Introduction to the Theory of Computation

Set 11 — Complexity (2)

Time Complexity

Definition

Let M be any deterministic Turing machine that halts on all inputs.

The running time or time complexity of M is the function $f:\mathbb{N}\rightarrow\mathbb{N}$, where f(n) is the maximum number of steps that M uses on any input of length n.

Time Complexity Class

The running time or time complexity of M is the function $f:\mathbb{N}\rightarrow\mathbb{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n.

Definition

Let $t:\mathbb{N}\rightarrow\mathbb{N}$ be a function.

The time complexity class, $TIME(t(n))$, is

$TIME(t(n))$ = {L | L is a language decided an $O(t(n))$-time Turing Machine}

Time Complexity for TMs

Relationship of time complexity for different TM models

If a problem can be solved in $O(t(n))$ time on a **multi-tape TM**, it can be solved in $O(t^2(n))$ time on a single-tape TM

If a problem can be solved in $O(t(n))$ time on a **nondeterministic TM**, it can be solved in $2^{O(t(n))}$ time on a deterministic TM

Polynomial vs. Exponential Time

We distinguish between algorithms that have polynomial running time and those that have exponential running time

- **Assume a single tape deterministic TM**

Polynomial functions – even ones with large exponents – grow less quickly than exponential functions

We can only process large data sets with polynomial running time algorithms

The Class P

P is the class of languages that are *decidable* in *polynomial time* on a single-tape Turing machine

$P = \bigcup_k TIME(n^k)$

P "roughly corresponds" to the problems that are realistically solvable on a computer

Solving *vs.* Verifying

What if we don't know how to **solve** the problem in $O(n^k)$ time?

Given a problem and a potential solution, can we **verify** the solution is correct?

Example

The bin-packing problem

- Given a set of **n** items with fractional weights $w_1, w_2, \ldots, w_n$, and **k bins** that can hold a maximum weight of 1 each, can we place these items into the bins?

There is no known $O(n^k)$ solution to this problem

What if we have a potential solution

- $b_1, b_2, \ldots, b_n$ $1 \le b_i \le k$ *b_i indicates the bin for item i*

Can we verify it in $O(n^k)$ time?

Verifier

M = “On input $<w_1, \ldots, w_n, b_1, \ldots, b_n, k>$

1. Initialize $s_1, s_2, \ldots, s_k$ to 0
2. For $i = 1, \ldots, n$
3. if $b_i \notin \{1, 2, \ldots, k\}$ Reject
4. $s_{b_i} = s_{b_i} + w_i$
5. if $s_{b_i} > 1$ Reject
6. Next i
7. Accept

The Class NP

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

A={w | V accepts <w,c> for some string c}

The string c is called a certificate of membership in A.

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

Why NP?

NP problems have polynomial-time solutions on nondeterministic TMs.

