CMSC 451: Lecture 18
NP-Completeness: General Definitions
Tuesday, Nov 14, 2017

Reading: Chapter 8 in DPV, Chapter 8 in KT. (Some of today's material is not covered in either text.)

Efficiency and Polynomial Time: Up to this point of the semester we have been building up your "toolkit" for solving algorithmic problems efficiently. Hopefully when presented with a computational problem, you now have a clearer idea the sorts of techniques that could be used to solve the problem efficiently (such as divide-and-conquer, DFS, greedy, dynamic programming, network flow).

What do we mean when we say "efficient"? If n is small, a running time of 2^n may be just fine, but when n is huge, even n^2 may be unacceptably slow. Algorithm designers observed long ago that there are two very general classes of combinatorial problems:

- those involving *brute-force search* of all feasible solutions, whose worst-case running time is an *exponential function* of the input size,
- those that are based on a *systematic solution*, whose worst-case running time is a *polynomial function* of the input size.

An algorithm is said to run in *polynomial time* if its worst-case running time is $O(n^c)$, where c is a nonnegative constant. (Note that running times like $O(n \log n)$ are polynomial time, since $n \log n = O(n^2)$.) By *exponential time* we mean any function that is at least $\Omega(c^n)$ for a constant $c > 1$. Henceforth, we will use the terms "efficient" and "easy" to mean solvable by an algorithm whose worst-case running time is polynomial in the input size. (irrespective of whether the polynomial is n or n^{1000}).

While the distinction between worst-case polynomial time and worst-case exponential time is quite crude, it has a number of advantages. For example, the composition of any two polynomials is a polynomial. (That is, if $f(n)$ and $g(n)$ are both polynomials, then so is $f(g(n))$.) This means that, if a program makes a polynomial number of calls to a function that runs in polynomial time, then the overall running time is a polynomial.

The Emergence of Hard Problems: Near the end of the 60's, although there was great success in finding efficient solutions to many combinatorial problems, there was also a growing list of problems which were "hard" in the sense that no known efficient algorithmic solutions existed for these problems.

A remarkable discovery was made about this time. Many of these believed hard problems turned out to be equivalent, in the sense that if you could solve *any one* of them in polynomial time, then you could solve *all* of them in polynomial time. Often these hard problems involved slight generalizations to problems that are solvable in polynomial time. A list of some of these problems is shown in Table 1.

The mathematical theory, which was developed by Richard Karp and Stephen Cook, gave rise to the notions of P, NP, and NP-completeness. Since then, thousands of problems were

Table 1: Computationally hard problems and their (easy) counterparts.

Hard problems (NP-complete)	**Easy problems (in P)**
3SAT	2SAT
Traveling Salesman Problem (TSP)	Minimum Spanning Tree (MST)
Longest (Simple) Path	Shortest Path
3D Matching	Bipartite Matching
Knapsack	Unary Knapsack
Independent Set in Graphs	Independent Set in Trees
Integer Linear Programming	Linear Programming
Hamiltonian Cycle	Eulerian Cycle
Balanced Cut	Minimum Cut

identified as being in this equivalence class. It is widely believed that none of them can be solved in polynomial time, but there is no proof of this fact. This has given rise to arguably the biggest open problems in computer science:

$$\text{P} = \text{NP?}$$

While we will not be able to provide an answer to this question, we will investigate this concept in the next few lectures.

Note that represents a radical departure from what we have been doing so far this semester. The goal is no longer to prove that a problem *can* be solved efficiently by presenting an algorithm for it. Instead we will be trying to show that a problem *cannot* be solved efficiently. The question is how to do this?

Reasonable Input Encodings: When trying to show the impossibility of achieving a task efficiently, it is important to define terms precisely. Otherwise, we might be beaten by clever cheats. We will treat the input to our problems as a string over some alphabet that has a constant number, but at least two, characters (e.g., a binary bit string or a Unicode encoding). If you think about it for just a moment, every data structure that we have seen this semester can be *serialized* into such a string, without increasing its size significantly.

How are inputs to be encoded? Observe that if you encode an integer in a very inefficient manner, for example, using *unary notation* (so that 8 is represented as 11111111), rather than an efficient encoding (say in binary or decimal[1]), the length of the string increases by exponentially. Why should we care? Observe that if the input size grows exponentially, then an algorithm that ran in exponential time for the short input size may now run in linear time for the long input size. We consider this a cheat because we haven't devised a faster algorithm, we have just made our measuring yardstick much much longer.

