CS3230 Design & Analysis of Algorithms **March 31 & April 7, 2022**

Lectures 10 & 11: Reductions and Computational Complexity

Lecturer: Warut Suksompong

The goal of these two lectures is to introduce to you the basics of the theory of computational complexity. We will look at polynomial-time reductions, a tool for relating the complexities of different problems. We will then describe a series of reductions between various problems. Finally, we discuss the complexity classes P, NP and NP-complete and connect them to the previous discussion.

Remark. For these lectures, it is essential to be clear about the distinction between *problems* and *algorithms*. So far in this module, we have mostly[1] focused on the asymptotic complexity of particular algorithms. The goal here, on the other hand, is to study, for a particular problem, the runtime of the fastest correct algorithm for it. Also, throughout these lectures, for the sake of simplicity, we will only consider deterministic algorithms, not randomized ones.

10.1 Problems

As defined in Lecture 1, a computational problem is described by a collection of pairs (x, S) where x describes a valid input (instance) and S is a set of valid solutions for the input x. Since our goal here is to relate different problems to each other, it helps to have a more uniform way to describe problems.

A *decision problem* is a problem where for any input x, the corresponding solution is either YES or NO. An input x whose solution is YES is said to be a *YES-instance*, and similarly those whose solution is NO is said to be a *NO-instance*. We'll typically use capital letters to name decision problems.

Example 1. A valid input to the KNAPSACK decision problem is of the form $\langle (w_1, v_1), (w_2, v_2), \ldots, (w_n, v_n), W, V \rangle$ where all the parameters are non-negative integers. Such an input is a YES-instance if there exists a subset A of $\{1, \ldots, n\}$ such that $\sum_{i \in A} w_i \leq W$ and $\sum_{i \in A} v_i \geq V$.

Contrast the decision problem to the usual formulation where V is not present, and one wants to find the largest total value V for which a knapsack exists satisfying the weight bound W.

Example 2. A valid input to the MAXIMUM SPANNING TREE decision problem is a description of an edge-weighted undirected graph G as well as a parameter k. Such an input is a YES-instance if there exists a spanning tree of weight at least k and NO-instance otherwise. The usual formulation does not have the k, and its output is a spanning tree of the graph of maximum possible weight.

Example 3. A valid input to the SORTING decision problem is of the form $\langle a_1, a_2, \ldots, a_n, i, j \rangle$ where $i, j \in \{1, \ldots, n\}$. The input is a YES-instance if the i-th smallest[2] element among $a_1, \ldots, a_n$ is a_j, and NO-instance otherwise.

The natural formulation of the problems which is not a decision problem is often called a *search problem*. In these examples, it seems obvious that the decision problems are easier than their search counterpart. Intuitively, this is because any algorithm that solves the search problem must be solving the decision problem inside. This feels correct, but how do you argue this formally?

The mathematically correct way to argue this is to say that you can take an algorithm that solves the search problem and modify it slightly so that it solves the decision problem. For example, if there is an

[1] Only in the very first lecture, when we discussed adversarial arguments for restricted computational models, did we consider the best runtime obtained by any algorithm for a specific problem.

[2] Here, I am assuming for convenience that $a_1, \ldots, a_n$ are distinct.

algorithm $\mathcal{A}_{\text{sor}}$ that solves the usual sorting problem, you can use $\mathcal{A}_{\text{sor}}$ to also solve the SORTING decision problem: use $\mathcal{A}_{\text{sor}}$ to sort $a_1, \ldots, a_n$ as $b_1, \ldots, b_n$, and output YES if $b_i = a_j$, and NO otherwise. Call this algorithm for the decision problem $\mathcal{B}_{\text{sor}}$. The extra overhead is only $O(n)$. So, if T_s and T_d are the running times of $\mathcal{A}_{\text{sor}}$ and $\mathcal{B}_{\text{sor}}$ respectively, we get:

$$T_d \leq T_s + O(n).$$

This is our first example of a *reduction*, a concept we formally introduce below. The idea is that we show that the decision problem can be solved by first invoking an algorithm for the search problem and then doing some overhead operations. Similarly, you can get decision-to-search reductions for the knapsack and maximum spanning tree problems as well with small overhead.

Although it seems counter-intuitive, often, there is an efficient reduction in the search-to-decision direction too. In other words, for many problems, an algorithm for the decision version can be used to solve the search version with small overhead.

Example 4. Let $\mathcal{B}_{\text{sor}}$ be an algorithm to solve the SORTING decision problem. Then, it can be used to solve the usual sorting problem as follows: run $\mathcal{B}_{\text{sor}}$ for every pair $i, j \in [n]$. Clearly, we can then recover what the sorted array is. The running time is $O(n^2)$ times the running time of $\mathcal{B}_{\text{sor}}$. However, this is not particularly efficient.

