LESSON TWELVE

NP-COMPLETE PROBLEMS

12.1 Introduction

In this lesson we shall focus on aspects of NP-Completeness that bear most directly on the analysis of algorithms. We also present techniques for determining when a problem is NP-complete. Many interesting computational problems are NP-complete, but no polynomial-time algorithm is known for solving any of them

The lesson covers:

- Lesson Objectives
- Overview
- Class P.
- Class NP
- NP-Hard
- Learning Activities
- Summary
- Further reading.

12.2 Lesson Objectives

By the end of this lesson you should be able to:

1. Understand NP-complete problems
2. Classify different types of problems.

12.3 Overview

There are two types of problems

- Decision problems
- optimization problems

Decision problems

- *In decision problems* we are trying to decide whether a statement is true or false.

Example 1

- A decision problem is a problem that you can always answer either "yes" or "no". For example, "Do you have a brother?". You can always answer "yes" or "no" for such questions.

Example 2

Hamiltonian cycles

- Given a directed graph, we want to decide whether or not there is a Hamiltonian cycle in this graph. This is a decision problem.

Optimization problems

- In *optimization problems* we are trying to find the solution with the best possible score according to some scoring scheme.
- Optimization problems can be either *maximization problems*, where we are trying to maximize a certain score, or *minimization problems*, where we are trying to minimize a cost function.

Example

TSP - The Traveling Salesman Problem.

- Given a complete graph and an assignment of weights to the edges, find a Hamiltonian cycle of minimum weight.
- This is the *optimization version* of the problem. In the *decision version*, we are given a weighted complete graph and a real number *c*, and we want to know whether or not there exists a Hamiltonian cycle whose combined weight of edges does not exceed *c*.

Inputs

Each of the problems discussed above has its characteristic input. For example,

- for the optimization version of the TSP, the input consists of a weighted complete graph;
- for the decision version of the TSP, the input consists of a weighted complete graph and a real number.

The input data of a problem are often called the *instance* of the problem. Each instance has a characteristic *size;* which is the amount of computer memory needed to describe the instance. If the instance is a graph of *n* vertices, then the size of this instance would typically be about *n(n-1)/2*.

Polynomial-time reducibility

Let *E* and *D* be two decision problems. We say that *D* is

polynomial-time reducible to E if there exists an algorithm

A such that

- *A* takes instances of *D* as inputs and always outputs the correct answer "Yes" or "No" for each instance of *D.*
- *A* uses as a subroutine a hypothetical algorithm *B* for solving *E.*
- There exists a polynomial *p* such that for every instance of *D* of size *n* the algorithm *A* terminates in at most *p(n)* steps *if each call of the subroutine B is counted as only m steps, where m is the size of the actual input of B.*

- The notion of polynomial-time reducibility can be extended to the case where only *E* is a decision problem.

An example of polynomial-time reducibility

Theorem: The Hamiltonian cycle problem is polynomial-time reducible to the decision version of TSP.

Proof: Given an instance G with vertices $v_1, \ldots, v_n$ of the Hamiltonian cycle problem, let H be the weighted complete graph on $v_1, \ldots, v_n$ such that the weight of an edge $\{v_i, v_j\}$ in H is 1 if $\{v_i, v_j\}$ is an edge in G, and is 2 otherwise. Now the correct answer for the instance G of the Hamiltonian cycle problem can be obtained by running an algorithm on the instance $(H,n+1)$ of the TSP.

Deterministic Turing Machine

- A deterministic Turing machine is the machine that we are used to normally. For example a computer is a deterministic Turing machine.

Nondeterministic Turing Machine

- A non-deterministic Turing machine is a machine that comes with unlimited parallelism. That is a computer that can run multiple threads in parallel.

Classification of problems

The subject of computational complexity theory is dedicated to classifying problems by how hard they are. There are many different classifications; some of the most common and useful are the following.

1) Unsolvable Problem
- No solution to a problem
2) Intractable Problem
- Requires exponential time.
3) P-Problem
- Problems that can be solved in polynomial time. ("P" stands for polynomial.)
4) NP-Problem
- Problems that can be quickly (in polynomial time) verified whether a solution is correct (without worrying about how hard it might be to find the solution).

12.4 The class P problems

A decision problem is in class P, if we can solve the problem in polynomial time using a deterministic Turing machine. It means that we can solve the problem very quickly. It shall finish the problem in some time n^k, where k is some constant.

