COMPSCI 330: Design and Analysis of Algorithms April 16

Lecture 22: Complexity and Reductions

Lecturer: Rong Ge *Scribe: Chenwei Wu*

1 Overview

In this lecture, we will learn the basic knowledge of **Complexity and Reductions** and see many examples of them. We begin with defining the hardness of problems and then study reduction, which is the standard way to compare the hardness of problems. We then formalize the notion of easy problem as the class P and introduce a more general class NP. Finally, we define NP-Complete problems, which are considered the hardest problems in NP.

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 are not sure whether this problem really does not have an efficient algorithm. It is possible that the problem does not have a fast algorithm, but it is also possible that there is a fast algorithm for the problem but we just haven't found it. In order to distinguish these two scenarios, we begin with comparing the hardness of two problems. By hardness of a problem we mean the time complexity of the best algorithm that solves the problem. However, how can we know this when we even don't know the best algorithms for these problems?

Note: In this lecture, for simplicity, when we say "easier", we are actually saying "strictly easier or equally hard", unless otherwise stated.

2.1 Idea 1: Special Case

To answer this question, our first idea is to observe that if A is simply a special case of B, then A is easier than B.

Example 1. Shortest path problems:

- Problem A: Shortest path on graphs with no negative edge
- Problem B: Shortest path on graphs with general edge weights

Example 2. Solving linear program:

- Problem A: Linear Program in canonical form
- Problem B: A general linear program

In these cases, we can see that problem A is a special case of problem B, indicating that problem A is easier than problem B. However, using this method, we cannot compare two problems that are not directly related, e.g., 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 than B. In Example 1, A is indeed much easier than B, but in Example 2, any general LP can be converted to canonical form, so A is not strictly easier than B.

2.2 Idea 2: 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 3. Shortest path vs. Longest Path:

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

In Example 5, Problem A is polynomial-time solvable, e.g., by Dijkstra's algorithm. However, Problem B is NP-Hard, i.e., people don't believe that this problem has an efficient algorithm.

Example 4. Graph coloring:

- Problem A (2-Coloring): Color vertices graph with 2 colors, such that all edges connect two vertices of different colors
- Problem B (3-Coloring): Color vertices graph with 3 colors, such that all edges connect two vertices of different colors

The same things happen in Example 4: Problem A can be solved in polynomial time using a BFS/DFS algorithm, but Problem B is NP-Hard.

In the two examples above, we have known that even if the statements of the two problems are really similar, the hardness of these problems can be completely different.

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. Informally, reduction can be defined as

Definition 1. Problem A can be reduced to Problem B, if one can design an algorithm for Problem A using an algorithm for Problem B as a subroutine. Equivalently, we can write "$A \leq B$".

Corollary 1. Runtime of $A \leq$ Runtime of B + Time of Reduction.

Now let's see some examples of reductions:

Example 5. Median finding and sorting:

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

The reduction is shown below:

```
Median(X)
{
  Sort(X);
  return X[(X.length+1)/2];
}
```

Algorithm 1: Median finding

In this case, we have an $O(n \log n)$ algorithm for sorting an array, and the reduction time is $O(1)$. Thus, using Corollary 1, we can conclude that the runtime of A is no larger than the runtime of B. This also matches our experience because the sorting algorithm takes $O(n \log n)$ time but median finding can be done in linear time.

Example 6. LIS vs. LCS:

- Problem A (LIS): Find the longest increasing subsequence of an array
- Problem B (LCS): Find the longest sequence that is a subsequence of both input arrays

For example, for array $\{4, 5, 2, 3, 6, 9, 7\}$, one of the longest increasing subsequences is $\{2, 3, 6, 7\}$. For arrays $\{3, 2, 1, 5, 4\}$ and $\{2, 5, 4, 3, 1\}$, the longest common sequence is $\{2, 5, 4\}$. Now we argue that the LIS can be reduced to LCS. Suppose we are given a function LCS(a[]; b[]) that returns the LCS between arrays a and b. Now using this function, one can easily solve LIS by the following algorithm:

```
LIS(X)
{
  Y = Sort(X);
  return LCS(X, Y);
}
```

Algorithm 2: Solving LIS using LCS

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

- Every common subsequence of X and Y is an increasing subsequence of X
- Every increasing subsequence of X is also a common sequence of X and Y

Thus, we can conclude that $LIS \leq LCS$. This also matches our experience because LIS can be solved in $O(n \log n)$ time, but the most efficient known algorithm for LCS runs in $O(n^2)$.

3.1 General Recipe for Reduction

Here we provide a general framework for doing reduction:

```
A(X)
{
  do something (per-processing);
  call B;
  do something else (post-processing);
}
```

Algorithm 3: Reducing A to B

From this framework, we can see that

- If we can solve B, we can also solve A
- If the additional steps in the program are easy (faster than the running time of B), we can claim A is easier than B because of Corollary 1

Thus, reduction can not only be used to compare the difficulties of problems, but can also be used to construct a solver for a new problem using a solver for a known problem as a subroutine.
Note: The reverse of Corollary 1 is not necessarily true, i.e., if the best algorithm for problem A is more efficient than the best algorithm for problem B, then this doesn't mean that A can be reduced to B.

4 Complexity

Complexity means how hard it is to solve a problem, and a **Complexity Class** is a set of problems that are "similar" in difficulty. In this section, we mainly consider problems of two complexity classes, 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 problems 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 problems in this course, but many of them can be converted into decision problems:

Example 7. 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 8. Minimum spanning tree corresponds to a decision problem: Is there a spanning tree with cost at most W?

We note that the decision version of a problem is a special case of the original problem, but most of the times the difficulty of solving them are very similar.

4.1 P vs. NP

Definition 2. P (polynomial time) is the class of **decision problems** that can be solved in polynomial time.

The problems in P are considered to be easy problems, and P included almost all problems we have talked about in this course. Unfortunately, P does not contain all problems, this brings us to a more general class called NP (nondeterministic polynomial time).

Definition 3. NP (nondeterministic polynomial time) is the class of **decision problems** that can be **verified** in polynomial time.

Think of Sudoku as an example: Given the solution, it is very easy to check that it is correct, but it can be much harder to find the solution given the puzzle.

Formally, by verification, we consider a *prover* and a *verifier*. The prover 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. The pseudocode of verification is given below:

```
Verify(input, answer, explanation)
{
if answer == YES then
    if explanation supports answer then
        accept;
    end
    else
        reject;
    end
end
}
```

Algorithm 4: Verification

From the pseudocode we can see that if the true answer is yes, then there exists an explanation that makes Verify() accept. However, if the true answer is no, then no matter what explanation is, Verify() will always reject. Now let's see some examples of NP problems:

Example 9. (Shortest Path). Is there a path from s to t that has cost at most L? If the answer is Yes, i.e., there is such a path, the prover will also provide the path. 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 10. (Composite Number). Is number x a composite number? If the answer is Yes, then the prover will give $x = yz$. Given the solution, the verifier can check in polynomial time that (1) $x = yz$ and (2) y, z are integers between 2 and x.

Example 11. (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 answer to an instance is Yes, then the prover can simply provide a valid coloring, and the verifier can check the solution in polynomial time that each vertex is colored and each pair of adjacent vertices have different color.

Given the definition of P and NP, we can conclude that P⊆NP because if one can solve the problem in polynomial time, he doesn't need an explanation at all. It is open whether they are equal, but we believe that P is not equal to NP. The intuition for this is that solving a problem is harder than verifying a problem.

4.2 Polynomial Time Reductions

In order to characterize problems in NP, we use the notion of **polynomial-time reduction**. Similar to the recipe of general reductions, the recipe for a polynomial time reductions are shown below:

```
A(X)
{
  spend polynomial time to prepare an input Y for problem B;
  return B(Y);
}
```

Algorithm 5: Polynomial time reduction from A to B

Note that this reduction can only take polynomial time, and cannot do any post-processing.
Now let's formally define polynomial time reductions:

Definition 4. We say that a problem A reduces to B in polynomial time if there is a polynomial time algorithm that can transform an instance X of problem A to an instance Y of problem B in polynomial time, such that if the answer to X is YES, answer to Y is also YES, and if answer to X is NO, answer to Y is also NO. Such an algorithm is called a polynomial-time reduction.

Corollary 2. 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.

4.3 NP Completeness

Now we are ready to introduce the concept of NP-Complete problem. These are the hardest problems in NP.

Definition 5. A problem B is NP-Complete if $B \in$ NP and for every $A \in$ NP, there is a polynomial time reduction from A to B ($A \leq B$).

This directly leads to the following corollary:

Corollary 3. 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. (Cook-Levin) Circuit-SAT is NP-Complete.

4.4 Harder Problems

There are harder problems that are not in NP.

Example 12. (Halting Problem) Given the code of a program, check if it will terminate or not.

This Halting Problem is not even solvable, i.e., there doesn't exist any algorithm that solves this problem no matter how long we allow the algorithm to run.

Example 13. (Playing Chess) Given a chess board with $n \times n$ size and $4n$ chess pieces, decide whether the first player can always win.

This is believed to be PSPACE-Complete and very unlikely to be in NP.