Com S 311 Section B
Introduction to the Design and Analysis of Algorithms

Lecture One for Week 11

Xiaoqiu (See-ow-chew) Huang

Iowa State University

April 6, 2021

Overview of NP-Completeness

The worst-case running time of a polynomial-time algorithm on inputs of size n is $O(n^k)$ for some constant k.

All algorithms we have studied so far in this course are polynomial-time algorithms.

There are problems that cannot be solved by any computer, no matter how much time is allowed.

For example, Halting Problem is to determine, given a program and an input to the program, whether the program will eventually halt when run with that input. This problem cannot be solved by any computer.

There are problems that can be solved, but not in polynomial time.

Overview of NP-Completeness

We study three classes of problems: P, NP, and NPC (NP-complete problems).

Informally, the class P consists of those problems that are solvable in polynomial time.

All the problems we have studied so far are in P.

Informally, the class NP consists of those problems that are ‘verifiable’ in polynomial time.

If we are somehow given a ‘certificate’ of a solution, then we could verify that the certificate is correct in polynomial time.

For example, a hamiltonian cycle of a directed graph $G = (V, E)$ is a simple cycle that contains each vertex in V.

Overview of NP-Completeness

The hamiltonian-cycle problem is to determine whether the directed graph G contains a hamiltonian cycle.

For this problem, a certificate would be a sequence $< v_1, v_2, v_3, ..., v_{|V|} >$ of $|V|$ vertices. We could check in polynomial time that (v_i, v_{i+1}) is in E for $i = 1, 2, 3, ..., |V| - 1$ and that $(v_{|V|}, v_1)$ is in E as well.

Thus, the hamiltonian-cycle problem is in NP.

Any problem in P is also in NP, since any problem in P can be solved in polynomial time without even being given a certificate.

The open question is whether or not P is a proper subset of NP.

Overview of NP-Completeness

Informally, a problem is **NP-complete** if it is in NP and is as 'hard' as any problem in NP. This means that if any NP-complete problem can be solved in polynomial time, then **every** problem in NP can be solved in polynomial time.

The class NPC consists of those problems that are NP-complete.

We will learn that the hamiltonian-cycle problem is NP-complete.

Hundreds of NP-complete problems have been studied to date, but no polynomial-time algorithm has been discovered for any of them.

As a computer scientist, you need to understand the concept of NP-completeness.

If a problem is NP-complete, you would do better to design an approximation algorithm, instead of an algorithm that solves the problem exactly.

Overview of showing problems to be NP-Complete

There are three key concepts in showing a problem to be NP-complete.

NP-completeness is confined to decision problems in which the answer is ‘yes’ or ‘no (or more formally, ‘1’ or ‘0’), instead of optimization problems.

For example, a decision problem related SHORTEST-PATH is PATH: given a directed graph G, vertices u and v, and an integer k, does G contains a path from u to v with at most k edges.

If a decision problem is hard, then its related optimization problem is hard as well.

In decison problems, we do not need to deal with how to represent optimal solutions.

Reductions

A second concept is to show that one problem is no harder or easier than another through reductions.

Consider a decision problem A, which we want to solve in polynomial time.

Suppose that we alreay know to solve a different problem B in polynomial time.

Assume that we have a method to transform any instance α of A into an instance β of B with the following properties:

The transformation takes polynomial time.
The transformation preserves the answers; the anser for α is ‘yes’ if and only if the answer for β is also ‘yes.’

Reductions

Such a method is called a polynomial-time reduction algorithm, which is a way to solve problem A in polynomial time:

First, given an instance α of problem A, transform it into an instance β of problem B with a polynomial-time reduction algorithm.

Second, run the polynomial-time algorithm for B on the instance β.

Third, report the answer for β as the answer for α.

These three steps decide on α in polynomial time.

By reducing problem A to problem B, the easiness of B is used to prove the easiness of A.

Reductions

We show how to use polynomial-time reductions to prove that no polynomial-time algorithm can exist for a particular problem B.

Assume that we know that no polynomial-time algorithm can exist for a decision problem A.

Also suppose that problem A can be reduced to problem B in polynomial time.

We use proof by contradiction to show that no polynomial-time algorithm can exist for problem B.

Assume that a polynomial-time algorithm exists for problem B.

Then the three-step procedure solves problem A in polynomial time, a contradiction.

A First NP-Complete Problem

We need a first NP-complete problem to prove a different problem NP-complete by reducing the first problem to the new problem.

