COMPSCI 330: Design and Analysis of Algorithms 11/18/2014

Lecture #23

Lecturer: Debmalya Panigrahi *Scribe: Nat Kell*

1 Introduction

For most (if not all) of the problems we have examined this semester, we have been able to define algorithms that solve these problems efficiently, i.e., in polynomial time (e.g., $O(n)$ time, $O(n^3)$ time, etc.). Unfortunately, problems that are known to be polynomial-time solvable are the exception and not rule—it turns out that for most problems the arise in both practice and theory, we do not know of any polynomial time algorithm that solves them. Furthermore, based on the current state of computer science research, there are no existing methods for showing efficient algorithm *cannot* exist (i.e., it is still possible that there are efficient algorithms for these problems that we have not discovered, although the general consensus/intuition of the computer science community is that this is likely not the case).

So suppose when you join the workforce after graduating, your boss gives you a problem and asks you to find/code an algorithm that solves it efficiently. After working for some time, you start to believe that this problem is too hard for a computer to solve in a reasonable amount of time. It is inaccurate to report back and claim: "It is impossible to solve this problem efficiently!" since even the brightest computer science researchers have been unable to show such a property. Ideally, you would like some method that would formally allow you to state that "I cannot solve this problem efficiently, but neither can the smartest people in the world!"

In this lecture, we will introduce the notion of *NP-hardness*, which gives us a formal method for making such a claim. At a high level, our goal will be to take a new problem and show it is at least as hard as some problem that is already known to be hard. By doing this, we show that even though we might not have an efficient algorithm for this new problem, such an algorithm would imply that there is an efficient algorithm for a hard problem that even the best researchers have been unable to crack.

2 Easy versus hard problems

In this section, we will begin by defining a suite of hard problems we have not examined before during the semester. Interestingly, many of these problems have seemingly similar variants that are easy to solve. We will mention these variants as we go in order to highlight how a small change in a problem definition can result in a large jump in complexity.

2.1 Satisfiability

The first hard problem we will examine is what is known as *Satisfiability* or SAT. As input, we are given a set of n boolean variables $X = \{x_1, x_2, \ldots, x_n\}$ (i.e., each variable can be set to either true or false). We are then given a *boolean formula* over these variables of the following form (noting that this is just a specific example where $X = \{x_1, x_2, x_3, x_4\}$):

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_2} \vee x_3 \vee x_4) \wedge (x_4 \vee \overline{x_1} \vee x_2) \tag{1}$$

In terms of notation, '$\vee$' and "$\wedge$" denote the conjunction and disjunction operations, respectively (which are analogous to the OR and AND operations for boolean variables in a programming language). $\overline{x_i}$ denotes the negation of the boolean variable x_i (analogous to $!x_i$). We deem sequence of conjunctions within a given set of parentheses (e.g., $(\overline{x_2} \vee x_3 \vee x_4)$) as a *clause*, and we call the unnegated/negated variables within the formula *literals*.

So more generally, as input we are given a boolean formula over the variables in X that the disjunction of m clauses, each of which is the conjunction of a sequence of boolean literals. Our goal then is to determine if there is way of setting the variables (i.e., assigning "true" or "false" to each x_i) such that the entire formula evaluates to "true". We call such an assignment a *satisfying assignment*. If there exist at least one assignment for a formula, we say it is *satisfiable*. For instance, one satisfying assignment for the above example is x_1 = true, x_2 = false, x_3 = true, and x_4 = true (and there are others, as well). A simple example of a formula that is not satisfiable is

$$(x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2}). \tag{2}$$

Observe that no matter how we set x_1 and x_2, at least one of the clauses will not be satisfied.

A more structured version of SAT is what is known as k-SAT, which is defined identically but with added requirement that each clause has exactly k literals. Therefore, the formulas in (1) and (2) are instances of 3-SAT and 2-SAT, respectively. Surprisingly, we can solve 2-SAT in $O(n+m)$, but for any $k \geq 3$, there is no known polynomial time solution. In other words, the jump from $k = 2$ to $k = 3$ makes the problem much harder, and in fact we will show that 3-SAT is at least as hard as generic SAT.

2.2 Traveling Salesman, Hamiltonian Cycles, Eulerian Cycles

One of the most famous hard problems is what called the *Traveling Salesman* problem, or TSP. Here, we are given edge-weighted graph G (we typically think of the vertices of the graph as representing cities and an edge weight of an edge (u, v) as specifying the distance between cities u and v). The goal is to find a cycle in the graph of minimum the graph that includes every vertex with minimum total weight (this models a salesman traveling to a set of cities and returning home; ideally he wants to minimize the total distance he has to travel).

Again, we will formally argue why TSP is hard later on, but to do this, let us first define a slightly simpler problem known as *Hamiltonian Cycle* or HAM. Now, we are given an unweighted graph G, and the goal is to determine whether there exists a cycle in the graph that visits every vertex exactly once. Even though HAM is somewhat simpler than TSP, HAM is still a hard problem.

As we mentioned a moment ago, the jump from 2-SAT to 3-SAT makes an easy problem into a hard one. A similar phenomenon occurs with an analogous problem to HAM known as *Eulerian Cycle*. For the Eulerian Cycle problem, we are again given a unweighted graph G, but now the objective is to determine if there exists a path in the graph that starts and ends at the same vertex, includes every *edge* in the graph, and does not repeat any edges. Eulerian Cycle is in fact polynomial-time solvable due to the following elegant characterization: *a graph has an eulerian cycle iff the degree of each vertex is even.* Therefore in order to determine if a graph is eulerian, we just need to iterate over each vertex v in V and check to see if $\deg v$ is even.

2.3 Matching

A problem we have seen earlier in the semester is that of *maximum matching*. We primarily focused on the bipartite setting, but the problem can be defined for general graphs as well: given an undirected graph

$G = (V,E)$, find the largest subset $E' \subseteq E$ such that $|E'| \leq k$ and there does not exist a vertex v such that $(v,u) \in E$ and $(v,w) \in E$ for some $w,u \in E$ (the latter is just enforcing that if a vertex is matched, then it is matched with a unique neighbor).

The above matching problem turns about to be solvable in polynomial time (even when the graph is not bipartite). However, we can make the problem hard by making it a *3D-matching problem*. That is, the problem is now defined for a graph where edges are now a subset of $V \times V \times V$ (instead of the typical $V \times V$; we typically refer to such a graph as a *hypergraph*). The problem is defined identically, except now we require that v belongs to at most one hyper-edge in E' (formally, there does not exist two non-identical hyper-edges $(v,u_1,u_2),(v,u_3,u_4) \in E'$). Making the jump from 2D-matching to 3D-matching makes the problem hard (this should seem somewhat similar to the jump from 2-SAT to 3-SAT).

3 NP-hardness

For the remainder of the lecture, we will introduce the concept of NP-hardness, which as mentioned earlier, gives us a means for arguing that a problem is hard. To do this, we will first introduce the problem classes P and NP.

3.1 The classes P and NP

There is an entire subfield of computer science called *complexity theory* dedicated to classifying problems based on different notions of hardness. The most of basic of these classes is that of P, which is the set of *decision problems* that can be solved in polynomial time. By decision problem, we mean that the problem just requires a "yes" or "no" as the output for the problem. Most of the problems we have seen this semester when formulated as decision problem are in P since we have defined polynomial-time algorithms for them (e.g., shortest path, minimum spanning tree, max flow, etc.). For example, the decision-problem version of the shortest path problem would be: given a graph G and a parameter k, does there exist a path from the source s to sink t that has length at most k (clearly, we can answer this in polynomial time since we can run Dijkstra's algorithm, and if the length of the shortest path is less than k, then we output "yes"; if the length of the shortest path is greater than k, then we output "no").

A broader class of decision problems that will be of interest to us is that of NP, which stands for *non-deterministic polynomial* [1] (note that it does *not* stand for non-polynomial). The set of problems in NP are those that can be *verified* in polynomial time. By verify, we mean that for any instance where the answer is "yes", it possible for someone to give us a "certificate" or "proof" that makes it possible for an algorithm to check in polynomial time that the instance is a yes instance (this "someone" is often referred to as an *oracle*). Clearly, any problem in P is also in NP because the certificate the oracle gives us can be empty, since to verify an instance we can simply use the polynomial time algorithm for problem to prove whether or not it is a yes instance.

All the hard problems we mentioned in the previous section are also in NP. For example, SAT is in NP because if there is a satisfying assignment to a formula, an oracle could specify the assignment itself (i,e., what each $x_i \in X$ is set to in the satisfying assignment). Then an algorithm could easy evaluate the formulate using this assignment to verify that it indeeds evaluate to true. Furthermore, it is easy to implement this evaluation so that it takes polynomial-time (the algorithm can just step through the formula and directly

[1] The term "nondeterministic polynomial" comes from an alternate but equivalent definition of the problem class NP (the languages in NP are those that can be accepted by a polynomial-time nondeterministic turing machine); however, for our purposes, just think of NP in terms of polynomial-time verification.

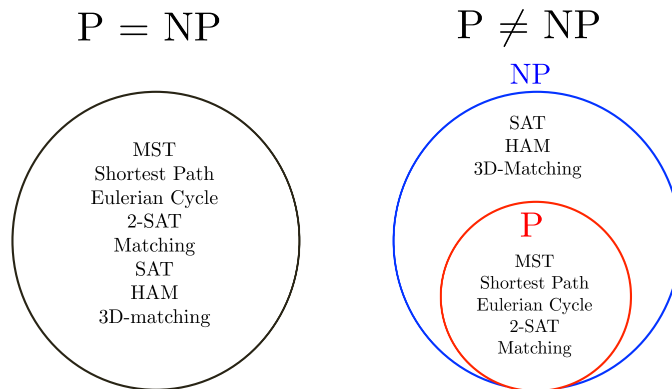

Figure 1: Possibilities for the relationship between P and NP.

evaluate each clause). A good exercise is to go through the other hard problems we saw (Hamiltonian Cycle, 3D-matching) and argue why these problems exist in NP.

The natural question to ask is: Are there problems that are in NP but are not in P? More explicitly, are there problems that can be verified in polynomial time but cannot be solved in polynomial time? Unfortunately, no one knows the answer to this question. In fact, a resolution to this question is one a greatest open questions in computer science and and even general mathematics (the Clay Institute would actually pay you one million dollars if you resolved the problem). A vast majority of computer scientist believe that NP $\neq$ P, but none of the existing tools make a formal resolution seem reachable in the near future.

Based on our current understanding, there are two possibilities for the relationship between P and NP (depicted in Figure 1). Either P = NP, and every problem we have discussed in this lecture (even the hard ones) is in fact solvable in polynomial time. The other (and more likely) possibility is that P$\neq$ NP, i.e. P is a strict subset of NP (recall that we argued that every problem in P is trivially in NP, as well). If this is the the case, then the hard problems in NP we have discussed lie strictly outside of P and do not have polynomial time algorithms.

As of now, it is unlikely that there are polynomial time algorithms for SAT, HAM, and 3D-matching, but given this uncertainty, how do we formally make a statement about their hardness?

3.2 Polynomial time reductions and NP-hard problems

To formalize the statement that "a problem B is at least as hard as hard as another problem A", we will use what is called a *polynomial time reduction* from A to B. The idea is that we design method for transforming a generic input instance for A into an instance for B. If the done correctly, the transformation is constructed in such a way so that if we assume we have an efficient solver for B, the structure of the transformed instance will allow us to map the solution we obtain from this B solver to a correct answer to the original instance for A. Once the reduction is established, it allows us to claim that if someone was to design an algorithm that solves problem B in polynomial time, then it would immediately imply a polynomial time algorithm for problem A (we usually denote this fact as $A \leq B$).

Figure 2 depicts the above recipe for constructing a reduction from decision problem A to decision problem B. We first take an input to problem A I_A and then transform I_A in some way to an instance to problem B I_B (in the figure, this is ALGO 1). We then solve the instance B using a solver for B (this is ALGO 2). When designing ALGO 2, we assume that we have an algorithm/oracle that can solve an instance

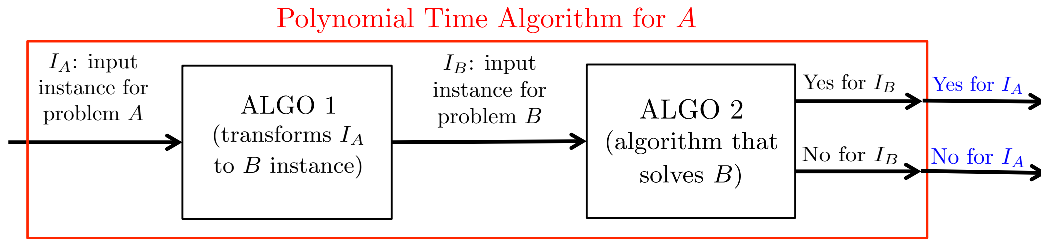

Figure 2: Showing problem A is polynomial time reducible to problem B.

of B in polynomial time. Usually ALGO 2 just entails querying the problem B oracle once and returning the same answer (either yes or no) as our answer to the instance to A (although, it is admissible to do more complicated constructions for ALGO 2 that involve multiple queries to the Problem B oracle). Regardless, if the reduction is constructed properly, then the final yes or no that is outputted from ALGO 2 will be the correct answer for the original input to problem A.

In summary, to define a reduction and argue its correctness, one needs to:

1. Specify polynomial time algorithms ALGO 1 and ALGO 2, i.e., how the input to A is transformed to a B instance and then what queries are made to the polynomial solver for B.

2. Argue that ALGO 2 outputs yes if and only if I_A is a yes instance for problem A.

We will now introduce another class of problems known as *NP-hard*. A problem B is said to be NP-hard if for all problem $A \in$ NP, A is polynomial time reducible to B. Informally, this class consists of all problems that are at least hard as *every* problem in NP. Recall that P and NP are restricted to only decision problems, but this NP-hard class does not have this restriction and contains a much broader class of problems (e.g. maximization and minimization problems; one can even construct bizarre problems that are both NP-hard and are at least as hard as the halting problem, i.e., such a problem cannot be solved any computer). However, there are some problems that both NP-hard and are in NP, and we refer to this sub-class of NP-hard problems as *NP-complete*. Figure 3 illustrates the relationship between classes NP, NP-hard, and NP-complete.

It should seem a bit surprsing that there are problems that are at least as hard every single problem in NP (let alone the fact such a property can be shown). Luckily, it was proved by Stephen Cook and Leonid Levin in the early 70s that SAT is NP-hard (and therefore also NP-complete since SAT is in NP). The proof is beyond anything we will talk about in this class, but the consequence of this result is that it allows us to show that many other problems are also NP-hard by *composing* input transformations.

More specifically, now that we know that SAT is NP-hard, suppose we would like to show some other problem B is also NP-hard. Also suppose we are able to show that SAT $\leq B$, i.e., SAT is polynomial time reducible to B. We know that every problem in NP is polynomial time reducible to SAT; therefore, for any problem A in NP, there is an ALGO 1 that transforms an instance to A problem into a SAT instance I_{SAT}. Again, we also showed/know SAT $\leq B$, and therefore we can then feed I_{SAT} into the reduction we specified when showing SAT $\leq B$. Overall, the process gives us a polynomial time reduction from A to B since the correctness of each reduction will "propagate" across the SAT transformations to imply a yes for A implies a yes for B . Also, we know that combined construction still yields a polynomial time transformation (for ALGO 1) since polynomials are closed under composition.

The punchline is the following: *if we know a problem A is NP-hard, it is sufficient to show $A \leq B$ in order to establish B is NP-hard, as well.* In the next two sections, we will actually get our hands dirty and

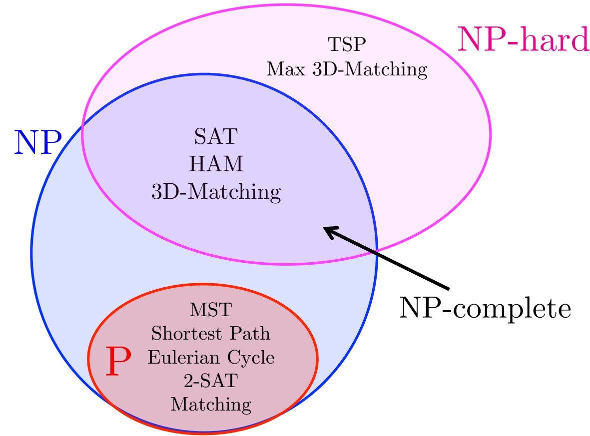

Figure 3: Relationships between P, NP, NP-complete, and NP-hard given that P $\neq$ NP.

walk through a couple reductions that show some of the problems mentioned earlier are in fact NP-hard.

3.3 k-SAT $\leq$ 3-SAT

It was mentioned earlier that 3-SAT is as hard as k-SAT for any $k \geq 3$. We will now formally show that k-SAT $\leq$ 3-SAT. If we are given that k-SAT is NP-hard, then the discussion from the previous section along with the following theorem (once established) implies that 3-SAT is NP-hard, as well.

Theorem 1. *k-SAT $\leq$ 3-SAT for $k \geq 3$, i.e., k-SAT is polynomial time reducible to 3-SAT.*

Proof. Recall for k-SAT, we are given a set of n boolean variables $X = \{x_1, \ldots, x_n\}$ and a boolean formula F_k over these variables in CNF with m clauses such that each clause contains exactly k literals. To perform our transformation, we will need modify this formula so that it is now an input instance to 3-SAT, i.e., each clause in the formula contains exactly 3 literals (we will call this transformed formula for 3-SAT F_3).

Consider a clause $c = (y_1 \vee y_2 \vee \ldots \vee y_k)$ in the k-SAT boolean formula (each $y_j \in \{x_i, \overline{x_i}\}$ for some $x_i \in X$). To construct the corresponding clauses in our 3-SAT formula F_3, we add $k-3$ new variables to the system $c_1, \ldots, c_{k-3}$ and "break-up " c into the following $k-2$ clauses :

$$(y_1 \vee y_2 \vee c_1) \wedge (\overline{c_1} \vee y_3 \vee c_2) \wedge (\overline{c_2} \vee y_4 \vee c_3) \wedge \ldots \wedge (\overline{c_{k-4}} \vee y_{k-2} \vee c_{k-3}) \wedge (\overline{c_{k-3}} \vee y_{k-1} \vee y_k) \quad (3)$$

If we perform this transformation for each clause in F_k to define our clauses for F_3, it is clear that the resulting formula will indeed by a 3-SAT formula. It is also straight forward that this transformation can be done in polynomial time (so this completes our description of ALGO 1). Given this instance for 3-SAT, we define ALGO 2 to simply query our oracle for 3-SAT and return this answer as our answer to the original k-SAT instance.

Thus, we are left with showing that the F_k is satisfiable if and only if F_3 is satisfiable. For the forward direction, assume that F_k is satisfiable. To show there is satisfying assignment for the corresponding F_3 formula, we will use the same setting of x_i variables given by the satisfying assignment for F_k, and then argue that there is a way of setting the added c_i variables (the variables we added for each clause) such that the clauses corresponding to c are satisfied in F_3.

More specially, consider a clause c from F_k. c evaluates to true in the given satisfying assignment for F_k; therefore, there exists some literal $y_j \in c$ that evaluates to true. Therefore, the clause $(c_{j-2} \vee y_j \vee \overline{c_{j-1}})$ evaluates to true in F_3. To satisfy the rest of the clauses that correspond to c in F_3, observe that we can set $c_1, \ldots, c_{j-3}$ to be true and $c_{j-1}, \ldots c_{k-3}$ to be false. If we repeat this process for each clauses $c \in F_k$, the assignment we produce for our 3-SAT instance will satisfy F_3.

For the reverse direction, assume we have a satisfying assignment to F_3. For the set of clauses corresponding to a clause c in F_k, it is not too hard to see that there must be at least one $y_j \in c$ that evaluates to true from the F_3 assignment (the c_i variables are not capable of satisfying all of c's 3-SAT clauses on their own). Therefore, this implies that if we take the same setting of x_i variables from the F_3 satisfying assignment, there will be at least one literal in each clause that evaluates to true. Thus, repeating this argument for each clause implies a satisfying assignment for F_k, which completes our proof. □

3.4 HAM $\leq$ **TSP**

(We'll leave this as a practice problem for now. See page 259 of DPV for a solution. Note that they refer to HAM as "Rudrata Cycle").