P, *NP*, and *NP-Complete* Problems

CS3230: Design and Analysis of Algorithms

Roger Zimmermann

National University of Singapore

Spring 2017

Chapter 11: Introduction

- Algorithms are powerful problem-solving tools.
- But, the power of algorithms is not unlimited.
- Some problems cannot be solved by any algorithm.
- Other problems can be solved, but not in polynomial time.
- Often, there is a lower bound on the efficiency of an algorithm.

Lower Bound Arguments

- We can look at the efficiency of algorithms in two ways:
  - Look at its asymptotic efficiency class (i.e., big-O notation).
    E.g., selection sort is $O(n^2)$,
    Tower of Hanoi algorithm is $O(2^n)$.
  - Look at its efficiency with respect to other algorithms that solve the same problem.

Lower Bound Arguments

- It is desirable to know the best possible efficiency any algorithm solving a specific problem may have.
- We would like to know such a lower bound.
- A lower bound is tight we already know an algorithm in the same efficiency class as the lower bound.
- If the lower bound is tight, then we can only hope for a constant-factor improvement at best.
- If a lower bound is not tight then either we may be able to find a faster algorithm or a better (i.e., “higher”) lower bound could be proven.

Trivial Lower Bounds

- The simplest method is based on counting
  - the number of items in the problem's input that must be processed, and
  - the number of output items that must be produced.
- Ex.: Generating all permutations of n items must be in $\Omega(n!)$ because the size of the output is $n!$.
- Ex.: Evaluation of a polynomial
  $p(x) = a_n x^n = a_{n-1}x^{n-1} + \ldots + a_0$
  at a given point x.
  Algorithm must be in $\Omega(n)$ because all coefficients $a_n, a_{n-1}, \ldots, a_0$ must be processed.

Lower Bound Methods

- Trivial lower bounds are often too low to be useful.
- Ex.: Traveling salesman problem lower bound is $\Omega(n^2)$, because there are $n(n-1)/2$ intercity distances. However, no known polynomial-time algorithm exists.
- Other methods:
  - Information-theoretic arguments: the amount of information an algorithm has to produce (example tool: decision trees).
  - Adversary arguments: adversary creates the most "time-consuming" path.
  - Problem reduction: given algorithm P, reduce alg. Q (with known lower bound) to algorithm P.

Problem Reduction

- Problems often used for establishing lower bounds by problem reduction.

Problem	**Lower Bound**	**Tightness**
sorting	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness problem	$\Omega(n \log n)$	yes
multiplication of n-digit integers	$\Omega(n)$	unknown
multiplication of square matrices	$\Omega(n^2)$	unknown