All the representations we have seen this semester (e.g., sets as lists, graphs as adjacency lists or adjacency matrices, etc.) are considered to be reasonable. To determine whether some new

[1]The exact choice of the numeric base is not important so long as it is as least 2, since all base representations can be converted to each other with only a constant factor change in the length.

representation is reasonable, it should be as concise as possible (in the worst case) and/or it should be possible to convert from an existing reasonable representation to this new form in polynomial time.

Decision Problem: Many of the problems that we have discussed involve *optimization* of one form or another: find the shortest path, find the minimum cost spanning tree, find the maximum flow. For rather technical reasons, most NP-complete problems that we will discuss will be phrased as decision problems.

A problem is called a *decision problem* if its output is a simple "yes" or "no" (or you may think of this as True/False, 0/1, accept/reject). For example, the minimum spanning tree decision problem might be: "Given a weighted graph G and an integer z, does G have a spanning tree whose weight is at most z?"

This may seem like a less interesting formulation of the problem. It does not ask for the weight of the minimum spanning tree, and it does not even ask for the edges of the spanning tree that achieves this weight. However, our job will be to show that certain problems *cannot* be solved efficiently. If we show that the simple decision problem cannot be solved efficiently, then certainly the more general optimization problem certainly cannot be solved efficiently either. (In fact, if you can solve a decision problem efficiently, it is almost always possible to construct an efficient solution to the optimization problem, but this is a technicality that we won't worry about now.)

Language Recognition: Observe that a decision problem can also be thought of as a language recognition problem. Define a *language* to be a set (finite or infinite) of strings. To express a computational problem as a language-recognition problem, we first should be able to express its input as a string. Let us define serialize(X) to be a function that maps any cominatorial structure X into a string. (For example, serializing a graph would involve outputting a string that encodes its vertices, edges, and any additional information such as edge weights.)

Using this, we could define a language MST encoding the Minimum Spanning Tree problem as a language (that is, a collection of strings):

$$\text{MST} = \{\text{serialize}(G, z) \mid G \text{ has a minimum spanning tree of weight at most } z\}.$$

What does it mean to solve the decision problem? When presented with a specific input string $x = \text{serialize}(G, z)$, the algorithm would answer "yes" if $x \in \text{MST}$, that is, if G has a spanning tree of weight at most z, and "no" otherwise. In the first case we say that the algorithm *accepts* the input and otherwise it *rejects* the input. Thus, decision problems are equivalent to language-recognition problems.

Given an input x, how would we determine whether $x \in \text{MST}$? First, we would decode x as G and z. We would then feed these into any efficient minimum spanning tree algorithm (Kruskal's, say). If the final cost of the spanning tree is at most z, we accept x and otherwise we reject it.

The Class P: We now present an important definition:

Definition: P is the set of all languages (i.e., decision problems) for which membership can be determined in (worst case) polynomial time.

Intuitively, P corresponds to the set of all decisions problems that can be solved efficiently, that is, in polynomial time. Note P is not a language, rather, it is a set of languages. A set of languages that is defined in terms of how hard it is to determine membership is called a *complexity class*. (Therefore, P is a complexity class.)

Since Kruskal's algorithm runs in polynomial time, we have MST $\in$ P. We could define equivalent languages for all of the other optimization problems we have seen this year (e.g., shortest paths, max flow, min cut).

A Harder Example: To show that not all languages are (obviously) in P, consider the following:

$$\text{HC} = \{\text{serialize}(G) \mid G \text{ has a simple cycle that visits every vertex of } G\}.$$

Such a cycle is called a *Hamiltonian cycle* and the decision problem is the *Hamiltonian Cycle Problem*.

