The theory of NP-completeness

Chapter 34

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2019; distribution outside this group of students is NOT allowed.*

All the algorithms that we have examined since the beginning of the semester are algorithms that can be characterized by the following properties.

- They are **polynomial-time** (deterministic) algorithms, i.e. algorithms whose worst-case running time $T(n)$ on an input instance of size n is a polynomial function in n, i.e. there exist a positive constant $k > 0$ such that $T(n) = O(n^k)$.
- They are optimization algorithms that solve optimization problems, whose output is a solution to the problem (in sorting for example, the output is the permutation of the input keys that lists the keys in non-decreasing order).

In many cases we are interested in solving problems whose output is a **simple YES or NO.** These problems are called **decision problems**. The decision problem for sorting would print a YES if a permutation of the input keys exists that lists the key in non-decreasing order: naturally for sorting, no matter what the input key sequence is the output of the algorithm will be a YES, and the running time of the algorithm would be polynomial in the number of input keys. There are many decision problems for which we yet do not know whether there exist polynomial time deterministic algorithms that solve them. An example of a decision problem that seems to be quite difficult is **primality testing** and its **complementary problem** that of **factoring**.

```
     Primality-Testing                    Factoring
Input : Integer x                 Input:  Integer x
        n bits long                       n bits long
Output: YES if x is prime         Output: NO  if no y,z  exist such that x=y*z and y, z != 1,x.
        NO  if x is not a prime           YES if there exist y,z such that x=y*z
```

Factoring becomes a **decision problem**, as its output is a YES or a NO.

Factoring becomes an **optimization problem**, if the factors y, z are printed instead of the YES.

Input size for both problems is the length (in bits, digits) of integer x. For a number x the number of digits or bits $\mid x \mid$ required to represent x is $\mid x \mid = O(\lg x)$.

(c) Copyright A. Gerbessiotis. All rights reserved.

Verifying a solution

The high school algorithm for primality testing checks $O(\sqrt{x})$ potential factors of x by examining in turn all primes from 2 through $\sqrt{x}$.

Such an algorithm is too slow as its running time is not $O(\mid x \mid^k) = O(\lg^k x)$ (i.e. polynomial to the number of bits required to represent x) but $O((2^{|x|})^k) = O(x^k)$ (i.e. exponential to the number of bits required to represent x).

If $x = 1,000,000,000,001$, it requires about $\sqrt{x} = 1,000,000$ operations to test x's primality, even though x has 13 digits.

Verification of solution for decision problems.

Although factoring seems to be a difficult problem, verification of a solution to the factoring problem is easier. In order to verify the solution we need a **certificate** for the verification of the solution.

The certificate for factoring x is the pair (y, z) of factors of x. The length of the certificate is $O(\mid x \mid)$ as each of y, z is at most $\mid x \mid$ bits/digits long.

Given a certificate (y, z) for input x we can verify that $x = y \cdot z$ by multiplying y and z. Multiplication of two n bits integers requires $O(n^2)$ bit operations (elementary school algorithm).

Therefore even if **factoring** is difficult to solve (we have no deterministic algorithm that factors in poly-time to $\mid x \mid$), its verification however is an **easy problem, the well-known integer multiplication problem**, which has a polynomial-time (poly-time for short) algorithm.

(c) Copyright A. Gerbessiotis. All rights reserved.

Decision problems for optimization problems with poly-time algorithms

Another challenging decision problem is the Hamiltonian cycle problem; no known polynomial algorithm for the decision or optimization variant of the problem exists.

```
    Hamiltonian cycle
Input:  An undirected graph G=(V,E) with |V|=n
Output: YES if there exists a cycle that traverses all vertices of $V$ exactly once.
        NO  otherwise.
```

A certificate for the Hamiltonian cycle is a permutation of the n vertices that forms a Hamiltonian cycle. Given a permutation of the $n =\mid V \mid$ vertices we can in time $O(n)$ or $O(n^2)$ (depending on whether an adjacency matrix or adjacency list representation is used for the representation of G) verify that they form a cycle or not.

From now on, we would be considering only decision problems, not optimization problems.

Remark. All optimization problems with poly-time algorithms have corresponding decision problems that can also be solved in poly-time. Every time a solution for a problem that accepts a poly-time algorithm is found, an affirmative answer to the existence of a solution for the decision problem is given, and a certificate (the solution) for the decision problem is also provided.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Complexity class P

The complexity class P describes all decision problems that can be solved by a polynomial-time algorithm.

More formally, by describing the decision problem as a language with all strings (representation of the input) in the language generating a YES decision, and all strings not in the language generating a NO decision, we can more formally define P as follows.

Class P. P consists of all languages L that have a polynomial time algorithm A such that for any input x,

1. for any $x \in L$ then, $A(x)$ accepts (i.e outputs YES), and
2. for any $x \notin L$ then, $A(x)$ rejects (i.e outputs NO).

Complexity class NP Similarly class NP, is **loosely and liberally** defined as follows.

The complexity class NP describes all decision problems that given an input instance and a certificate of the solution, the existence/validity of the solution can be verified in polynomial time.

More formally,

Class NP. NP consists of all languages L that have a polynomial time algorithm A such that for any input x,

1. for any $x \in L$ then, there exists a y (a.k.a. certificate) such that $A(x, y)$ accepts (i.e outputs YES), and the size $\mid y \mid$ of y is bounded by a polynomial in the size of $\mid x \mid$, and
2. for any $x \notin L$ then, FOR ALL y, $A(x, y)$ rejects (i.e outputs NO).

The y in the definition serves the role of the certificate. For the Hamiltonian cycle problem the graph G is the input x, and if there is a solution then y serves the role of the certificate, which is a solution.