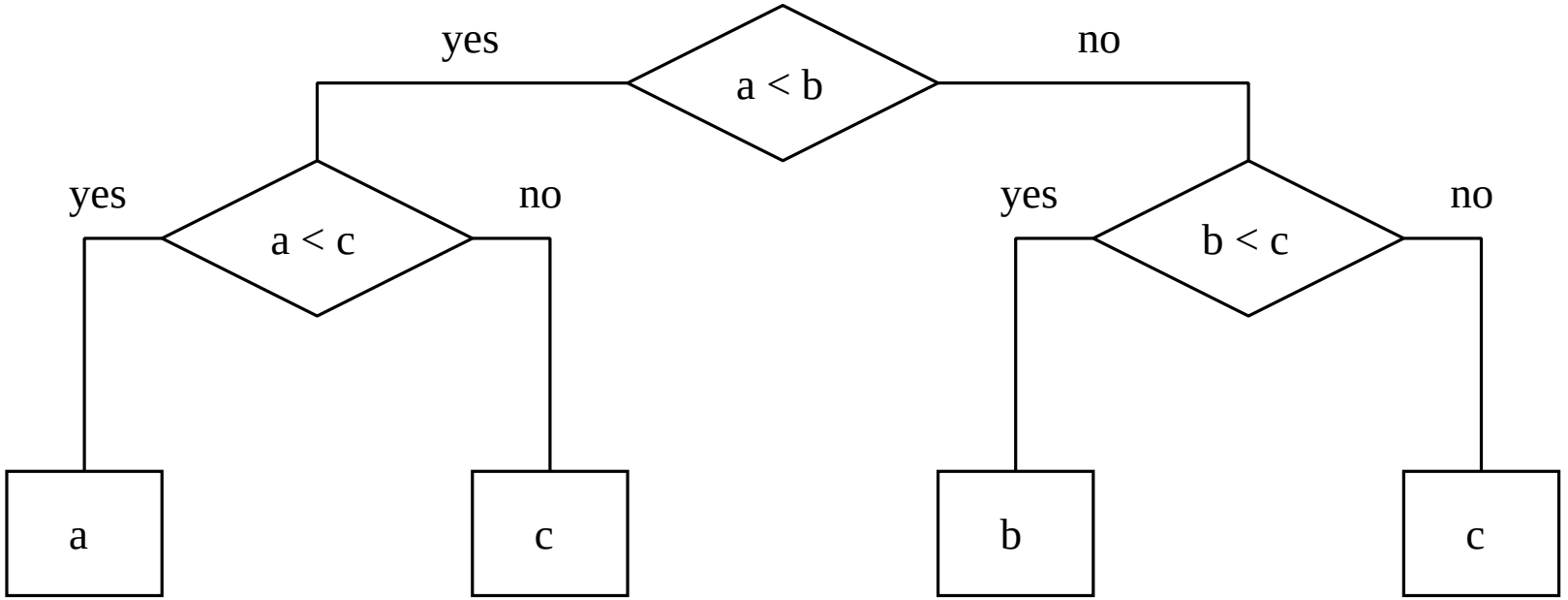
Decision Trees

- Many algorithms compare items of their input.
- Ex.: finding the minimum of three numbers $a, b,$ and c.
- Each node in the tree represents a comparison.
- Each leaf in the tree represents a possible outcome.
- Note: the number of leaves can be larger than the number of outcomes.
- The number of the comparisons in the worst case is equal to the height of the algorithm's decision tree

  $h \geq \lceil \log_2 l \rceil$.

Decision Trees

- Decision tree for finding a minimum of three numbers.



Decision Trees

- Each leaf represents a possible outcome of the algorithms run on some input size n.
- Note that the number of leaves can be greater than the number of outcomes.
- However, the number of leaves must be at least as large as the number of outcomes.
- The algorithm's work can be traced by a path from the root to a leaf.
- The number of comparisons made is equal to the number of edges along such a path.
- Hence, the number of comparisons in the worst case is equal to the height of the algorithm's decision tree.

Decision Trees for Sorting

- We can interpret an outcome of a sorting algorithm as finding a permutation of the element indices of an input list that puts the list's elements in ascending order.
- Hence, the number of outcomes for sorting an arbitrary n-element list is equal to $n!$.
- Therefore
  $C_{worst}(n) \geq \lceil \log_2 \ n! \rceil \approx \log_2 \sqrt{2\pi n}(n/e)^n \approx n \log_2 \ n$.
  Using Sterling's formula for $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ for large values of n.
- We can also analyze the average-case behavior
  $C_{avg}(n) \geq \log_2 \ n! \approx n \log_2 \ n$.

Algorithm Complexity

- An algorithm with $O(n^3)$ complexity is not bad because it can still be run for fairly large input sets in a reasonable amount of time.
- For the rest of this chapter we are concerned with problems with exponential complexity.
- Tractable and intractable problems.
- Study the class of problems for which no reasonably fast algorithms have been found, but no one has proven that fast algorithms do not exist.

Our Old List of Problems

- Sorting
- Searching
- Shortest path in a graph
- Minimum spanning tree
- Traveling salesman problem
- Knapsack problem
- Towers of Hanoi

Tractability

- An algorithm solves the problem in polynomial time if its worst-case time efficiency belongs to $O(p(n))$, where $p(n)$ is a polynomial of the problem's input size n.
- Problems that can be solved in polynomial time are called tractable.
- Problems that cannot be solved in polynomial time are called intractable.

Classifying a Problem's Complexity

- Is there a polynomial-time algorithm that solves the problem?
- Yes
  - We found a polynomial-time algorithm.
- No
  - Because it can be proved that all algorithms take exponential time.
  - Because it can be proved that no algorithm exists at all to solve this problem.
- Do not know
  - But if such algorithms were to be found, then it would provide a means of solving many other problems in polynomial time.

Types of Problems

- Optimization problem: construct a solution that maximizes or minimizes some objective function.
- Decision problem: a question that has two possible answers, yes and no.
  - Example: Hamiltonian circles: A Hamiltonian circle in an undirected graph is a simple circle that passes through every vertex exactly once. The decision problem is: Does a given undirected graph have a Hamiltonian circle?