Example 5. Let $\mathcal{B}_{\text{ks}}$ be an algorithm to solve the KNAPSACK decision problem. Now, suppose we want to use $\mathcal{B}_{\text{ks}}$ to solve the usual knapsack problem, where we want to output the largest value possible under the weight constraint. Consider the following algorithm $\mathcal{A}_{\text{ks}}$:

```
A_ks(w_1, v_1, ..., w_n, v_n, W)
1  beg = 0
2  end = sum_{i=1}^{n} v_i
3  while beg < end
4       mid = ⌊(beg + end)/2⌋
5       if B_ks(w_1, v_1, ..., w_n, v_n, W, mid) = YES
6            beg = mid
7       else
8            end = mid − 1
9  return beg
```

The algorithm $\mathcal{A}_{\text{ks}}$ does binary search to find the largest value of V for which $\mathcal{B}_{\text{ks}}(w_1, v_1, \ldots, w_n, v_n, W, V) =$ YES. If T_s and T_d denote the running times of $\mathcal{A}_{\text{ks}}$ and $\mathcal{B}_{\text{ks}}$ respectively, then $T_s \leq O(T_d \cdot \lg(\sum_i v_i))$.

10.2 Polynomial Running Time

Since in these lectures, we compare different problems to each other, we need a uniform measure of efficiency. As we have discussed earlier, we measure running time asymptotically as a function of the input size. But inputs can be in different formats for different problems. They can be arrays of integers (e.g., sorting), strings (e.g., longest common subsequence), graphs (e.g., shortest path), or even computer programs (e.g., compiling into assembly). How do we have a meaningful measure of input size that is valid for all problems?

To do this, we assume that all problems have their inputs encoded as binary strings. The exact encoding is either assumed as a matter of convention or is explicitly specified. If the input is an integer, then it is represented in binary in the usual way. If the input is a string, then it is represented as a concatenation[3] of the encodings of characters, where each character is encoded as an integer in a specific way involving a

[3] If you care about details, you may be worried about how concatenation can be represented. For simplicity, you can assume that there is a special symbol to denote concatenation (equivalent of comma or whitespace). The alphabet is not exactly binary, but all of our discussion remains valid if the alphabet size is 3 or 4, instead of 2.

constant number of bits (e.g., ASCII encoding with 7 bits, UTF-32 with 32 bits). If the input is a graph, then it can be represented by its adjacency matrix. In other cases, the encoding needs to be specified so that the input size is meaningful.

Definition 10.2.1. We say a decision problem X is *in* P if there exists a correct (deterministic) algorithm for X that runs in time $O(n^k)$ for some constant k, where n is the size of the input to X.

P is an example of a *complexity class*, as it is the family of all decision problems that can be solved in polynomial time. We will see other examples of complexity classes later.

Membership in P is considered as a hallmark for a problem to be efficiently solvable. Another phrase used for problems in P is *computationally tractable*. There is reason to complain that equating efficiency with membership in P is overly liberal. If the most efficient algorithm for a problem runs in time n^{100}, then the problem is in P but the algorithm is hardly usable. While this is true, it seems that in practice, there are few such problems. Most natural problems that are known to be in P also have truly efficient and practical algorithms where the constant k is small. So, P is a meaningful designation. Also, people do study finer distinctions between problems in P, but such a discussion is beyond the scope of the present module.

For problems that take in numerical inputs, there is an additional notion that is useful. An algorithm for such a problem runs in *pseudo-polynomial* time if the runtime is polynomial in the *value* of the numerical input. For example, the dynamic programming algorithm from Lecture 8 for Knapsack runs in pseudo-polynomial time.

10.3 Reductions

We informally discussed reductions above, but now let us define them formally. The notion is easiest to define for decision problems.

Decision Problems

Definition 10.3.1. Given two decision problems X and Y, we say that X *reduces in polynomial time to* Y if there exists a polynomial-time algorithm $\mathcal{R}$ that takes as input an instance x of problem X and outputs an instance y of problem Y such that x is a YES-instance of X if and only if y is a YES-instance of Y. If X reduces to Y in polynomial time, we write $X \leq_p Y$.

The reduction algorithm $\mathcal{R}$ takes in as input one string (input to problem X) and outputs another string (input to problem Y). Let us see an example right away.

Example 6. The Subset Sum decision problem takes in as input $\langle a_1, \ldots, a_n, T\rangle$ where all the parameters are integers. In a YES-instance, there exists a subset I such that $\sum_{i\in I} a_i = T$; in a NO-instance, there is no such subset. We claim that Subset Sum $\leq_p$ Knapsack. Given an input $\langle a_1, \ldots, a_n, T\rangle$, the reduction outputs $\langle (a_1, a_1), \ldots, (a_n, a_n), T, T\rangle$, i.e., a list of n items with the i-th item having weight and value equal to a_i, the weight limit equal to T, and the value target equal to T. The reduction is clearly in polynomial time as it just duplicates the parameters in the Subset Sum instance.

