COMPSCI 330: Design and Analysis of Algorithms November 28

Lecture 22: NP-Completeness and Reductions

Lecturer: Rong Ge *Scribe: Fred Zhang*

1 Overview

In this lecture, we study the limitations of algorithms and hardness of problems. In particular, we formalize the notion of easy problem as the class P and introduce a more general class NP. We then define NP-Complete problems, which are considered the hardest problems in NP. To compare the hardness of problems, we define the notion of reductions and, in particular, polynomial-time reductions.

2 Hardness of Problems

In the past, we have seen many algorithmic techniques, including divide-and-conquer, greedy, dynamic programming and linear programming and relaxations. Today we discuss the limitations of algorithms. A common scenario in algorithm design is that we cannot find a fast algorithm for some problem but not sure if this problem really does not have an efficient algorithm. Is it just that we cannot find the algorithm, or is the problem intrinsically hard that there is no fast algorithm?

2.1 Idea I: Special Case

To answer this question, it is natural to compare two problems A and B. We would like to know if A is a easier problem than B. Our first idea is to observe that if A is simply a special case of B, then A is easier than B.

Example 2.1. Shortest path problem:

- Problem A: shortest path with no negative edge;
- Problem B: shortest path with general edge weights.

Example 2.2. Solving linear program:

- Problem A: linear programming with canonical form LP;
- Problem B: linear programming with general LP.

But the problem with this idea is that we cannot compare problems that are not directly related. For example, one cannot compare minimum spanning tree with shortest path. Another issue is that it is hard to tell if A is slightly easier or much easier. In Example 2.1, A is indeed much easier than B, but in Example 2.2, any general LP can be converted to canonical form, so A is not strictly easier than B.

2.2 Idea II: Similarity in Statements

Let's take another idea. It seems natural to conjecture that if problems A and B have similar problem statements, they also have similar difficulty. But it is not true. There are problems with very similar statements but have different hardness.

Example 2.3. Shortest Path vs. Longest Path:

- Problem A: Given graph G with non-negative edge weights, find the **shortest** path from s to t that has no duplicate vertex;
- Problem B: Given graph G with non-negative edge weights, find the **longest** path from s to t that has no duplicate vertex.

Problem A is just shortest path, solvable by the Dijkstra's algorithm. A shortest path never repeats a vertex if all edge weights are nonnegative. Problem B, however, is different. It is known to be an *NP-Hard* problem, and we do not believe such a problem has efficient an algorithm.

Example 2.4. Graph coloring:

- Problem A (2-COLORING): Color vertices graph with 2 colors, such that the two endpoints of any edge have different colors.
- Problem B (3-COLORING): Color vertices graph with 3 colors, such that the two endpoints of any edge have different colors.

Problem A can be solved by BFS, where we color odd layers blue and even layers red. If this does not produce a valid coloring, then the graph is not 2-colorable. However, problem B is again NP-Hard, so it is unlikely there is an efficient algorithm for it.

3 Reductions

Now we would like to have a general, reliable way of comparing hardness of problems. This brings up to the topic of *reduction*. We say that problem A is *reduced to* problem B, if given a solution to B, we can also solve A. By solution to problem B, think of it as a magic function that is already implemented and can always return the correct answer, given an instance of problem B.

Example 3.1. Median finding and sorting

- Problem A: Find the median of an array.
- Problem B: Sort the array.

Problem A can be reduced to B, because we can first sort the array, and then return the middle element. In this case, we know median-finding is in linear-time, but sorting takes $O(n \log n)$ time, so problem B is indeed harder than A. But we do not need to know the best algorithm to reach this conclusion.

Example 3.2. LIS vs. LCS:

- Problem A: Longest increasing subsequence.
- Problem B: Longest common subsequence.

We argue that the LIS can be reduced to LCS. Suppose we are given a function $LCS(a[], b[])$ that returns the LCS between array a and b. Now using this function, one can easily solve LIS by the following algorithm.

Algorithm 1: Solving LIS using LCS

```
$b \to \text{sort}\,(a[])$ in increasing order.
**return** $LCS\,(a[], b[])$.
```

Why does this work? Array b contains all elements in A in increasing order. Then one can prove that

- Any common subsequence of $a[], b[]$ is an increasing subsequence of $a[]$.
- Any increasing subsequence of $a[]$ is a common subsequence of $a[], b[]$.

Therefore, any solution to the $LCS(a, b)$ problem is also a solution to the $LIS(a)$ problem, and *vice versa*, so our reduction gives solves LIS.

Example 3.3. Linear programming:

- Problem A: linear programming with canonical form LP;
- Problem B: linear programming with general LP.

Problem A can be reduced to B, because in a previous class we talked about how to convert every linear program into canonical form. In this case, A is also a special case of B, so these two problems are equally hard.

4 NP-Completeness

In the theory of *NP-Completeness*, we consider problems of two categories, easy and hard.

- Easy problems are problems that can be solved in polynomial time, such as $O(n), O(n \log n), O(n^3)$.
- Hard problem are problems that (we believe) cannot be solved in polynomial time.

In this lecture, we restrict ourselves to *decision problems.* These are problems that have Yes/No answers. We have considered many optimization problems in this course, but many of them have a corresponding decision problem.

Example 4.1. Shortest Path: find the shortest path from s to t. It corresponds to a decision version: Is there a path from s to t that has cost at most L?

Example 4.2. Minimum spanning tree corresponds to a decision problem: Is there a spanning tree with cost at most W?

4.1 P and NP

Formally, we call the set of decision problems solvable within polynomial-time the class P. These are considered the easy problems, and this includes almost all problems we have seen. Unfortunately, P does not contain all problems.

This brings us to a more general class called NP. This is all decision problems such that we can *verify* the correctness of a solution in polynomial time. By verification, we consider a *prover* and a *verifier.* The prover will provide with the verifier an answer and an explanation. Then to verify, the verifier takes the problem instance, the answer and explanation. If it is a YES instance, there should exist an explanation that can convince the verifier. Otherwise, no matter what explanation is provided, the verifier should not be convinced. Moreover, the verifier only takes polynomial time to draw a conclusion on whether the provided answer is correct or not.

Example 4.3 (3-Coloring)**.** Recall that in 3-Coloring, we ask if an input graph G can be colored using 3 colors such that no edge connects two vertices of same color.

- If the instance is YES, then the prover can simply provide a coloring, and the verifier can check the solution in polynomial and in fact linear time.

- If the instance is NO, then there is no valid coloring, so the verifier would never accept.

Example 4.4 (Shortest Path)**.** Is there a path from s to t that has cost at most L? If there is such a path, the prover can provide it. Given a path, the verifier can just check in polynomial time that (1) it is a path from s to t, and (2) the total length is at most L.

Example 4.5 (Composite Number)**.** Is number x a composite number? If it is, then the prover can give $x = yz$. Then the verifier can check in polynomial time that (1) $yz = x$ and (2) y, z are integers between 2 and x.

4.2 Polynomial Time Reductions

We now observe that all problems in P are also in NP. The intuition is that if I can solve the problem, I do not need an explanation. It is open whether they are equal. Most people would say no because of the intuition that solving a problem is harder than verifying a solution.

In order to characterize problems in NP, we use the notion of *polynomial-time reduction.*

Definition 4.1 (Polynomial-Time Reduction)**.** We say that a problem A *reduces to* B *in polynomial time* if there is a polynomial time algorithm that can transform an instance of problem A to an instance of problem B in polynomial time, such that the answers are the same, and such an algorithm this is called a *polynomial-time reduction.*

If there is a polynomial time reduction from A to B, we say A is easier than B.

Corollary 4.1. If A reduces to B in polynomial time and B can be solved in polynomial time, then A can also be solved in polynomial time,

Now we are ready to introduce the concept of *NP-Complete* problem. These are the hardest problems in NP. Formally, if A is in NP, and B is a NP-Complete problem, then A can be reduced to B.

Definition 4.2. A problem B is said to be a *NP-Complete* if for every problem A in NP, there exists a polynomial-time reduction from A to B.

Corollary 4.2. If any of the NP-Complete problems can be solved in polynomial time, then $P = NP$.

A famous result by Cook and Levin shows that such a complete problem exists. This is a problem called Circuit Satisfiability (CIRCUIT-SAT), which we will discuss next time.

Theorem 4.3 (Cook-Levin)**.** CIRCUIT-SAT is NP-Complete.