Some More Problems

- Many problems will have decision and optimization versions.
- E.g.: Traveling salesman problem
  - Optimization problem: Given a weighted graph, find a Hamiltonian cycle of minimum weight.
  - Decision problem: Given a weighted graph and an integer k, is there a Hamiltonian cycle with total weight of at most k?

Some More Problems

- E.g.: Knapsack: Suppose we have a Knapsack of capacity W (a positive integer) and n objects with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ (where $w_1, \ldots, w_n$ and $v_1, \ldots, v_n$ are positive integers).
  - Optimization problem: Find the largest total value of any subset of the objects that fits in the Knapsack (and find a subset that achieves the maximum value).
  - Decision problem: Given k, is there a subset of the objects that fits in the Knapsack and has a total value of at least k?

Some More Problems

- Note: the general Knapsack problem can be solved in pseudo-polynomial time.

Some More Problems

- E.g.: Bin packing: Suppose we have an unlimited number of bins each of capacity one, and n objects with sizes $s_1, \ldots, s_n$, where $0 \leq s_i \leq 1$ (s_i are rational numbers).
  - Optimization problem: Determine the smallest number of bins into which the objects can be packed (and find an optimal packing).
  - Decision problem: Given, in addition to the inputs described, an integer k, do the objects fit into k bins?

Some More Problems

- E.g.: Graph coloring:
  - Coloring: Assign a color to each vertex so that adjacent vertices are not assigned the same color.
  - Chromatic number: The smallest number of colors needed to color G.
- Given: an undirected graph $G = \langle V, E \rangle$ to be colored
  - Optimization problem: Given G, determine the chromatic number.
  - Decision problem: Given G and a positive integer k, is there a coloring of G using at most k colors? If so, G is said to be k-colorable.

The Class P

- *P*: The class of decision problems that can be solved in $O(p(n))$, where $p(n)$ is a polynomial on n.
- Why use the existence of a polynomial time bound as the criterion?
  - If not, very inefficient.
  - Nice closure properties.
  - Machine independent in a strong sense.

The Class P

- What is the solvability of a decision problem?
  - Solvable/decidable in polynomial time.
  - Solvable/decidable, but intractable.
  - Unsolvable/undecidable problems: e.g., the halting problem.
  - No polynomial algorithm has been found, nor has the impossibility of such an algorithm been proved.

Alan Turing

- Brilliant British mathematician (23 June 1912 to 7 June 1954).
- Often considered the “father of modern computer science”.
- “Turing machine”.
- “Turing test”.
- Conjectured the halting problem in 1936.
- Turing award (sometimes called the “Nobel Price” for Computer Science).

The Halting Problem

- Given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.
- Proof by contradiction: Assume an algorithm A solved the halting problem, for any program P and input I,

$$A(P, I) = \begin{cases} 1, & \text{if program } P \text{ halts on input } I; \\ 0, & \text{if program } P \text{ does not halt on input } I. \end{cases}$$

The Halting Problem

- We can consider program P as an input to itself and use the output of A for (P, P) to construct a program Q as follows:

  $Q(P) =$

  $$\begin{cases} \text{halts, if } A(P,P) = 0, \text{ i.e., program } P \text{ does not halt on input } P; \\ \text{does not halt, if } A(P,P) = 1, \text{ i.e., program } P \text{ halts on input } P. \end{cases}$$

- Then on substituting Q for P, we obtain:

  $Q(Q) =$

  $$\begin{cases} \text{halts, if } A(Q,Q) = 0, \text{ i.e., program } Q \text{ does not halt on input } Q; \\ \text{does not halt, if } A(Q,Q) = 1, \text{ i.e., program } Q \text{ halts on input } Q. \end{cases}$$

- Contradiction!

The Class NP

- Informally, **NP** is the class of decision problems for which a given proposed solution for a given input can be checked quickly (in polynomial time) to see if it really is a solution.
- Formally, **NP** is the class of decision problems that can be solved by nondeterministic polynomial (NP) algorithms.

The Class NP

- A nondeterministic algorithm: A two-stage procedure that takes as its input an instance I of a decision problem and does the following:
  - "Guessing" stage: An arbitrary string S is generated that can be thought of as a guess at a solution for the given instance (but may be complete gibberish as well).
  - "Verification" stage: A deterministic algorithm takes both I and S as its input and checks if S is a solution to instance I, (outputs yes if S is a solution and outputs no or not halt at all otherwise).