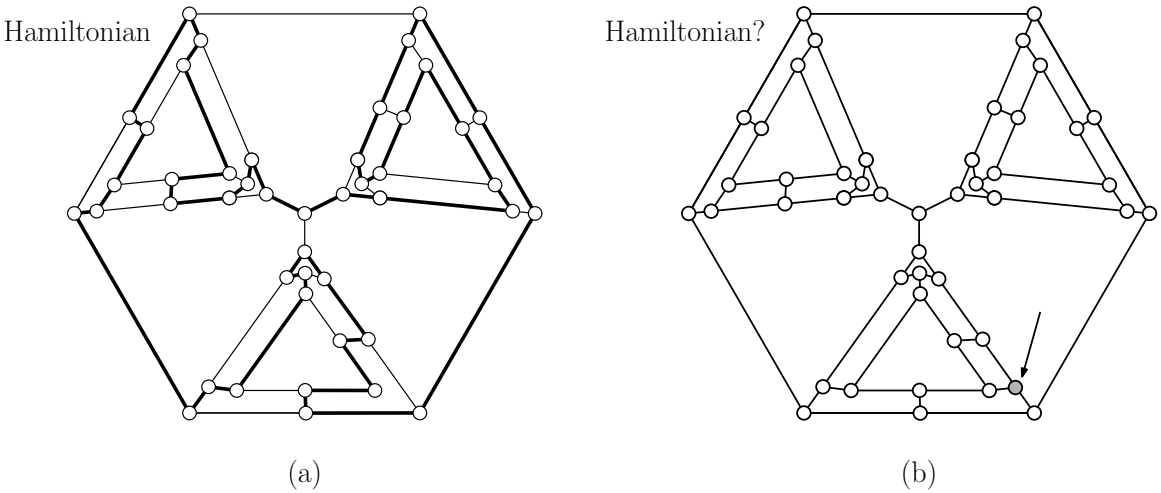

Fig. 1: The Hamiltonian cycle (HC) problem.

In Fig. 1(a) we show an example of a Hamiltonian cycle in a graph. If you think that the problem is easy to solve, try to solve the problem on the graph shown in Fig. 1(b), which has one additional vertex and one additional edge. Either find a Hamiltonian cycle in this graph or show than none exists. To make this even harder, imagine a million-vertex graph with many slight variations of this pattern.

Is HC $\in$ P? No one knows the answer for sure, but it is conjectured that it is not. (In fact, we will show that later that HC is NP-complete.)

In what follows, we will be introducing a number of classes. We will jump back and forth between the terms "language" and "decision problems", but for our purposes they mean the same things. Before giving all the technical definitions, let us say a bit about what the general classes look like at an intuitive level.

Polynomial-Time Verification and Certificates: In order to define NP-completeness, we need to first define NP. Unfortunately, providing a rigorous definition of NP will involve a presentation of the notion of *nondeterministic* models of computation, and will take us away from

our main focus. (Formally, NP stands for *nondeterministic polynomial time*.) Instead, we will present a very simple, "hand-wavy" definition, which will suffice for our purposes.

To do so, it is important to first introduce the notion of a verification algorithm. Many language recognition problems that may be *hard to solve*, but they have the property that they are *easy to verify* that a string is in the language. Recall the Hamiltonian cycle problem defined above. As we saw, there is no obviously efficient way to find a Hamiltonian cycle in a graph. However, suppose that a graph did have a Hamiltonian cycle and someone wanted to convince us of its existence. This person would simply tell us the vertices in the order that they appear along the cycle. It would be a very easy matter for us to inspect the graph and check that this is indeed a legal cycle that it visits all the vertices exactly once. Thus, even though we know of no efficient way to *solve* the Hamiltonian cycle problem, there is a very efficient way to *verify* that a given graph has one. (You might ask, but what if the graph did not have one? Don't worry. A verification process is not required to do anything if the input is not in the language.)

The given cycle in the above example is called a *certificate*. A certificate is a piece of information which allows us to verify that a given string is in a language in polynomial time.

More formally, given a language L, and given $x \in L$, a *verification algorithm* is an algorithm which, given x and a string y called the *certificate*, can verify that x is in the language L using this certificate as help. If x is not in L then there is nothing to verify. If there exists a verification algorithm that runs in polynomial time, we say that L can be *verified in polynomial time*.

Note that not all languages have the property that they are easy to verify. For example, consider the following languages:

$$\begin{aligned} \text{UHC} &= \{G \mid G \text{ has a unique Hamiltonian cycle}\} \\ \overline{\text{HC}} &= \{G \mid G \text{ has no Hamiltonian cycle}\}. \end{aligned}$$

There is no known polynomial time verification algorithm for either of these. For example, suppose that a graph G is in the language UHC. What information would someone give us that would allow us to verify that G is indeed in the language? They could certainly show us one Hamiltonian cycle, but it is unclear that they could provide us with any easily verifiable piece of information that would demonstrate that this is the only one.

The class NP: We can now define the complexity class NP.

Definition: NP is the set of all languages that can be verified in polynomial time.

Observe that if we can solve a problem efficiently without a certificate, we can certainly solve given the additional help of a certificate. Therefore, $\text{P} \subseteq \text{NP}$. However, it is not known whether $\text{P} = \text{NP}$. It seems unreasonable to think that this should be so. In other words, just being able to verify that you have a correct solution does not help you in finding the actual solution very much. Most experts believe that $\text{P} \neq \text{NP}$, but no one has a proof of this. Next time we will define the notions of NP-hard and NP-complete.

There is one last ingredient that will be needed before defining NP-completeness, namely the notion of a *polynomial time reduction*. We will discuss that next time.