Classes P and NP continued

In other words, in order to prove that a decision problem or equivalently a language is in NP we work as follows: (a) "guess the solution (one always guesses right) of polynomial size in the input" and (b) then "verify that it is indeed a solution in polynomial time".

We show that factoring is in NP given input x: (a) We first guess the two factors p, q of x of polynomial size in $\mid x \mid$ ($p, q \leq x$ and thus $\mid q \mid, \mid q \mid \leq \mid x \mid$) and then, (b) we verify that they indeed factor x by performing the naive elementary-school multiplication of pq in $O(\mid x \mid^2)$ time to verify the solution.

Factoring and Hamiltonian-cycle are two decision problems that are in NP but we do not yet know whether they are in P or not.

Sorting is a decision problem that is in P and NP as well.

The complement $\bar{X}$ of a decision problem X is one that answers YES when X answers NO, and answers NO when X answers YES.

```
    Complement-of-Factoring
Input:  Integer x n bits long
Output: NO  if there exist positive y,z  such that x=y*z and y,z != 1,x.
        YES  otherwise.
```

This is the well-known problem of **primality testing**. If y and z exist NO means x is not-prime (i.e. it can be factored and thus is composite) and YES means that x can not be factored, i.e. it is prime.

```
    Primality-Testing
Input:  Integer x n bits long
Output: NO  if x is not prime (i.e. it is composite and can be factored)
        YES if x is prime.
```

(c) Copyright A. Gerbessiotis. All rights reserved.

Class co-NP

The class co-NP contains all those problems $\bar{X}$ whose complement X is in NP
Thus in order to prove that a problem Y is in $co-NP$ it suffices to prove that its complement $\bar{Y}$ belongs to NP.

Therefore primality-testing is in co-NP as its complement (i.e. factoring) is in NP.
The amazing thing is that primality-testing is also in NP (the certificates of a prime number x that verify the primality of x is what we call the Pratt numbers). Since August, 2002 primality-testing is also in P.
For every decision problem in P its complement is also in P.

Therefore the intersection of NP and $co-NP$ includes P, and the problem of primality testing (until August 2002 we didn't know whether it was in P; at that time it was proven to be in P). In the computer science field of **computational complexity** an interesting question is whether $P = NP$ which also means $P = NP = co-NP$. Most computer scientists think this is highly unlikely. Hamiltonian cycle, the traveling-salesman problem, many scheduling problems (e.g. decide course-classroom assignments optimally in a college) are problems that are known to be in NP but we do not know whether they accept a polynomial-time algorithm as a solution. All these problems are what we call **NP-complete** problems.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Definitions and Implications

Some (loosely-defined) definitions are required before we introduce NP-completeness.

Poly-time reductions Suppose we have two problems P_1 and P_2. P_1 is poly-time reducible to P_2 if we can find a poly-time reduction/function $f()$ of P_1 into P_2 such that x is a solution of P_1 if and only if $f(x)$ is a solution of P_2.

More formally, **Polynomial Reduction.** A polynomial reduction from a language L_1 to a language L_2 is a function f such that

1. there is a polynomial time algorithm that computes f, and
2. for all x, $x \in L_1$ if and only if $f(x) \in L_2$.

Definition NP-hard. A language L is NP-hard if for all $L' \in NP$ there is a polynomial reduction from L' to L.
Definition NP-complete A language L is NP-complete if L is in NP and L is also NP-hard.

Theorem Fundamental If L is NP-complete and we know that $L \in P$ (i.e. that L can be decided in polynomial time), then $P = NP$.

(c) Copyright A. Gerbessiotis.

Class RP. The class RP (Randomized Polynomial time) consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x,

1. for any $x \in L$ then, the probability that $A(x)$ accepts is at least $1/2$, and
2. for any $x \notin L$ then, the probability that $A(x)$ accepts is 0 (i.e. $A(x)$ rejects).

The choice of the bound for the error probability $1/2$ is arbitrary.

An RP algorithm is a Monte Carlo algorithm that can err only for an $x \in L$. This is what we call an one-sided error. The class co-RP errs only for $x \notin L$.

A problem belonging to both RP and co-RP can be solved by a randomized algorithm with zero-sided error i.e. a Las-Vegas algorithm.

Class ZPP. The class ZPP (Zero-error Probabilistic Polynomial time) is the class of languages that have Las Vegas algorithms running in expected polynomial time.

A Las Vegas algorithm is a Monte Carlo algorithm for $x \in L$ and a Monte Carlo algorithm for $x \notin L$. It never lies but its running time is probabilistic.

Class PP. The class PP (Probabilistic Polynomial time) consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x,

1. for any $x \in L$ then, the probability that $A(x)$ accepts is greater than $1/2$, and
2. for any $x \notin L$ then, the probability that $A(x)$ accepts is less than $1/2$.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

The casinos (continued)

The definition of PP is weak as we have no bound on how far from 1/2 the two probabilities are and therefore we cannot be sure whether we can reduce the error in a polynomial number of repeated trials. A more useful complexity class is BPP.

Class BPP. The class BPP (Bounded-error Probabilistic Polynomial time) consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input x,

1. for any $x \in L$ then, the probability that $A(x)$ accepts is greater than 3/4, and
2. for any $x \notin L$ then, the probability that $A(x)$ accepts is less than 1/4.

The 3/4 and 1/4 bounds are arbitrary and we can have $1/2 + \epsilon$ and $1/2 - \epsilon$ as long as ϵ is bounded below by the inverse of a polynomial in input size $\mid x \mid$.

We sometimes call a BPP algorithm an Atlantic-city algorithm.

(c) Copyright A. Gerbessiotis. All rights reserved.