The first problem is the circuit-satisfiability problem, which is to determine whether a boolean combinational circuit made of AND, OR, and NOT gates can produce an output of 1 (true) on some inputs.

$(\neg x_1 \vee x_2) \wedge (x_3 \vee \neg x_4)$

We will show that this problem is NP-complete later.

Polynomial Time

We first formalize our notion of polynomial-time solvable problems, which are regarded as tractable for philosophical reasons.

A first supporting argument for this view is that the polynomial-time computable problems in real worlds require time in polynomials of low degrees.

A second one is that a problem solvable in polynomial time on one model (such as the serial random-access machine) is also solvable in polynomial time on another (such as the Turing machine).

A third one is that the class of polynomial-time solvable problems is closed under common operations such as set operations, concatenation, and Kleene star.

Abstract Problems

Formally, we define an abstract problem Q to be a binary relation on a set I of problem instances and a set of problem solutions.

Here we use a binary relation, instead of a function, because a given problem instance may have more than one solution.

We define an abstract decision problem as a function that maps the instance set I to the solution set $\{0, 1\}$.

Encodings

We use encodings to represent problem instances to specify the input length of the problem instance.

An **encoding** of a set S of abstract objects is a mapping from S to the set of binary strings.

For example, the natural numbers $\mathbb{N} = \{0, 1, 2, 3, 4, ..., ...\}$ are encoded as the strings $\{0, 1, 10, 11, 100, ...\}$.

A **concrete problem** is a problem whose instance set is the set of binary strings.

An algorithm solves a concrete problem in time $O(T(n))$ if it can produce the solution on a problem instance i of length $n = |i|$ in $O(T(n))$ time.

Encodings

A concrete problem is **polynomial-time** solvable if there exists an algorithm to solve it in $O(n^k)$ time for some constant k.

The **complexity class P** is defined as the set of concrete decision problems that are polynomial-time solvable.

We use encodings to map abstract problems to concrete problems.

Let Q denote an abstract decision problem mapping an instance I to $\{0,1\}$.

We use an encoding e mapping I to $\{0,1\}^*$ (the set of all strings over $\{0,1\}$) to map the abstract problem Q to a related concrete decision problem denoted by $e(Q)$.

If the solution to an abstract-problem instance i in I is $Q(i)$ in $\{0,1\}$, then the solution to the concrete-problem instance $e(i)$ in $\{0,1\}^*$ is also $Q(i)$.

Expensive Encodings

The efficiency of solving a problem depends on the encoding.

For example, suppose that an integer k is to be provided to an algorithm with a running time of $O(k)$.

If the integer k is provided in unary (a string of k 1s), then the running time of the algorithm is $O(n)$ on the input of length n.

If the integer k is provided in the more natural binary representation of length $n = \lfloor lgk \rfloor + 1$, then the running time of the algorithm is $O(k) = O(2^n)$.

In this course, we do not use expensive encodings such as unary ones. We often use binary ones.

Polynomially Related Encodings

A function f from $\{0,1\}^*$ to $\{0,1\}^*$ is **polynomial-time computable** if there exists a polynomial-time algorithm A that, on any input x in $\{0,1\}^*$, produces $f(x)$ as output.

For some set I of problem instances, two encodings e_1 and e_2 are **polynomially related** if there exist two polynomial-time computable functions f_{12} and f_{21} such that for any i in I, we have
$f_{12}(e_1(i)) = e_2(i)$ and
$f_{21}(e_2(i)) = e_1(i)$.

Lemma 34.1
Let Q be an abstract decision problem on an instance set I, and let e_1 and e_2 be polynomially related encodings on I. Then $e_1(Q)$ is in P if and only if $e_2(Q)$ is in P.

Proof

Assume that $e_1(Q)$ is in P. Then there is an algorithm to solve $e_1(Q)$ in $O(n^k)$ time for some constant k.

Also assume that for any problem instance i, the encoding $e_1(i)$ can be computed from the encoding $e_2(i)$ in time $O(n^c)$ for some constant c, with $n = |e_2(i)|$.

To solve problem $e_2(Q)$, on input $e_2(i)$, we first convert $e_2(i)$ into $e_1(i)$ and then run the algorithm for $e_1(Q)$ on $e_1(i)$.

Converting encodings takes time $O(n^c)$, with $|e_1(i)| = O(n^c)$.

Solving the problem on $e_1(i)$ takes time $O(|e_1(i)|^k) = O(n^{ck})$, which is polynomial because both c and k are constants.

The other direction is similar.