COSC 240: Lecture #23: Complexity Theory Basics

Introduction

The goal for these verbose lecture notes is to carefully build up to the formal definitions of the P and NP complexity classes, allowing us to hit the ground running after the holiday break by putting these definitions to work proving fundamental results about the difficulty of specific problems.

The below notes begin by reviewing the formal definitions of *problems* and *solving problems*. We covered these on Monday, but I am reproducing them here for easy reference, as the details in these definitions matter. We will then introduce the formal definition for the complexity class P. This complexity class is intuitive and easy to define: it contains every decision problem that can be solved by an algorithm in a number of steps that is polynomial in the input size.

Finally, we will introduce the formal definition for the complexity class NP. This definition is more subtle and much less intuitive, so we will take our time to try to make it clear. It might not be obvious at first how we will leverage the definition the NP complexity class to prove useful things about problems. Do not worry: our strategy will become clear next week, when we will deploy a clever set of theoretical tools to put this seemingly strange definition to work toward the useful goal of proving that certain concrete problems almost certainly cannot be solved *fast* (i.e., in polynomial time).

Problem Definitions

We begin by reminding ourselves of our formal definitions solving problems with algorithms. As established on Monday, we express these concepts in terms of formal language theory (e.g., alphabets, strings, languages, and so on).

Recall, there are two types of problems that an algorithm can attempt to solve: *optimization* and *decision*. Optimization problems are what we are used to studying. The algorithm is given an input and eventually returns a corresponding output. Decision problems, by contrast, are more constrained: the algorithm is given an input and the algorithm has only two options: it can *accept* (i.e. output 1) or *reject* (i.e., output 0).

In both computability theory (what can and cannot be solved by algorithms) and complexity theory (what can and cannot be solved efficiently by algorithms) we focus on decision problems as they can be expressed with cleaner mathematical representations that support more powerful theory. As argued on Monday, we are not losing much by this restriction. For one thing, we are focusing primarily on proving certain problems are hard, so by restricting our attention to easier versions of the problems makes our hardness results even stronger. Second, decision problems are not necessarily even easier in most cases. A consistent result in theoretical computer science is that the decision version of a problem is often as hard as the optimization version. That is, if you can solve the decision version efficiently, than you can use that solution as a subroutine and solve the optimization version with only a little extra work.[1]

Here are the relevant formal definitions for decision problems:

- From the perspective of language theory: a decision problem specifies which strings from Σ^* should be accepted (where Σ is the relevant input alphabet, and Σ^* is the language containing all strings made up of symbols from Σ).

[1] Consider, for example, the standard optimization problem of finding the largest size clique in a graph G provided as input. A well-studied decision version of this problem takes as input a graph G and size k, and accepts the input if and only if G contains a clique of size k. The decision version seems easier because you do not have to find the largest clique but simply decide whether or not any clique of a given size exists in the graph. It turns out, however, that once you can solve the decision problem, it is pretty easy to use this solution as a subroutine to efficiently find the largest clique in the graph. That is, the hard part about seeking cliques is already present in the decision version.

- Therefore: a decision problem can be formalized as the set of input instances to accept.

  e.g.,

  $$PATH = \{\langle G, u, v, k\rangle : G = (V, E) \text{ is an undirected graph with a path from } u \text{ to } v \text{ of length } k.\}$$

  (Recall: *the $\langle O\rangle$ notion denotes a reasonable encoding of the mathematical object(s) O into a string.*)

- We call this set of input instances that should be accepted a language.
- An algorithm A accepts string x, if given input x algorithm A accepts x.
- An algorithm A rejects string x, if given input x algorithm A rejects x.
- The language accepted by A is the set of every string that A accepts. (We also say A accepts this language.)
- An algorithm A decides language L if A accepts every string in L and rejects every string not in L.
- There are two ways algorithm A can "solve" a decision problem: it can either accept or decide the corresponding language. To decide a language is stronger as it guarantees that the algorithm will make a final decision on every input. If an algorithm only accepts a language, it is possible that it might loop forever on some input strings that are not in the language.

The P Complexity Class

A major reason we like using decision problems is because they can be formally defined as languages, which in turn are just sets of strings. These mathematical objects are flexible and easy to work with and extend.

For example, if a language is a set of strings, than we define a *complexity class* to be a set of languages. Typically, a complexity class will include all languages that all share some property about how hard they are to solve with algorithms (hence the use of the term "complexity" in the name). Let us define our first useful complexity class: (in the following, we assume our input alphabet $\Sigma = \{0, 1\}$):

- The complexity class P consists of every decision problem that can be solved by some algorithm in time that is polynomial in the input size. That is, for each $L \in P$, there exists an algorithm A_L that decides L. Furthermore, there exists a constant $k \geq 0$, such that for every input w, A_L accepts or rejects w in $O(n^k)$ time, where $n = |w|$.
- Formally: $P = \{L \subseteq \{0, 1\}^* \mid \text{there exists an algorithm } A_L \text{ that decides } L \text{ in polynomial time.}\}$.