The Class NP

- A *nondeterministic* algorithm solves a decision problem if and only if for every yes instance of the problem it returns yes on some execution.
- A *nondeterministic* algorithm is said to be polynomially bounded if there is a polynomial p such that for each input of size n for which the answer is yes, there is some execution of the algorithm that produces a yes output in at most $p(n)$ steps.

Example: Graph Coloring

- Nondeterministic graph coloring:
  - First phase: generate a string s, a list of characters $c_1 c_2 \ldots c_q$, which the second phase interprets as a proposed coloring solution.
  - Second phase: interpret the above characters as colors to be assigned to the vertices $c_i \rightarrow v_i$.

Example: CNF Satisfiability

- The problem: Given a boolean expression expressed in conjunctive normal form (CNF), can we assign values true and false to variables to satisfy the CNF?
- This problem is in *NP*.
- Nondeterministic algorithm:
  - Guess a truth assignment.
  - Check assignment to see if it satisfies CNF formula.
- Example:

$$(A \vee \neg B \vee \neg C) \wedge (\neg A \vee B) \wedge (\neg B \vee D \vee E) \wedge (F \vee \neg D)$$

Example: CNF Satisfiability

- Example:

  $(A \vee \neg B \vee \neg C) \wedge (\neg A \vee B) \wedge (\neg B \vee D \vee E) \wedge (F \vee \neg D)$

- Truth assignments:

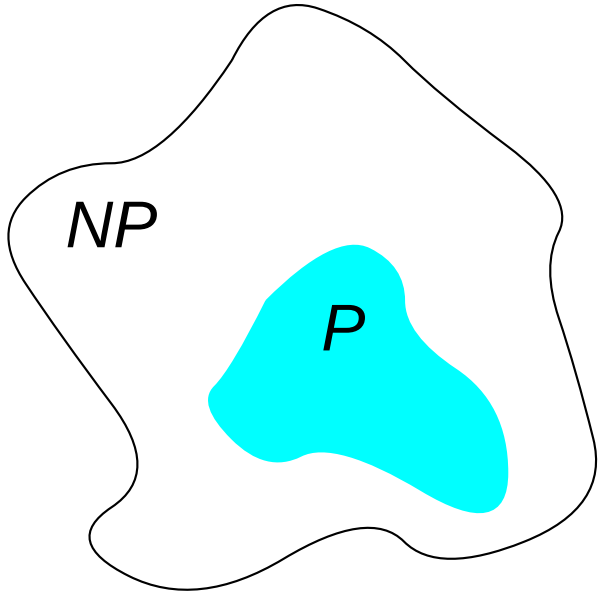
	A	B	C	D	E	F
1.	0	1	1	0	1	0
2.	1	0	0	0	0	1
3.	1	1	0	0	0	1
4.	...					

The Relationship between P *and* NP *(1)*

- $P \subseteq NP$
  - ($I \in P \Longrightarrow I \in NP$): Every decision problem solvable by a polynomial time deterministic algorithm is also solvable by a polynomial time nondeterministic algorithm.
- To see this, observe that any deterministic algorithm can be used as the checking stage of a nondeterministic algorithm.
  - If $I \in P$ and A is any polynomial deterministic algorithm for I, we can obtain a polynomial nondeterministic algorithm for I by using A as the checking stage and ignoring the guess. Thus $I \in P$ implies $I \in NP$.

The Relationship between P *and* NP *(2)*

- $P \stackrel{?}{=} NP$ ($NP \subseteq P$?)
  - Can the problems in *NP* be solved in polynomial time?
  - A tentative view:



NP-Completeness

- *NP*-completeness is the term used to describe decision problems that are the hardest ones in *NP*.
  - If there were a polynomial bounded algorithm for an *NP*-complete problem, then there would be a polynomial bounded algorithm for each problem in *NP*.
  - Examples:
    Hamiltonian cycle, Traveling salesman, Knapsack, Bin packing, Graph coloring, Satisfiability.

Informal Def. of NP-Completeness

- Informally, an *NP*-complete problem is a problem in *NP* that is as difficult as any other problem in this class, because by definition, any other problem in *NP* can be reduced to it in polynomial time.
- A decision problem D_1 is said to be polynomially reducible to a decision problem D_2 if there exists a function t that transforms instances of D_1 to instances of D_2 such that:
  1. t maps all yes instances of D_1 to yes instances of D_2 and all no instances of D_1 to all no instances of D_2, and
  2. t is computable by a polynomial-time algorithm.