The N in NP stands for

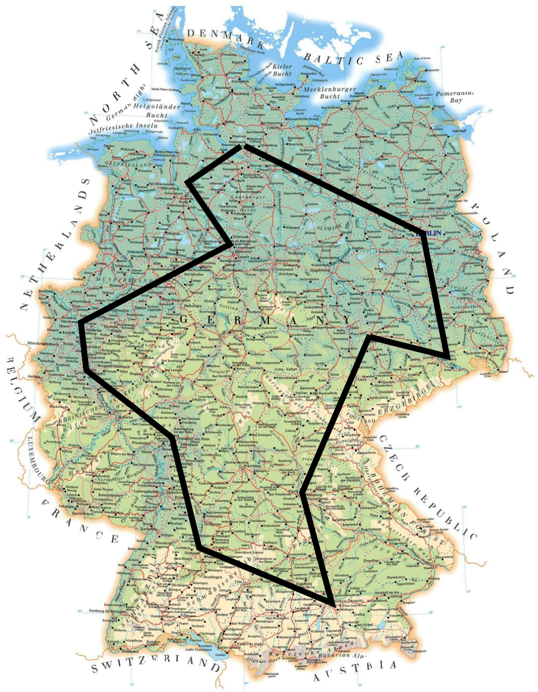
non-deterministic

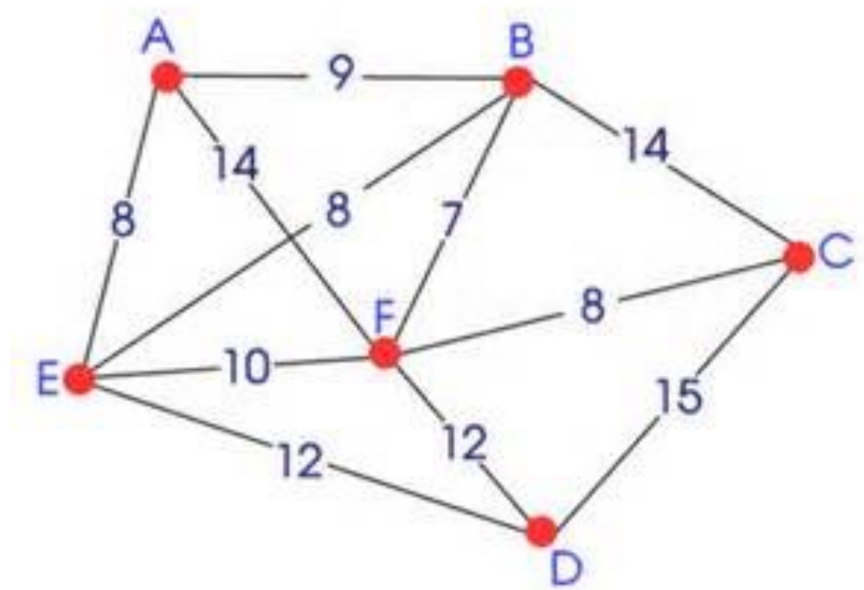
Any language in NP can be non-deterministically solved in polynomial time using the verifier

- Guess the certificate
- Verify

Example

Find a verifier for the traveling-salesperson problem





Berlin
Bremen
Dresden
Düsseldorf
Frankfurt
Hamburg
Hannover
Köhn
Leipzig
Munchan
Nümberg
Stuttgart

$d \stackrel{?}{=} 2735\text{km}$

Hamburg→Bremen→Hannover→Düsseldorf →Köhn→Frankfurt→Stuttgart→Munchan→ Nümberg→Dresden→Berlin→Hamburg

A
B
C
D
E
F

$d \stackrel{?}{=} 54$

$d \stackrel{?}{=} 59$

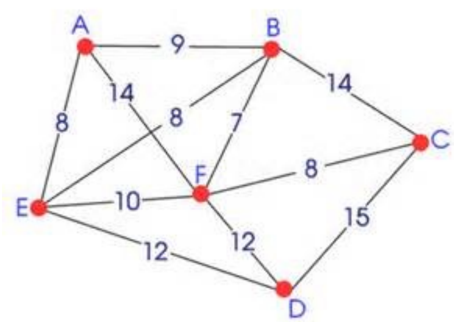
Example

Find a verifier for the traveling-salesperson problem

- **Given a weighted graph G (where each Edge has an associated weight) and a distance d, does there exist a cycle through the graph that visits each Vertex exactly once (except for the start/end vertex) and has a total distance d?**

Berlin
Bremen
Dresden
Düsseldorf
Frankfurt
Hamburg
Hannover
Köhn
Leipzig
Munchan
Nümberg
Stuttgart

$d \stackrel{?}{=} 2735km$



A
B
C
D
E
F

$d \stackrel{?}{=} 54$

$d \stackrel{?}{=} 59$

Example *(continued)*

Find a verifier for the traveling-salesperson problem

Verifier takes input $<v_{i_1}, v_{i_2}, \ldots, v_{i_n}>$ $v_{i_k} \in V$

Check that the input is a permutation of the nodes of the graph

- $O(|V|^2)$

Check that the sum of the edges between adjacent v_{i_k} is equal to d

- $O(|E| \times |V|)$

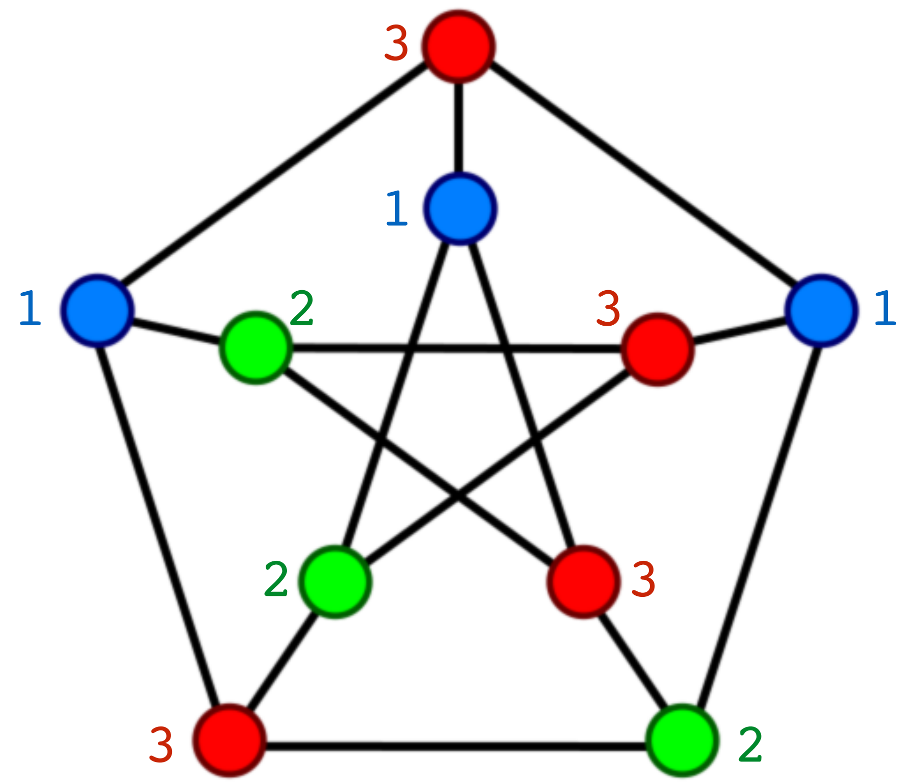
Example: Colorability

Color the vertices of a graph such that no two adjacent vertices share the same color.

The 3-Color Problem

- **INPUT: Graph G with vertices V and edges E**
- **PROPERTY: There is a function $f\colon \mathbb{N} \to \{1, 2, 3\}$ such that if u and v are adjacent then $f(u) \neq f(v)$**

Determine if the graph can be colored using **at most 3** colors such that no two adjacent vertices are given the same color.



Verifiable in polynomial time

The Class co-NP

$$L \in \textbf{co-NP} \iff \overline{L} \in \textbf{NP}$$

A language is in co-NP if and only if its complement is in NP.

For example,

$\overline{\text{3-Color}} = \{\langle G \rangle \mid G$ is a graph that **cannot** be colored using only 3 colors$\}$

NP: A language L is in NP if and only if a *qualifying* certificate can be checked efficiently.

co-NP: A language L is in co-NP if and only if a *disqualifying* certificate can be checked efficiently.

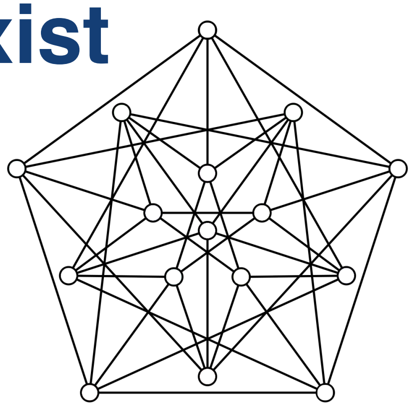
Is NP closed under complementation?

The 3-Color problem is in NP.

