

EL9343

Data Structure and Algorithm

Lecture 13: NP-Completeness

Instructor: Pei Liu

Last Lecture

- ▶ Single-Source Shortest Paths
 - ▶ Nonnegative edge weights: Dijkstra's algorithm
 - ▶ Unweighted graphs: BFS
 - ▶ General case: Bellman-Ford algorithm
 - ▶ DAG: Topological sort + one pass Bellman-Ford
- ▶ All Pairs Shortest Paths
 - ▶ Nonnegative edge weights: $|V|$ times of Dijkstra's algorithm
 - ▶ Unweighted graphs: $|V|$ times of BFS
 - ▶ General case: Floyd-Warshall algorithm

NP Complete Problems

- ▶ The course so far: techniques for designing efficient algorithms, e.g., divide-and-conquer, dynamic programming, greedy algorithms.
- ▶ What happens if you can't find an efficient algorithm?
 - ▶ Is it your “fault” or the problem's?
- ▶ Showing that a problem has an efficient algorithm is, relatively, easy. “All” that is needed is to demonstrate an algorithm.
- ▶ Proving that no efficient algorithm exists for a particular problem is difficult. How can we prove the nonexistence of something?
- ▶ We will now learn about **NP Complete Problems**, which provide us with a way to approach this question.

NP-Complete Problems

This is a very large class of thousands of practical problems for which

- ▶ It is not known if the problems have “efficient” solutions
- ▶ It is known that if any one of the NP-Complete Problems has an efficient solution then all of the NP-Complete Problems have efficient solutions
- ▶ Researchers have spent innumerable man-years trying to find efficient solutions to these problems and failing
- ▶ There is a large body of tools that often permit us to prove when a new problem is NP-complete.
- ▶ The problem of finding an efficient solution to an NP-Complete problem is known, in shorthand as $P = NP$? There is currently a US \$1,000,000 award offered by the Clay Institute (<http://www.claymath.org/>) for its solution

Optimization & Decision Problems

▶ **Decision problems**

- ▶ Given an input and a question regarding a problem, determine if the answer is yes or no

▶ **Optimization problems**

- ▶ Find a solution with the “best” value
- ▶ Optimization problems can be cast as decision problems that are easier to study
 - ▶ *E.g.:* Shortest path: G = unweighted directed graph
 - ▶ Find a path between u and v that uses the fewest edges
 - ▶ *Does a path exist from u to v consisting of at most k edges?*

Class of “P” Problems

- ▶ **Class P** consists of (decision) problems that are solvable in polynomial time
- ▶ Polynomial-time algorithms
 - ▶ Worst-case running time is $O(n^k)$, for some constant k
- ▶ Examples of polynomial time:
 - ▶ $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
- ▶ Examples of non-polynomial time:
 - ▶ $O(2^n)$, $O(n^n)$, $O(n!)$
- ▶ Are non-polynomial algorithms always worse than polynomial algorithms?
 - ▶ - $n^{1,000,000}$ is *technically* tractable, but really impossible -
 $n^{\log \log \log n}$ is *technically* intractable, but easy

Tractable/Intractable Problems

- ▶ Problems in P are also called **tractable**
- ▶ Problems **not** in P are **intractable or undecidable**
 - ▶ Can be solved in reasonable time only for small inputs, as they grow large, we are unable to solve them in reasonable time
 - ▶ Or, can not be solved at all (e.g, Halting Problem)

Example of Unsolvable Problem

- ▶ Turing discovered in the 1930's that there are problems **unsolvable/undecidable** by *any* algorithm.
- ▶ The most famous of them is the ***halting problem***
 - ▶ Given an arbitrary algorithm and its input, will that algorithm eventually halt, or will it continue forever in an “*infinite loop*?”
 - ▶ Decision problem: answer is yes or no
 - ▶ Uncomputable: no algorithm solves it (correctly in finite time on all inputs)

We don't care about such problems here; take a theory class

Intractable Problems

- ▶ Can be classified in various categories based on their degree of difficulty, e.g.,
 - ▶ NP
 - ▶ NP-complete
 - ▶ NP-hard
- ▶ Let's define NP algorithms and NP problems ...

Nondeterministic and NP Algorithms

- ▶ **Nondeterministic algorithm** = two stage procedure:
- ▶ Nondeterministic (“guessing”) stage:
 - ▶ generate randomly an arbitrary string that can be thought of as a candidate solution (“certificate”)
- ▶ Deterministic (“verification”) stage:
 - ▶ take the certificate and the instance to the problem and returns YES if the certificate represents a solution
- ▶ **NP algorithms (Nondeterministic polynomial)**
 - ▶ verification stage is polynomial

Class of “NP” Problems

- ▶ **Class NP** consists of problems that could be solved by NP algorithms
 - ▶ i.e., verifiable in polynomial time
- ▶ If we were given a “certificate” of a solution, we could verify that the certificate is correct in time polynomial to the size of the input
- ▶ Warning: NP does **not** mean “non-polynomial”

Example: Hamiltonian Cycle

- ▶ **Given:** a directed graph $G = (V, E)$, determine a simple cycle that contains each vertex in V

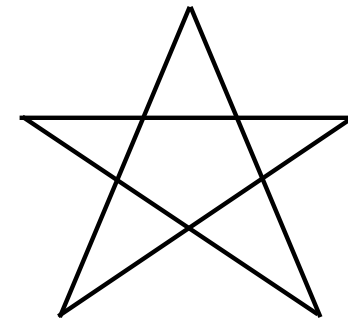
- ▶ Each vertex can only be visited once

- ▶ **Certificate:**

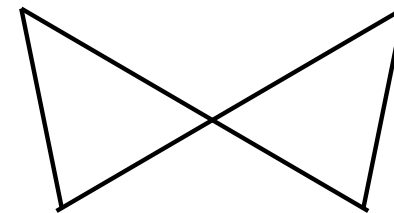
- ▶ Sequence: $\langle v_1, v_2, v_3, \dots, v_{|V|} \rangle$

- ▶ Cannot solve in polynomial time

- ▶ Can verify solution in polynomial time



hamiltonian

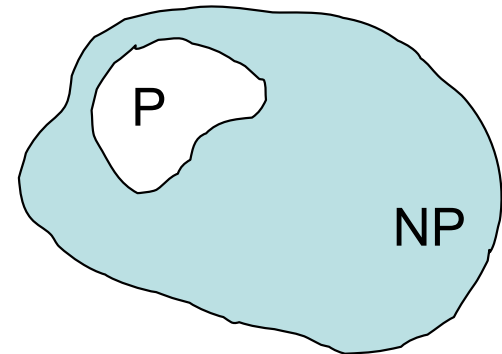


not
hamiltonian

Is $P = NP$?

- ▶ Any problem in P is also in NP :

$$P \subseteq NP$$



- ▶ The big (and **open question**) is whether $NP \subseteq P$ or $P = NP$
 - ▶ i.e., if it is always easy to check a solution, should it also be easy to find a solution?
- ▶ Most computer scientists believe that this is false but we do not have a proof ...

NP-Completeness (informally)

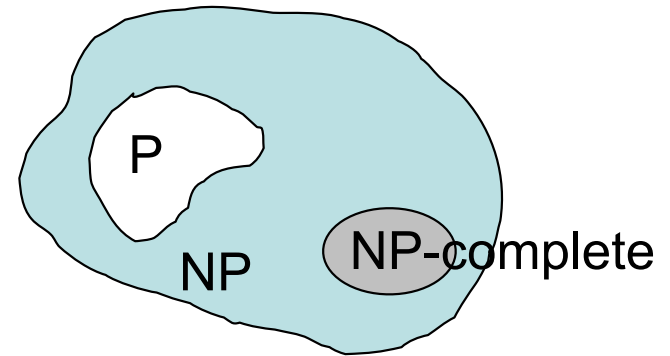
- ▶ **NP-complete** problems are

defined as the hardest

problems in NP: an interesting class of problems

whose status is unknown

- ▶ Most practical problems turn out to be either P or NP-complete.

