CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 23

April 19, 2022

Instructor: Sepehr Assadi

1 Introduction to (Complexity) Classes P and NP

Our focus in this course so far has been to design "efficient" algorithms for various problems. For the remainder of the course however, we examine a complementary question: What are some "natural" problems that do *not* admit such efficient algorithms? This requires to delve into the notions of *NP-completeness* and *NP-hardness*. To do so, we start this lecture by a series of definitions.

Polynomial-Time Algorithms

A minimal (and widely accepted) requirement for an algorithm to be considered "efficient" is that it should be a *polynomial-time* (or *poly-time* for short) algorithm: An algorithm is considered poly-time if its worst case runtime is $O(N^k)$ for some constant $k \geq 0$ where N denotes the *number of bits* used to represent the input (namely, size of the input).

Almost (but not quite) all algorithms we studied in this course are poly-time algorithms:

- All the sorting algorithms we studied including insertion sort, selection sort, quick sort, and merge sort, take $O(n^2)$ time on inputs of n numbers (and hence $N = \Omega(n)$), and are hence poly-time. The same holds for binary search and linear search algorithms[1].
- Most dynamic programming algorithms we studied (with the exception of Fibonacci numbers and Knapsack which we will discuss below) were poly-time algorithms: for instance, in the LIS problem, we are given a string of length n (and hence $N = \Omega(n)$ bits) and the runtime of the algorithm was $O(n^2)$ which is polynomial in the input size. The same holds for greedy algorithms that we studied.
- The graph algorithms that we studied primarily such as connectivity, minimum spanning tree, shortest paths, longest path in a DAG, and bipartite matching were again all poly-time algorithms[2].

Remark: Recall that the runtime of our algorithm for the Fibonacci number problem (i.e., outputting the n-th Fibonacci number) was $O(n)$ (after doing dynamic programming) where n *itself* is given as input. In order to represent the input n, we only need to use $N = \lceil \log n \rceil$ bits and hence the *input size* to this problem was $O(\log n)$ while the runtime was *exponentially* larger, i.e., $O(n)$ (which is $2^{\Theta(N)}$). This means that our Fibonacci algorithm was *not* a poly-time algorithm. The same also holds for the Knapsack problem where our algorithm had runtime of $O(nW)$ while the input representation was $O(n \log W)$ bits and thus the runtime was not polynomial in $\log W$.

[1] Note that merge sort takes only $O(n \log n)$ which is faster than $O(n^2)$ and binary search takes $O(\log n)$ which is much faster than $O(n)$ of linear search. However, considering O-notation provides an *upper bound*, we could simply consider the runtime of the former to be $O(n^2)$ and the latter to be $O(n)$ for the purpose of the discussion above.

[2] We note that, the Ford-Fulkerson algorithm discussed for maximum flow itself is again *not* a poly-time algorithm (due to its dependence on F; see the remark below) but there are many poly-time algorithms known for the maximum flow problem.

Decision Problems

A decision problem is any problem whose answer is either *True* or *False* (or alternatively, *Yes/No*, or *1/0*). Many of the problems we studied in this course were decision problems:

- Is element e belong to the array A? (binary-search or linear-search algorithm)
- Is there a way to pick a subset of items in a knapsack to make the knapsack completely full? (dynamic programming algorithm)
- Is undirected graph G connected? (DFS or BFS algorithms)
- Is directed graph G a DAG? (topological sorting algorithm)
- ...

Moreover, most problems that are *not* decision problems also have similar variants that are decision problems (by adding an extra parameter to the problem statement – in the examples below, parameter K):

- Is the weight of MST of a given graph G larger (or smaller) than a given number K?
- Is the shortest path from a vertex s to a vertex t in a given graph G larger (or smaller) than a given number K?
- ...

For the purpose of this lecture, we mostly focus on decision problems although some of the definitions (but definitely not all of them) provided can be extended to non-decision problems as well.

Polynomial-Time Verifiers

For the entirety of this course (and most likely in all the courses you studied so far), we have focused on algorithms that *solve* a problem Π. This means the following:

- An algorithm for a decision problem Π, gets as input the input x to problem Π, and outputs the value of $\Pi(x)$, i.e., the answer to the problem on the input x.

For instance:

- Problem Π can be the problem of whether the given graph G is connected or not. Here, $x = G$ corresponds to any input graph and $\Pi(x) = 1$ means the graph corresponding to x is connected and $\Pi(x) = 0$ means the graph is not connected. An algorithm for the connectivity problem (say DFS or BFS) takes as input the graph $x = G$ and output $\Pi(x)$, i.e., whether the graph x is connected or not.

However there is a *seemingly* simpler task than solving a problem and that is *verifying* (a solution to) a problem. Roughly speaking, a verifier for a problem Π, gets as input the input x to Π plus a *proof* y that $\Pi(x) = 1$ and only needs to verify *with the help of this proof* whether $\Pi(x) = 1$ truly or not. Formally,

- A verifier for a decision problem Π, gets as input the input x to problem Π and a proof[3] y that "tries" to prove $\Pi(x) = 1$: The verifier then should simply use x and y to determine whether $\Pi(x) = 1$ or not. In other words, if $\Pi(x) = 1$, there should always be a proof y that the verifier given x, y outputs $\Pi(x) = 1$ and if $\Pi(x) = 0$, no proof y should results in the verifier to output $\Pi(x) = 1$ given x, y.

[3]Think of the proof being provided by an "oracle" that knows everything and can solve any problem – we are not worried at all how the proof itself is being provided but only that such a proof *do exists*.

Considering this definition can be somewhat cumbersome (although the notion itself is quite intuitive), let us examine a couple of verifiers (let us emphasize that verifiers are also still algorithms; however, instead of solving the problem, they are simply verifying whether a given answer to the problem is correct or not).

- **Graph connectivity problem:** Given an undirected graph $G = (V, E)$, is G connected or not?
  - A verifier for graph connectivity: Any connected graph has a spanning tree and vice versa. So we can use any of the spanning trees of G (corresponding to x) as a proof (corresponding to y). We can design a verifier that gets the graph G as input and a supposed spanning tree T of G as a proof; the verifier then simply needs to check whether T is indeed a spanning tree of G or not (by (1) making sure every vertex of G belongs to T, (2) every edge of T belongs to G, and (3) T is a tree, namely, T has $n - 1$ edges). It is easy to design an algorithm for this and prove its correctness (we omit the details).
- **Hamiltonian cycle problem:** Given an undirected graph $G = (V, E)$, is there any *Hamiltonian cycle* in G? A Hamiltonian cycle is any cycle that goes through every vertex. See Figure 1 below.

Figure 1: An example of a Hamiltonian cycle (red edge).

  - A verifier for Hamiltonian cycle: If the graph has a Hamiltonian cycle, we can simply use that as a proof. So the input to our verifier is the input graph G and a supposed Hamiltonian cycle C of G (here G is the input x and C is the proof y). The verifier then goes over the edges of C one by one to ensure that it is a cycle and that every vertex of G is visited in the cycle (so C is indeed a Hamiltonian cycle).
- **3-coloring problem:** Given an undirected graph $G = (V, E)$, is there a way to color vertices of G with only 3 colors so that no edge is *monochromatic*, i.e., has both its endpoints colored the same? (See Figure 2 below)

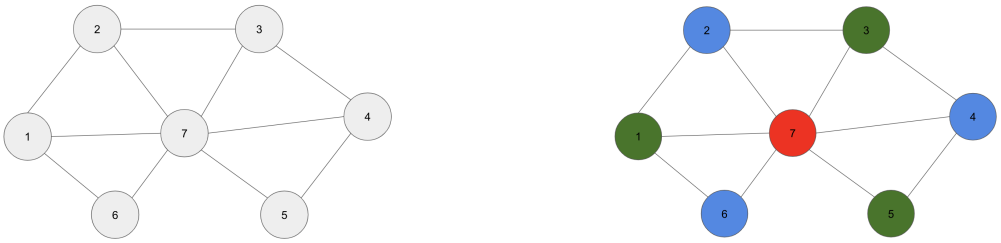

Figure 2: An example of a 3-colorable graph.

  - A verifier for 3-coloring problem: If the graph is 3-colorable, we can simply use any 3-coloring of the graph as a proof. So the input to our verifier is the input graph G and a supposed 3-coloring of G (here G is the input x and the 3-coloring is the proof y). The verifier then goes over the edges of G one by one to ensure that no edge is monochromatic.

Important Note: One good way to think of verifiers is as follows (this comes very handy when the goal is to *design* a verifier): Suppose your goal is to provide a solution to some problem Π on an input x, and suppose that someone approaches you and claims that $\Pi(x) = 1$, i.e., the answer to the problem Π on this input x is 1. Of course, there is a-priori no reason for you to believe this claim. What kind of a proof do you demand from this person that allows you to *verify* the claim that $\Pi(x) = 1$ indeed with the *help* of this proof? Note that the "burden" of providing the proof is *not* on you! You merely need to specify what type of a proof you need and the second person would provide that to you. In other words, you do not need to worry about *how* that person is finding this person, but only that such a proof exists *if and only if* the answer to Π on x is 1. What you also need is that you should be able to with the help of this proof y, and the input x, determine for yourself whether $\Pi(x) = 1$ or not.

Finally, a simple hint is that for the majority of the problems (but *not all* problems!), the best proof is the solution to the problem itself (e.g., if a graph has a Hamiltonian cycle, we can use this Hamiltonian cycle as a proof that the graph has a Hamiltonian cycle; similarly if a graph is 3-colorable we can use a 3-coloring of the graph as a proof).

Polynomial-time Verifiers. Similar to before, the minimal requirement for a verifier to be considered efficient is that its runtime should be polynomial in the length (bit representation) of the input x *and* the length of the proof y itself is also polynomial in x (after all, it shouldn't take "forever" for the verifier to just read the proof in the first place!).

Complexity Classes P and NP

We are now ready to define classes P and NP. Here, by *class*, we simply mean a set of problems.

- **Class P:** The set of all decision problems that admit a poly-time algorithms. In other words, these are problems that can be *solved* efficiently.
- **Class NP:** The set of all decision problems that admit a poly-time verifier. In other words, these are problems that can be *verified* efficiently. Intuitively, NP is the set of decision problems where we can verify a Yes answer (or True or 1) "efficiently" *assuming* we have the solution (or some other form of proofs) in front of us.

Basic observation: We have P $\subseteq$ NP, namely, any problem in P also belongs to NP. If we can solve a problem in polynomial time, we do not need a proof to check what is the solution to the problem: the verifier algorithm can simply run the poly-time algorithm to solve the problem and check for itself whether the answer to the problem is true or false.

It turns out that most of the decision problems we encounter in practice belong to the class NP. On the other hand, as stated earlier, we could only expect having an efficient algorithm for problems in P. The natural question is then:

Is P = NP or not?

I.e., is it the case that any problem that can be verified efficiently can also be solved efficiently?

This is perhaps the single most important open question in computer science. At this point, we are very far from proving or disproving this statement. However, it should be said that majority of researchers strongly believe that P $\neq$ NP. We discuss the reasons behind this belief and how we can use this when approaching problems in the next lecture.

2 NP-Hard and NP-Complete Problems

We have the following two definitions:

- **NP-Hard Problems:** We say that a problem Π is NP-hard if having a polynomial-time algorithm for Π implies that P = NP, namely, by giving a poly-time algorithm for Π, we obtain a poly-time algorithm for *all* problems in NP. Note that NP-hard problems are *not* necessarily decision problems.
- **NP-Complete Problems:** We say that a *decision* problem Π is NP-complete if it is NP-hard and it also belongs to the class NP. Intuitively, NP-complete problems are the "hardest" problems in NP: if we can solve any one of them in polynomial time, we can solve *every* NP problem in polynomial time. Alternatively, we can also think of NP-complete problems as "easiest" problems that are NP-hard.

So why do we care about NP-hard (or NP-complete) problems? The answer is two-fold:

- If you believe (or suspect, or think, or guess, or whatever) P = NP, then "all" you need to do is to pick your favorite NP-hard problem and design a polynomial time algorithm for that. There are tons and tons of very famous NP-hard and NP-complete problems (and literally thousands of no-so-famous ones) and numerous people have studied them in depth in the hope of obtaining a polynomial time algorithm for any of them (so far with no success)[4].
- On the other hand, if you believe (or suspect, yada yada) P $\neq$ NP, then by proving that a problem is NP-hard (or NP-complete), you have effectively convinced yourself (and tons of other people) that this problem does not have a polynomial time (or "efficient") algorithm[5].

So how do we prove a problem is NP-hard? By **reduction** to any other NP-hard problem!

Reductions for Proving NP-Hardness

Suppose you want to prove that problem A is NP-hard. Suppose you also already know that problem B is NP-hard. "All" you have to do now is to prove that *any* polynomial-time algorithm for problem A can be used to solve problem B in polynomial-time also. But this is basically reducing problem B to A that you have done multiple times in this course (and surely elsewhere). The only thing we need to ensure that the reduction takes polynomial time itself.

The plan for testing whether a problem A is NP-hard then is simply to see if we can do reduce *any* other NP-hard problem B to this problem (in polynomial time). Note that in this plan, it is completely irrelevant that we do *not* know a polynomial-time algorithm for A. All we need to do is to show that *if* there is ever a polynomial-time algorithm for A, then the same algorithm would also imply a polynomial-time algorithm for B. But then we are done, since, *by definition*, having a polynomial-time algorithm for B implies P = NP.

Wait! For the above plan to work, we also need to have at least one NP-hard problem to begin with. That original problem is *Circuit-SAT* problem (proven to be NP-hard separately by Cook and Levin).

Remark: Before moving on, let us briefly also state what it takes to prove that a problem A is NP-complete (instead of NP-hard). In that case, we need to (1) do a reduction to prove that A is NP-hard *and* (2) give a poly-time verifier to prove that A belongs to NP (recall the definition of NP-complete problems).

Circuit-SAT Problem: The "First" NP-Complete Problem

The Circuit-SAT (short form for (boolean) circuit satisfiability) problem is defined as follows.

Problem 1 (**Circuit-SAT Problem**)**.** We are given a boolean circuit C with (binary) AND, OR, (unary) NOT gates, and n input bits. For any $x \in \{0,1\}^n$, we use $C(x)$ to denote the value of circuit on x. The

[4]Perhaps, the strongest evidence that P $\neq$ NP is that despite tremendous effort, none of these problems have been solved in polynomial time (yet!). Although admittedly one could also argue that despite tremendous effort, no one has yet prove P $\neq$ NP either so maybe this is not such a strong evidence after all.

[5]Proving that a problem is NP-hard has this extra benefit that no one in their right mind would still expect you to give a polynomial-time algorithm for that problem even if that person sincerely believes P = NP.

Circuit-SAT problem then asks does there exists at least one boolean input x such that $C(x) = 1$ or not? In other words, is this circuit *ever* satisfiable (outputs one) or not?

Circuit-SAT is in NP. A simple verifier is as follows:

- Proof: If C is satisfiable then there should exists some x where $C(x) = 1$. We can use any such x as a proof.
- The verifiers then gets the input circuit C and the proof x that $C(x) = 1$. We can evaluate $C(x)$ in the circuit in polynomial time by computing the value of each gate from bottom-up (or in a top-down fashion using a DFS-type algorithm). Either way, given a value x, computing $C(x)$ can be done in polynomial time and once we computed $C(x)$, we can check whether $C(x) = 1$ or not.

As we designed a poly-time verifier for this problem, this means that Circuit-SAT is in NP.

Circuit-SAT is NP-Hard (NP-Complete). The is the so-called Cook-Levin theorem. Proving this result is beyond the scope of our course at this point and we shall assume its correctness for this course.