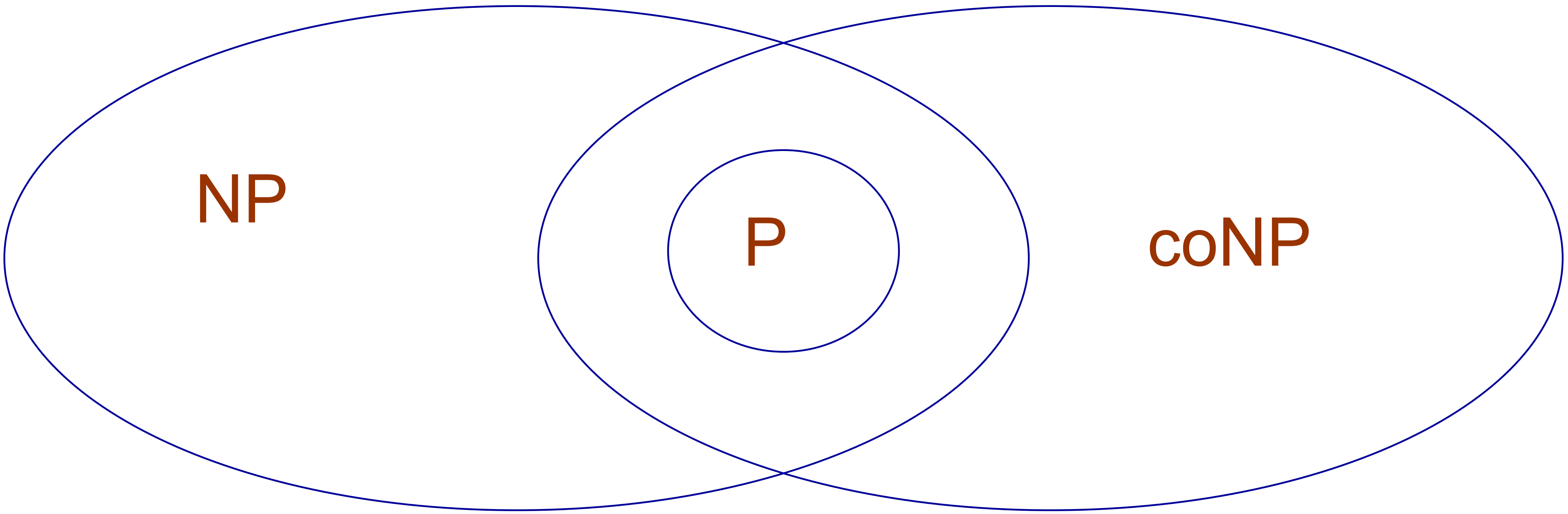
What about $\overline{\text{3-Color}}$?

Can we *verify* in polynomial time that a graph cannot be 3-colored?

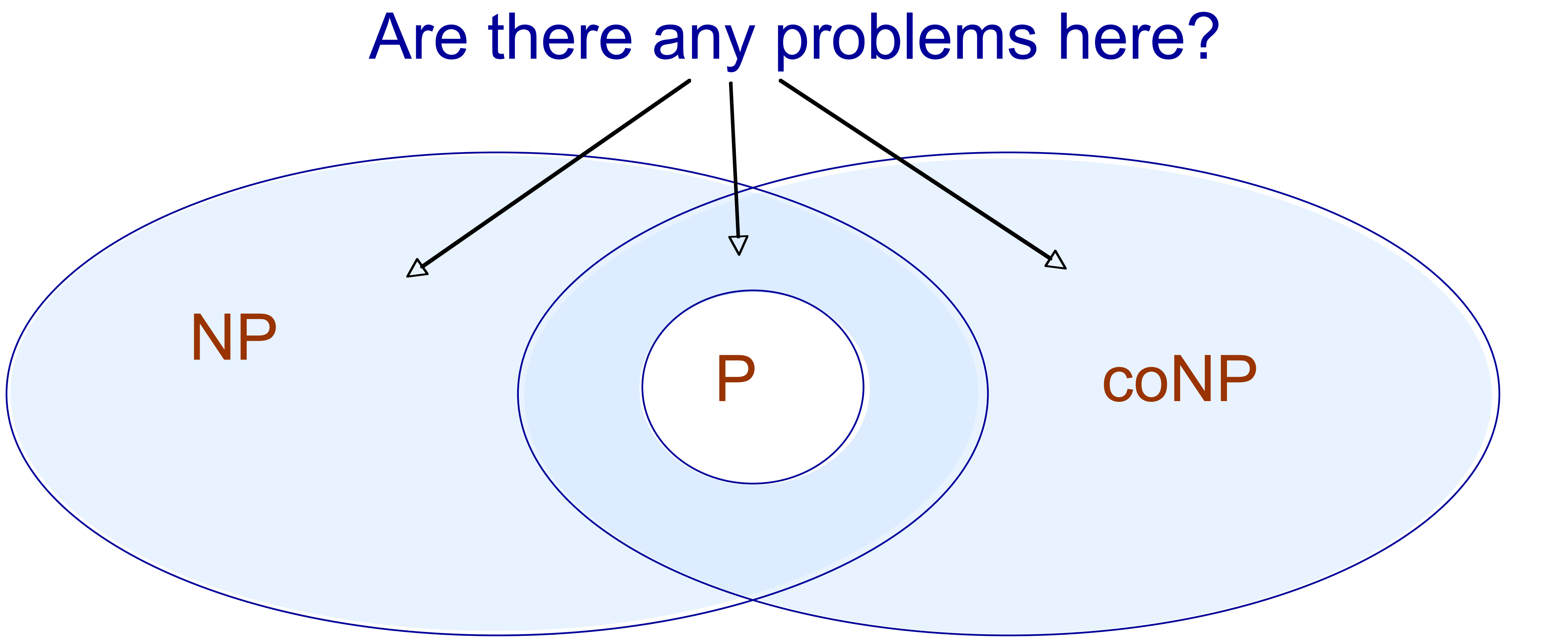
- Not obviously
- It seems we need to check many 3-colorings before we can conclude that none exist



What we know



What we don’t know



Who wants $1,000,000?

In May, 2000, the Clay Mathematics Institute named seven open problems in mathematics the Millennium Problems

- Anyone who solves any of these problems will receive $1,000,000
- Proving whether or not P equals NP is one of these problems

http://www.claymath.org/millennium-problems/p-vs-np-problem

Solving NP Problems

The best-known methods for solving problems in NP that are not known to be in P take exponential time

- Brute force search

NP-completeness

A problem C is NP-complete if finding a polynomial-time solution for C would imply P=NP

Definition: Language B is NP-complete if it satisfies two conditions:

- **B is in NP, and**
- **Every A in NP is polynomial time reducible to B**

Reductions and NP-completeness

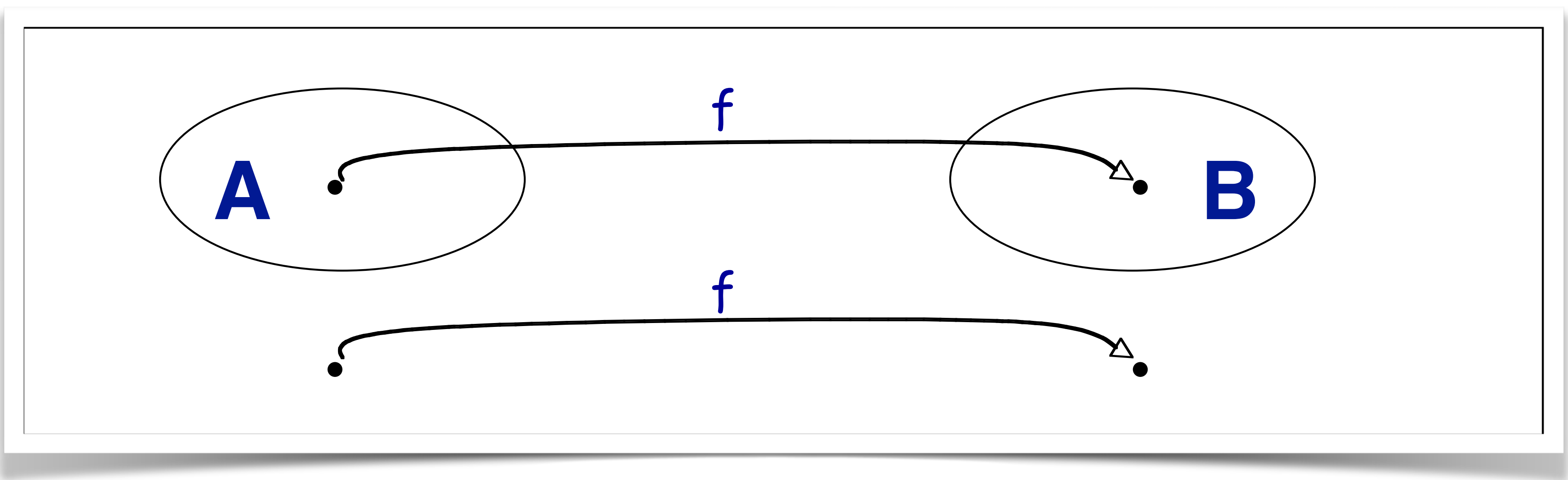
If we can prove an NP-complete problem C can be polynomially reduced to a problem A, then we've shown A is NP-complete

- A polynomial-time solution to A would provide a polynomial-time solution to C, which would imply P=NP

Polynomial Reductions

Definition: Language A is polynomial-time reducible to language B, written $A \leq_P B$, if a polynomial time computable function $f:\Sigma^* \to \Sigma^*$ exists, where for every w

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



Reductions & NP-completeness

Theorem: If $A \leq_P B$ and $B \in P$, then $A \in P$

Proof: Let M be the polynomial time algorithm that decides B and let f be the polynomial reduction from A to B. Consider the TM N

N = "On input w

- Compute f(w)
- Run M on f(w) and output M's result"

Then N decides A in polynomial time.

Implications of NP-completeness

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

Theorem: If B is NP-complete and B $\leq_P$ C for some C in NP, then C is NP-complete

Showing a Problem is NP-complete

Two steps to proving a problem L is NP-complete

- Show the problem is in NP
  - Demonstrate there is a polynomial time verifier for the problem
- Show some NP-complete problem can be polynomially reduced to L

https://en.wikipedia.org/wiki/List_of_NP-complete_problems

Summary

To show a language L is NP-complete

- Demonstrate L is in NP
- Find a language C that is known to be NP-complete
- Create a function f from C to L
- Demonstrate that if x is in C then $f(x)$ is in L
- Demonstrate that if $f(x)$ is in L then x is in C
- Demonstrate f is computable in polynomial time

Course Recap — Goals

Explore the capabilities and limitations of computers

- Automata theory
  - How can we mathematically model computation?
- Computability theory
  - What problems can be solved by a computer?
- Complexity theory
  - What makes some problems computationally hard and others easy?

Course Recap

Automata Theory ✔

- Introduced DFA, NFA, Regular Grammar, RE
  - Showed that they all accept the same class of languages
- Introduced CFG, PDA
  - PDA is essentially an NFA with a stack
  - PDAs and CFGs accept the same class of languages

Course Recap

Computability Theory ✔

- Introduced TM
  - Like PDA's with more general memory model
- Importance of TM
  - Church-Turing Thesis
  - Any algorithm can be implemented on a TM
- Use the TM model and Church-Turing Thesis to understand and classify languages
  - Decidable languages
  - Undecidable languages
  - Recognizable languages
  - Unrecognizable languages
  - Complements of languages in these classes

Course Recap

Complexity Theory ✔

- Use TM model to determine how long an algorithm takes to run
  - Function of input length
- Classify algorithms according to their complexity
- Deciders vs Verifiers
- P , NP , NP-completeness