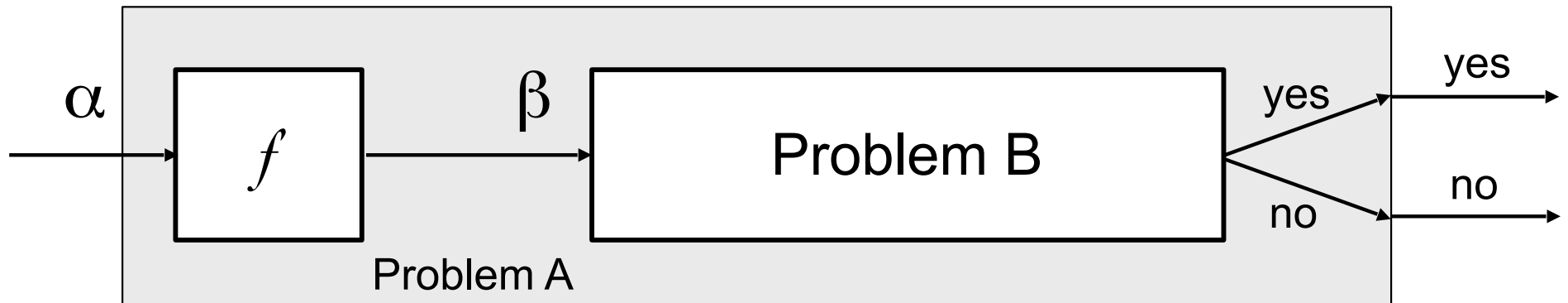


Reductions

- ▶ Reduction is a way of saying that one problem is “**easier**” than another.
- ▶ We say that problem A is easier than problem B, (i.e., we write “ **$A \leq B$** ”)
if we can solve A using the algorithm that solves B.

Intuitively: If A reduces in polynomial time to B, A is “no harder to solve” than B

Idea: transform the inputs of A to inputs of B



Polynomial Reductions

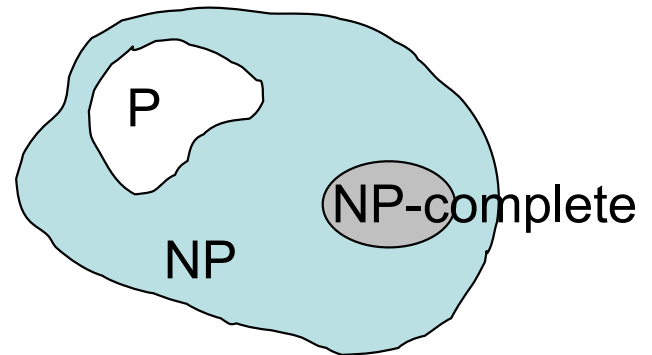
- ▶ Given two problems A , B , we say that A is polynomially **reducible** to B ($A \leq_p B$) if:
 - ▶ There exists a function f that converts the input of A to inputs of B in polynomial time
 - ▶ $A(i) = \text{YES} \Leftrightarrow B(f(i)) = \text{YES}$

NP-Completeness (formally)

- ▶ A problem B is **NP-complete** if:

(1) $B \in \text{NP}$

(2) $A \leq_p B$ for all $A \in \text{NP}$

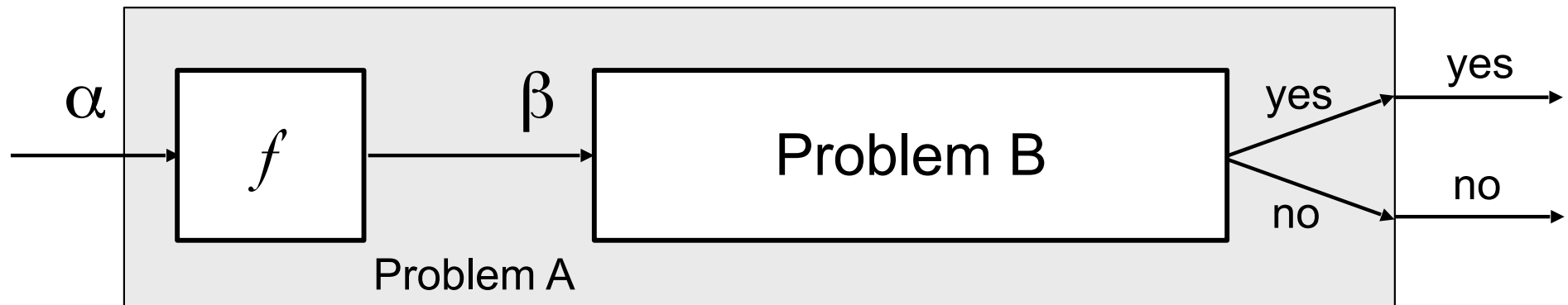


- ▶ If B satisfies only property (2) we say that B is **NP-hard**
- ▶ No polynomial time algorithm has been discovered for an **NP-Complete** problem
- ▶ No one has ever proven that no polynomial time algorithm can exist for any **NP-Complete** problem

NP-naming convention

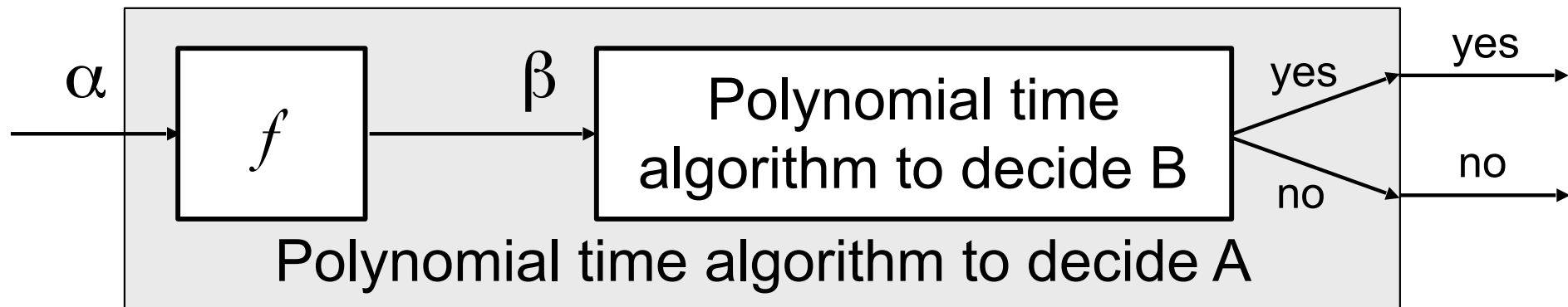
- ▶ **NP-complete** - means problems that are 'complete' in NP, i.e. the most difficult to solve in NP
- ▶ **NP-hard** - stands for 'at least' as hard as NP (but not necessarily **in** NP);
- ▶ **NP-easy** - stands for 'at most' as hard as NP (but not necessarily **in** NP);
- ▶ **NP-equivalent** - means equally difficult as NP, (but not necessarily **in** NP);

Implications of Reduction



- ▶ If $A \leq_p B$ and $B \in P$, then $A \in P$
- ▶ If $A \leq_p B$ and $A \notin P$, then $B \notin P$
- ▶ If $A \leq_p B$ and A is NP-Complete, B is NP-Hard. In addition, if $B \in NP \Rightarrow B$ is NP-Complete

Proving Polynomial Time



1. Use a **polynomial time** reduction algorithm to transform A into B
2. Run a known **polynomial time** algorithm for B
3. Use the answer for B as the answer for A

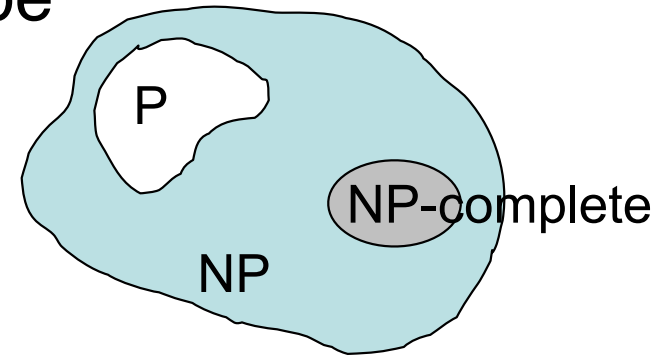
Proving NP-Completeness In Practice

- ▶ Prove that the problem B is in NP
 - ▶ A randomly generated string can be checked in polynomial time to determine if it represents a solution
- ▶ Pick a known NP-Complete problem A
- ▶ Reduce A to B: show that **one known** NP-Complete problem A can be transformed to B in polynomial time
 - ▶ No need to check that **all** NP-Complete problems are reducible to B

Revisit “Is $P = NP$?”

