poly time reduction from A to B. We describe a poly time decider N for A as follows.

N = "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."



Language A is poly time ***mapping reducible*** to language B, written $A \leq_{\text{m}} B$, if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***reduction*** from A to B.

Next Time: *3SAT* is polynomial time reducible to *CLIQUE*.

Check-in Quiz 4/25

On gradescope