Lakehead
UNIVERSITY

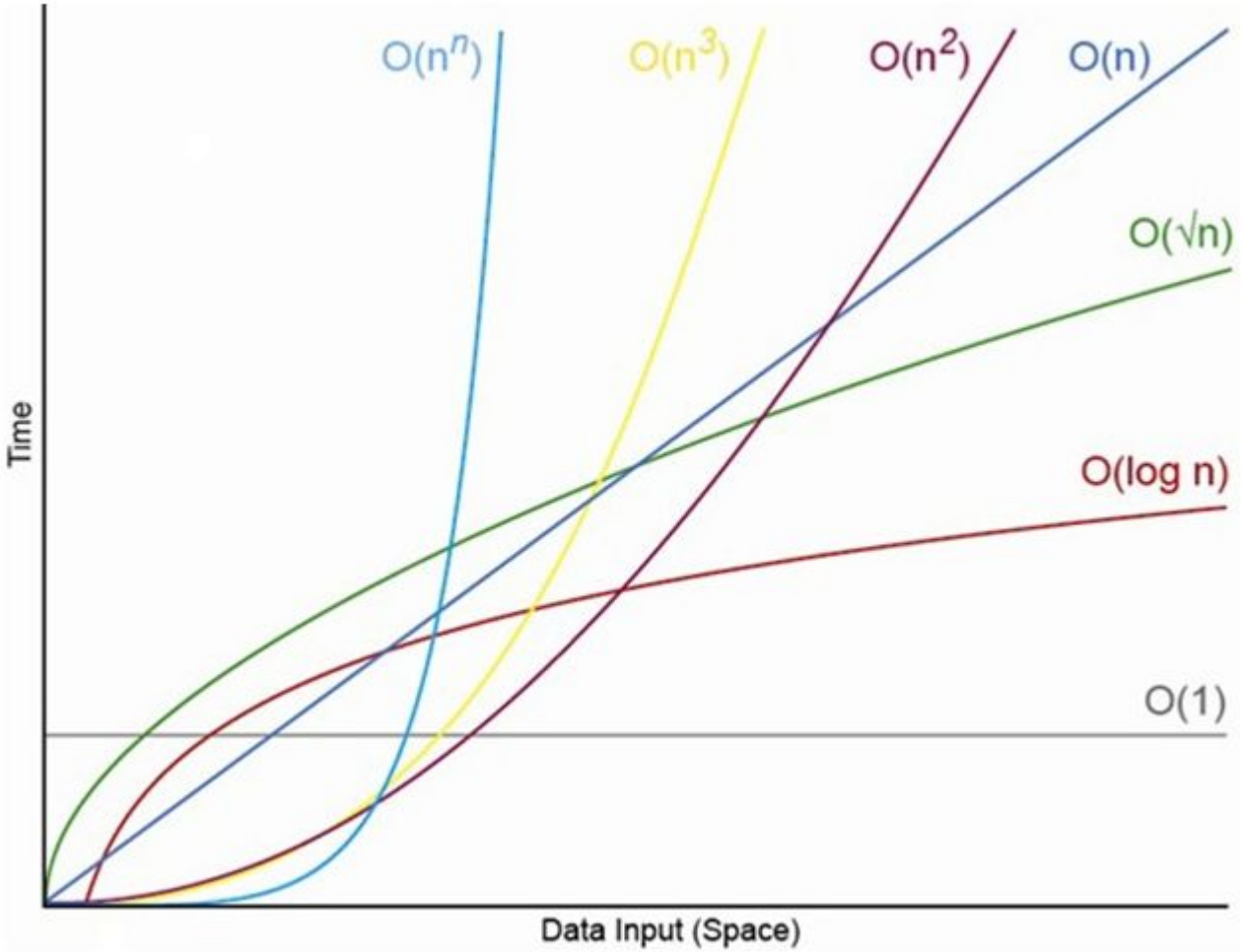
COMP 4433: Algorithm Design and Analysis

Dr. Y. Gu

March 27 2023 (Lecture 18)

NP-Completeness (Chapter 34)

Introduction



Introduction

In the following discussion, we use n to represents the size of the input of a problem.

- Almost all the algorithms we have studies so far are **polynomial-time algorithms**, i.e., their worst-case running time is $O(n^k)$ for some constant k.
- Generally, we think of problems that are solvable by polynomial-time algorithms as **being tractable**, or **easy**, and problems that require superpolynomial time as being **intractable**, or **hard**.
- There are a lot of problems which cannot be solved by polynomial time. A simple example is that a problem needs to print out 2^n data.

P vs. NP

P=?= NP

NP completeness

Informal Descriptions of P and NP

- The **class P** (represents **polynomial** time) consists of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved, i.e., find the solutions, in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.
- The **class NP** (represents for **non-deterministic polynomial** time) consists problems for which an answer is "verifiable" in polynomial time: If we were somehow given an answer (a.k.a. certificate of a solution), then we could verify the answer is correct in time polynomial ("quickly") in the size of the input to the problem.
- **P⊆ NP**

NP Example – Sudoku

	2	4	
1			3
4			2
	1	3	

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

		F			6	B				C		3			7
				5	A		F		E						6
C	B			2				4	7				E		D
D		7	3				C		B	F				4	A
	D	1	6		C		5		G		B	9			F
		9	E					C		4		7	2		5
	3		F	1	8			5		9			A	D	4
		G	7		D	4			3					6	
	5	D	8		B			2			6				E
		B						D			C		6	F	G
6	C			8		G	9	A	F		7		D	5	
G	F					6			9	E	5	1		8	3
		5	2		7			1		6	3			E	
9	E	3		C			A						F		
	6	8		4		9		E				B	5	G	
F					E		G				8				2

NP Examples – Sudoku

Consider Sudoku, a game where the player is given a partially filled-in grid of numbers and attempts to complete the grid following certain rules.

Given an incomplete Sudoku grid, **of any size**, is there at least one legal solution? Any proposed solution is easily verified, and the time to check a solution grows slowly (polynomially) as the grid gets bigger. However, all known algorithms for finding solutions take, for difficult examples, time that grows exponentially as the grid gets bigger. So, Sudoku is in NP (quickly checkable) but does not seem to be in P (quickly solvable). Thousands of other problems seem similar, in that they are fast to check but slow to solve.

NP-Completeness

A problem is in the class of NP-complete, if it is in NP and is as "hard" as any problem in NP. More formally, a problem is NP-complete when:

1. It is an NP problem, i.e., the correctness of each solution can be verified quickly (namely, in polynomial time)
2. A brute-force search algorithm can find a solution by trying all possible solutions.
3. The problem can be used to simulate every other problem for which we can verify quickly that a solution is correct. In this sense, NP-complete problems are the hardest of the problems to which solutions can be verified quickly.

If we could find solutions of some NP-complete problem quickly, we could quickly find the solutions of every other problem to which a given solution can be easily verified.

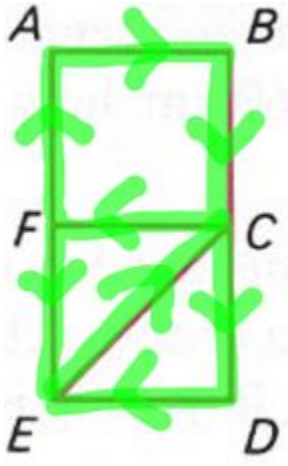
More Examples

Shortest vs. longest simple paths:

We saw that even with negative edge weights, we can find shortest paths from a single source in a directed graph G = (V, E) in O(|V| |E|) time.

However, merely determining whether a graph contains a simple path with at least a given number of edges is NP-complete.

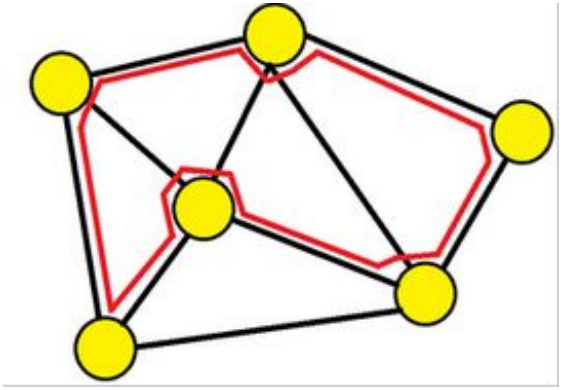
More Examples



Euler tour vs. Hamiltonian cycle:

An Euler tour of a connected graph G = (V, E) is a cycle that traverses each edge of G exactly once, although it is allowed to visit each vertex more than once. We can find the edges of the Euler tour in O(|E|) time.

A Hamiltonian cycle of a graph G = (V, E) is a **simple cycle** that contains each vertex in V . Determining whether a directed graph has a hamiltonian cycle very difficult to solve in general, it is NP-complete.

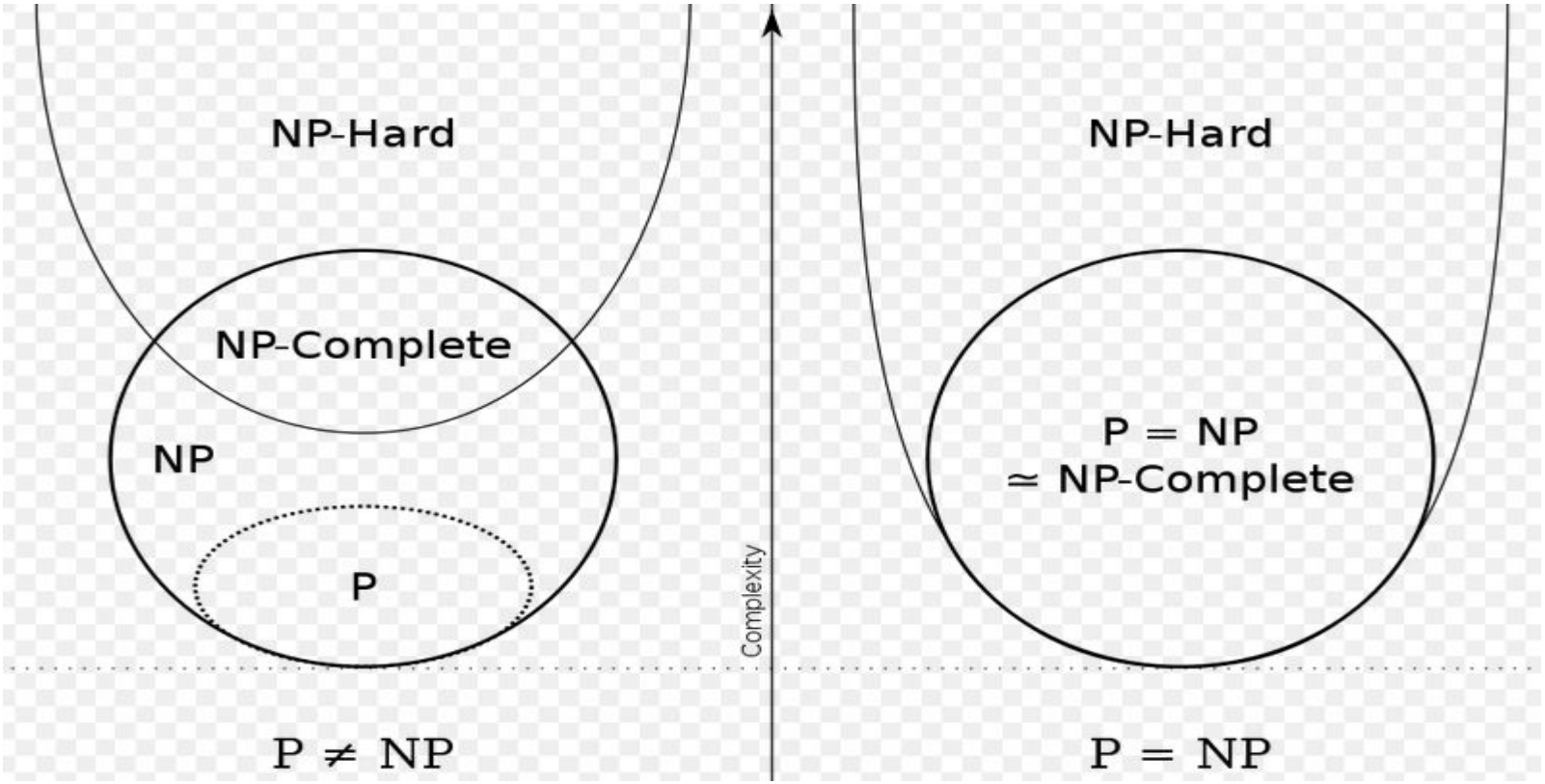


P =?= NP

The precise statement of the P versus NP problem was introduced in 1971 by Stephen Cook in his seminal paper "The complexity of theorem proving procedures"[3] (and independently by Leonid Levin in 1973[4]).

Although most scientists tends to believe P is not the equal to NP, this problem is still not solved and is one of the seven **Millennium Problems**.

P=NP? and NP-Complete



Further Discussion

Most theoretical computer scientists believe that the NP-complete problems are intractable, since given the wide range of NP-complete problems that have been studied to date, without anyone having discovered a polynomial time solution to any of them, it would be truly astounding if all of them could be solved in polynomial time.

To become a good algorithm designer, you must understand the rudiments of the theory of NP-completeness. If you can establish a problem as NP-complete, you provide good evidence for its intractability.

Further Discussion

Many problems of interest are optimization problems, in which each feasible (i.e., legal) solution has an associated value, and we wish to find a feasible solution with the best value.

NP-completeness applies directly not to optimization problems; however, but to decision problems, in which the answer is simply “yes” or “no” (or, more formally, “1” or “0”).

We can take advantage of a convenient relationship between optimization problems and decision problems. We usually can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized.

Relating Optimization with Decision

SHORTEST-PATH problem is the single-pair shortest-path problem in an unweighted, undirected graph.

A decision problem related to SHORTEST-PATH is PATH: given a directed/undirected graph G, vertices u and v, and an integer k, does a path exist from u to v consisting of at most k edges.

Relating Optimization with Decision

The relationship between an optimization problem and its related decision problem works in our favor when we try to show that the optimization problem is “hard”. That is because the decision problem is in a sense “easier”, or at least “no harder”.

In other words, if an optimization problem is easy, its related decision problem is easy as well.

Stated in a way that has more relevance to NP-completeness, if we can provide evidence that a decision problem is hard, we also provide evidence that its related optimization problem is hard.

Showing Problems to be NP-Complete

One important method used to show problems to be NP-complete is reductions.

Let us consider a decision problem A, which we would like to solve in polynomial time. We call the input to a particular problem an *instance* of that problem. For example, in PATH, an instance would be a particular graph G, particular vertices u and v of G, and a particular integer k.

Now suppose that we already know how to solve a different decision problem B in polynomial time.

Showing Problems to be NP-Complete

Finally, suppose that we have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:

- The transformation takes polynomial time.
- The answers are the same. That is, the answer for α is "yes" if and only if the answer for β is also "yes".

We call such a procedure a polynomial-time reduction algorithm.

Showing Problems to be NP-Complete

The polynomial-time reduction algorithm provides us a way to solve problem A in polynomial time:

1. Given an instance α of problem A, use a polynomial-time reduction algorithm to transform it to an instance β of problem B.
2. Run the polynomial-time decision algorithm for B on the instance β.
3. Use the answer for β as the answer for α.

As long as each of these steps takes polynomial time, all three together do also, and so we have a way to decide on α in polynomial time. In other words, by "reducing" solving problem A to solving problem B.

Showing Problems to be NP-Complete

For NP-completeness, we cannot assume that there is absolutely no polynomial time algorithm for problem A. The proof methodology is similar; however, in that we prove that problem B is NP-complete on the assumption that problem A is also NP-complete.

We define an abstract problem Q to be a binary relation on a set I of problem instances and a set S of problem solutions.

Showing Problems to be NP-Complete

For example, an instance for SHORTEST-PATH is a triple consisting of a graph and two vertices.

A solution is a sequence of vertices in the graph, with perhaps the empty sequence denoting that no path exists.

The problem SHORTEST-PATH itself is the relation that associates each instance of a graph and two vertices with a shortest path in the graph that connects the two vertices.

Since shortest paths are not necessarily unique, a given problem instance may have more than one solution.

Showing Problems to be NP-Complete

The theory of NP-completeness restricts attention to decision problems: those having a yes/no solution.

In this case, we can view an abstract decision problem as a function that maps the instance set ***I*** to the solution set {0, 1}.

For example, a decision problem related to SHORTEST-PATH is the problem PATH that we saw earlier.

If i = ⟨G, u, v, k⟩ is an instance of the decision problem PATH, then PATH.(i) = 1 (yes) if a shortest path from u to v has at most k edges, and PATH.(i) = 0 (no) otherwise.

After Class

After class reading: Part VII 34.1-34.2.