Theorem: If any NP-Complete problem can be solved in polynomial time \Rightarrow then $P = NP$.

- ▶ If any *one* NP-Complete problem can be solved in polynomial time...
- ▶ ...then *every* NP-Complete problem can be solved in polynomial time...
- ▶ ...and in fact *every* problem in **NP** can be solved in polynomial time (which would show **$P = NP$**)
- ▶ Thus: solve hamiltonian-cycle in $O(n^{100})$ time, you’ve proved that **$P = NP$** . Retire rich & famous.



Reductions Examples

Convert your problem into a problem you already know how to solve (instead of solving from scratch)

- ▶ unweighted shortest path → weighted (set weights = 1)
- ▶ min-product path → shortest path (take logs)
- ▶ longest path in DAG → shortest path in DAG (negate weights)
 - ▶ longest path in general graph is NP-Complete

Satisfiability Problem (SAT)

Definition: A **Boolean formula** is a logical formula

which consists of

boolean variables (0=false, 1=true),

logical operations

\bar{x} , **NOT**,

$x \vee y$, **OR**,

$x \wedge y$, **AND**.

These are defined by:

x	y	\bar{x}	$x \vee y$	$x \wedge y$
0	0	1	0	0
0	1		1	0
1	0	0	1	0
1	1		1	1

- ▶ **SAT problem:** Determine whether an input Boolean formula is satisfiable. If a Boolean formula is satisfiable, it is a yes-input; otherwise, it is a no-input.
- ▶ SAT was the first problem shown to be NP-complete!

Satisfiability Problem (SAT)

A given Boolean formula is *satisfiable* if there is a way to assign truth values (0 or 1) to the variables such that the final result is 1.

Example: $f(x, y, z) = (x \wedge (y \vee \bar{z})) \vee (\bar{y} \wedge z \wedge \bar{x})$.

x	y	z	$(x \wedge (y \vee \bar{z}))$	$(\bar{y} \wedge z \wedge \bar{x})$	$f(x, y, z)$
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	1	0	1

For example, the assignment $x = 1, y = 1, z = 0$ makes $f(x, y, z)$ true, and hence it is satisfiable.

Satisfiability Problem (SAT)

Example:

$$f(x, y) = (x \vee y) \wedge (\bar{x} \vee y) \wedge (x \vee \bar{y}) \wedge (\bar{x} \vee \bar{y}).$$

x	y	$x \vee y$	$\bar{x} \vee y$	$x \vee \bar{y}$	$\bar{x} \vee \bar{y}$	$f(x, y)$
0	0	0	1	1	1	0
0	1	1	1	0	1	0
1	0	1	0	1	1	0
1	1	1	1	1	0	0

There is no assignment that makes $f(x, y)$ true, and hence it is NOT satisfiable.

K-CNF-SAT Problem

For a fixed k , consider Boolean formulas in k -conjunctive normal form (k -CNF):

$$f_1 \wedge f_2 \wedge \cdots \wedge f_n$$

where each f_i is of the form

$$f_i = y_{i,1} \vee y_{i,2} \vee \cdots \vee y_{i,k}$$

where each $y_{i,j}$ is a variable or the negation of a variable.

An example of a 3-CNF formula is

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee x_4).$$

- ▶ **3-SAT: NP Complete**
- ▶ **2-SAT: P**

k -SAT problem: Determine whether an input Boolean k -CNF formula is satisfiable.

Other NP-Complete Problems

- ▶ Knapsack
- ▶ 3-Partition: given n integers, can you divide them into triples of equal sum?
- ▶ Traveling Salesman Problem: shortest path that visits all vertices of a given graph — decision version: is minimum weight $\leq x$?
- ▶ Longest common subsequence of k strings
- ▶ Shortest paths amidst obstacles in 3D