Informal Def. of NP-Completeness

- If D_2 is polynomially solvable, then D_1 can also be solved in polynomial time!

- Definition: A decision problem D is said to be NP-complete if
  1. it belongs to class NP.
  2. every problem in NP is polynomially reducible to D.

An Example of Polynomial Reduction

- Problem P: Given a sequence of Boolean values, does at least one of them have the value *true*?
- Problem Q: Given a sequence of integers, is the maximum of the integers positive?
- Transformation T:

  $t(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_n)$, where $y_i = 1$ if $x_i =$ *true*, and $y_i = 0$ if $x_i =$ *false*

An Example of Polynomial Reduction

- A Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.
  - Hamiltonian circuit problem (HCP): Does a given undirected graph have a Hamiltonian cycle?
  - Traveling salesman problem (TSP): Given a weighted graph and an integer k, is there a Hamiltonian cycle with total weight at most k?

An Example of Polynomial Reduction

- A Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.
  - Transformation t:

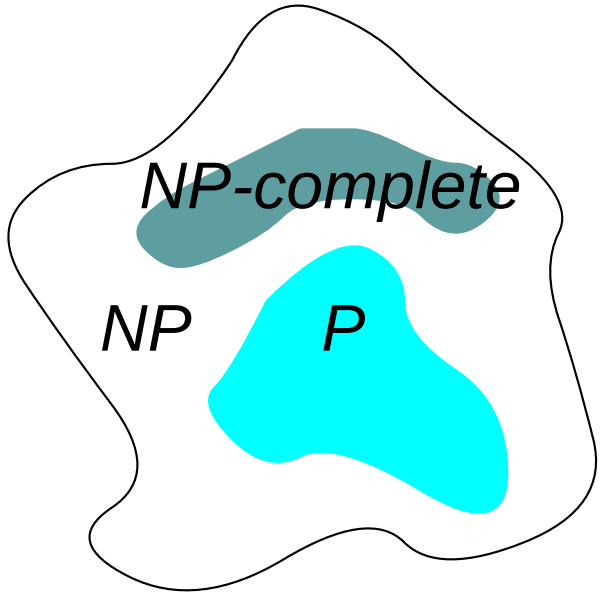
1. Map G, a given instance of the HCP to a weighted complete graph G', an instance of the TSP by assigning 1 as the edge weight to each edge in G and adding an edge of weight 2 between any pair of nonadjacent vertices in G.
2. Let k be n, the total number of vertices in G.

Formal Definition of NP*-Completeness*

- A decision problem D is said to be *NP*-complete if:
  1. It belongs to class *NP*.
  2. Every problem in *NP* is polynomially reducible to D.
- The class of *NP*-complete problems is called *NPC*.
- Cook-Levin Theorem (1971): discovered the first *NP*-complete problem, CNF-satisfiability problem.
- How to show a decision problem is *NP*-complete:
  - Show that the problem is *NP*.
  - Show that a known *NP*-complete problem can be transformed to the problem in question in polynomial time (transitivity of polynomial reduction).

Formal Definition of NP*-Completeness*

- Practical value of *NP*-completeness.



Example

- IEEE 802.11 and Graph Coloring.
- Wireless networks employing the IEEE 802.11 standard have become pervasive for both home and office use.
- Dependent on the physical layer used, current implementations have a certain number of non-overlapping radio frequencies (colors in the interference graph).
- Examples:
  - 802.11b and 802.11g have 3 frequencies.
  - 802.11a has 12 frequencies.

Pseudo-Polynomial Time

- If a problem is in *P*, then it *per force* has a pseudo-polynomial time algorithm. But *NP*-complete problems may also have pseudo-polynomial time algorithms (for example, the Knapsack problem).
- (Garey and Johnson) “A pseudo-polynomial time algorithm ... will display ‘exponential behaviour’ only when confronted with instances containing ‘exponentially large’ numbers, [which] might be rare for the application we are interested in. If so, this type of algorithm might serve our purposes almost as well as a polynomial time algorithm.”

  (where by “exponentially large numbers”, they mean “numbers with many digits”). Ex.: Knapsack: $O(nW)$.

Additional Resources

- *Complexity Zoo* maintained at Stanford University; currently lists 489 complexity classes.
- “The Status of the P versus NP Problem,” *Communications of the ACM* article; September 2009 issue, page 78, by Lance Fortnow.
- P-vs-NP prize: http://www.claymath.org; established in year 2000.