Theoreticians think about P as the set of all decision problems that can solved *efficiently* by algorithms. You might, at first, balk at this claim. For a language L to be in P all that is required is that some algorithm A_L solves it in $O(n^k)$ time for some k. This k, however, might be really large. If the fastest possible solution to a problem, for example, requires $\Theta(n^{100})$ steps, how could we call that "efficient"?

Theoreticians do not really worry much about such details. Experience as shown that for the most part, once you can solve an problem in polynomial time for *any* polynomial, it is usually possible to improve that polynomial down to something more tractable. That work of improving n^{100} to something like n^2, however, is considered to lowly for the exalted theoretician who will often pass along such matters to an "algorithms person" to tackle.

Algorithms people, by contrast, feel like the theory work is the easy part—just math tricks with sets—and all the *real* hard thinking goes into finding super efficient solutions that people can actually use. This argument is old. But both groups do agree, however, that once a problem is shown in P, than it can safely be considered part of the list of things (probably) solvable efficiently by computers.

The NP Complexity Class

We now introduce the NP complexity class. Next week, we will use this class to start proving that certain specific problems almost certainly cannot be solved *efficiently*—a claim we can now make more formally: we will prove that certain problems almost certainly are not in P. Our goal here is to just get its definition right.

Let us begin by clearing up a common misconception. Because NP is used to help prove things are not in P, people sometimes incorrectly assume that NP stands for *not polynomial* and that it is the set of all problems not in P. Neither of these two assumptions are true. NP actually stands for *nondeterministic polynomial*[2] and it contains every problem in P and some problems not in P (probably). There are also many problems that are neither in P nor NP.

Verification Algorithms. Before we can give a formal definition for NP we must first define the useful concept of verification algorithms (in the following, we assume our input alphabet $\Sigma = \{0, 1\}$):

- A verification algorithm A receives two strings x and y as input. String x is called the input string and string y the certificate.
- The language L verified by a verification algorithm A is the set of inputs x such that there exists some y where A accepts input (x, y). Formally:

$$L = \{x \in \{0,1\}^* \mid \text{there exists } y \in \{0,1\}^* \text{ such that } A(x, y) = 1.\}$$

- This definition is a little bit confusing. What it is saying, in essence, is that the language verified by such an algorithm is the set of every string such that there exists *some* certificate that could convince the verifier to accept the input. These definitions make a lot more sense when you approach them from the language perspective. That is, you first fix a language and *then* ask: is there a verification algorithm that verifies this language?
- For example, consider the PATH language also described earlier in these notes:

$$PATH = \{\langle G, u, v, k\rangle : G = (V, E) \text{ is an undirected graph with a path from } u \text{ to } v \text{ of length } k.\}$$

- Can we come up with a verification algorithm that verifies this language? We can. Consider the verification algorithm A that does the following with input (x, y).
  - First, it checks that x is in the right format to potentially be in the language PATH; i.e., that $x = \langle G, u, v, k\rangle$ for some graph G, nodes u, v in G and length $k > 0$. If it is not in the right format, it rejects (x, y).
  - Second, it checks that the certificate y encodes a sequence of k nodes from G. If not, it rejects (x, y).
  - Finally, it studies the sequence of nodes in y to see they describe a path in G. If they do, it accepts (x, y), otherwise it rejects (x, y).
- In other words, this verification algorithm will accept an input x as belonging to PATH only if it is accompanied with a certificate that shows that a proper length path exists in the graph.

[2]The notion of nondeterminism is beyond the scope of this class. In a formal theory course, however, you would learn where this name comes from. In this class, we use an alternative definition of the class NP that requires less formal background.

- It is easy to verify that the language L verified by A is exactly PATH. (If x is not in PATH, then there is no y that could convince A to accept (x, y), while if x is in PATH, then by definition there exists a y that causes A to accept it.)
- When dealing with verification algorithms we care about how fast they conduct the verification. In more detail, we say a verification algorithm is a polynomial time verification algorithm if there exists a constant $k \geq 0$, such that for every input (x, y), it decides to *accept* or *reject* in $O(|x|^k)$ time.

The NP Complexity Class. We can now use our definition of verification algorithms to formally define the NP complexity class (as before, we will assume for now the input alphabet $\Sigma = \{0, 1\}$):

$$\text{NP} = \{L \subseteq \{0,1\}^* \mid \text{there exists a polynomial time verification algorithm } A \text{ that verifies } L\}.$$

In other words, a language L is in NP if for every $w \in L$, there exists some sort of evidence you could provide me that could convince me that w is in fact in L (without having to do too much work). It is, in other words, the set of efficiently verifiable languages.

It is not hard to show that everything in P is also in NP (formally, P $\subseteq$ NP). Most theoreticians suspect that there are also things in NP that are not in P (formally, P $\subset$ NP). Proving this conjecture is considered very hard. In fact, there is a million dollar prize for the first person to prove definitively that either P $=$ NP or P $\subset$ NP. But for our purposes, we will go forward assuming the latter is true.

In other words, we suspect that there are languages where it is hard to solve the problem from scratch, but once you have already solved it, it is easy to convince me that you have a solution. We will use these assumptions next week to start exploring concrete problems that likely fall into this category...