For example,

- Finding the max element in an array,
- Checking whether a string is palindrome or not,
- Checking whether a number is prime or not,

A decision problem D is solvable in polynomial *time* or *in the class P*, if there exists an algorithm *A* such that

- *A* takes instance of *D* as inputs.
- *A* always outputs the correct answer “Yes” or “No”.
- There exists a polynomial *p* such that the execution of *A* on inputs of size *n* always terminates in *p(n)* or fewer steps.

12.5 The class NP problems

NP stands for "nondeterministic polynomial time" where nondeterministic is just a fancy way of talking about guessing a solution.

A problem is in NP if you can quickly (in polynomial time) test whether a solution is correct (without worrying about how hard it might be to find the solution).

Problems in NP are still relatively easy: if only we could guess the right solution, we could then quickly test it.

A decision problem is *nondeterministically polynomial-time solvable* or *in the class NP* if there exists an algorithm *A* such that

- *A* takes as inputs potential witnesses for “yes” answers to problem *D.*
- *A* correctly distinguishes true witnesses from false witnesses.
- There exists a polynomial *p* such that for each potential witnesses of each instance of size *n* of *D,* the execution of the algorithm *A* takes at most *p(n)* steps.

- Note that if a problem is in the class *NP*, then we are able to verify "yes"-answers in polynomial time, provided that we are lucky enough to guess true witnesses.

Example

- Verifying if a given solution for the Sudoku is valid or not?.
- Verification takes polynomial time (we are not finding solution but verifying a solution that was guessed).

NP-complete problems

A decision problem *E* is *NP-complete* if *every* problem in the class *NP* is polynomial-time reducible to *E*. The Hamiltonian cycle problem, the decision versions of the TSP and the graph coloring problem, as well as literally hundreds of other problems are known to be NP-complete.

It is not hard to show that if a problem *D* is polynomial-time reducible to a problem *E* and *E* is in the class *P*, then *D* is also in the class *P*. It follows that if there exists a polynomial-time algorithm for the solution of *any* of the NP-complete problems, then there exist polynomial-time algorithms for all of them, and *P= NP*.

A problem is NP-complete if the problem is both

- NP-hard, and
- In NP.

12.6 NP-Hard problems

Optimization problems whose decision versions are NP-complete are called *NP-hard.*

Theorem: If there exists a polynomial-time algorithm for finding the optimum in any *NP*-hard problem, then *P = NP*.

Proof: Let *E* be an *NP*-hard optimization (let us say minimization) problem, and let *A* be a polynomial-time algorithm for solving it.

Now an instance *J* of the corresponding decision problem *D* is of the form *(I,c),* where *I* is an instance of *E,* and *c* is a number.

Then the answer to *D* for instance *J* can be obtained by running *A* on *I* and checking whether the cost of the optimal solution exceeds *c.*

Thus there exists a polynomial-time algorithm for *D,* and *NP*-completeness of the latter implies *P= NP.*

A can be solved by reducing it to a different problem. That is Problem B can be reduced to Problem A. again given a solution to Problem A, you can easily construct a solution to Problem B. (In this case, "easily" means "in polynomial time.")

Therefore, if a problem is NP-hard, means you can reduce any problem in NP to that problem. This also means if you can solve that problem, then you can easily solve any problem in NP.

If therefore, we could solve an NP-hard problem in polynomial time, this would prove P = NP.

Example

- Solving a Sudoku puzzle.
- It takes a long time.

5.4 Learning Activities

(i) Show that the problem is in NP. (Show that a certificate can be verified in polynomial time)

(ii) Describe how to define NP-complete.

5.5 Summary

Several classic problems have been proved to be NP-complete. These problems include the following:

- Determining whether a given graph has Hamiltonian cycle.
- Determining satisfiability of a Boolean formula.
- Determining if a given set of numbers has a subset that adds up to a given target value.

Any problem which is in NP-complete means that we are unlikely to find a way to solve it in polynomial time. It was also proved that the famous TSP is NP-complete.

5.6 Further Reading

1. Hartmanis, J. (1982). Computers and intractability: a guide to the theory of NP-completeness (michael r. garey and david s. johnson). *Siam Review*, *24*(1), 90.
2. Agrawal, M., Allender, E., Impagliazzo, R., Pitassi, T., & Rudich, S. (2001). Reducing the complexity of reductions. *Computational Complexity*, *10*(2), 117-138.