If the Subset Sum instance is a YES-instance, meaning that there exists I such that $\sum_{i\in I} a_i = T$, then I is also a valid knapsack solution for the reduction's output having total weight at most T and total value at least T. Thus, the reduction carries a YES-instance of Subset Sum to a YES-instance of Knapsack.

We also need to show that the reduction carries a NO-instance of Subset Sum to a NO-instance of Knapsack. We argue by contradiction. Suppose the output of the reduction $\langle (a_1, a_1), \ldots, (a_n, a_n), T, T\rangle$ is a YES-instance, meaning there is a knapsack solution I satisfying $\sum_{i\in I} a_i \leq T$ and $\sum_{i\in I} a_i \geq T$. But this means $\sum_{i\in I} a_i = T$, and so, the Subset Sum instance must be a YES-instance.

This example illustrates the steps involved in showing the validity of a reduction between decision problems. You need to **explicitly argue** each of the following aspects to show $X \leq_p Y$:

(i) The reduction runs in time polynomial in the size of the instance for X.

(ii) If the input to the reduction is a YES-instance for X, then its output is a YES-instance for Y.

(iii) If the output of the reduction is a YES-instance for Y, then its input is a YES-instance for X.

The notion of reductions is of course hugely important in designing algorithms. Often there is some library routine which solves a general problem, and you can solve your problem by reducing your problem to the more general problem. This is such a commonplace idea in algorithm design that we use it implicitly all the time without mention.

The way reductions come into play in these lectures is in the opposite sense, i.e., in showing that one problem is as hard as another. First, we make the following basic observation.

Lemma 10.3.2. *Let X and Y be two decision problems. If $X \leq_p Y$ and Y is in P, then X is also in P.*

Proof. Let $\mathcal{A}_Y$ be an algorithm solving problem Y that runs in time $O(n^k)$ time where n is the size of the Y-instance. Let $\mathcal{R}$ be the reduction from X to Y that runs in time $O(n^c)$ time where n is the size of the X instance. Now, note that the output of $\mathcal{R}$ must be of size $O(n^c)$ where n is the size of the X instance, since otherwise, the running time of $\mathcal{R}$ would be $\omega(n^c)$ just to write down its output.

Let $\mathcal{A}_X$ be the algorithm that given an instance x of X, outputs $\mathcal{A}_Y(\mathcal{R}(x))$. It is correct because of the correctness of $\mathcal{A}_Y$ and $\mathcal{R}$. Its runtime is $O(n^c + (n^c)^k) = O(n^{ck})$ where we are measuring the runtime of $\mathcal{A}_Y$ on inputs of size $O(n^c)$. The runtime is polynomial if c and k are constants. □

Corollary 10.3.3. *Let X and Y be two decision problems. If $X \leq_p Y$ and X is not in P, then Y is also not in P.*

Proof. Contrapositive of Lemma 10.3.2. □

These observations justify the $\leq_p$ notation: $X \leq_p Y$ means that X is "easier" than Y. Here, I am using "easier" in a very loose sense to mean that if Y can be solved in polynomial time, so can X. The optimal algorithm for X may run slower than the algorithm for Y, but it cannot be that the first runs in exponential time and the second runs in polynomial time.

General Problems

For general search problems (where the output is not a binary YES or NO answer), the definition of a reduction also involves converting the solution of the target problem to a solution of the source problem. There are two ways to do so. Let X and Y denote two problems.

- The reduction consists of two algorithms $\mathcal{R}_1, \mathcal{R}_2$ where $\mathcal{R}_1$ converts instances of X to instances of Y and $\mathcal{R}_2$ converts solutions of Y to solutions of X. The requirement is that both algorithms for the reduction must run in polynomial time in their inputs and that if $\mathcal{A}_Y$ is an algorithm to solve Y, then $\mathcal{R}_2(\mathcal{A}_Y(\mathcal{R}_1(x)))$ must be a solution to problem X with input x. This notion of a reduction is called a *many-one* or *Karp-Levin reduction.*

- A more general type of reduction is one which can invoke an algorithm $\mathcal{A}_Y$ for Y as a subroutine a polynomial number of times in order to solve X. (In contrast, in many-one reductions, the algorithm for Y can only be called once.) This notion of reduction is called a *Cook reduction.*

In the lecture, we only defined many-one reductions informally. You are not required to know the names of these two types of reductions or their distinction.

10.4 NP-Completeness

Please see Sections 34.2 and 34.3 of CLRS.

10.5 Some Reductions between NP-Complete Problems

- The reduction from 3-SAT to Independent Set is described in Section 12.7 of Jeff Erickson's book, "Algorithms".[4]
- The reduction from Independent Set to Vertex Cover is in Section 12.9 of "Algorithms".
- The reduction from Vertex Cover to Set Cover is not explicitly in CLRS or "Algorithms" but a good writeup is here.
- The reduction from 3-SAT to Directed Hamiltonian Cycle is in Section 12.11 of "Algorithms".
- The reduction from 3-SAT to Subset Sum is in Section 34.5.5 of CLRS.

[4] http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf