Lecture 9: More NP-complete problems and Oracle machines

Anup Rao

January 31, 2023

In the last class, we introduced the concept of NP and NP-completeness, and showed that a few problems are NP-complete. In this lecture, we begin by giving two more examples of NP-complete problems.

Hamiltonian Path

Given a directed graph G, a Hamiltonian path is a path that visits every vertex of the graph exactly once. We define the function

$$\mathsf{HPATH}(G) = \begin{cases} 1 & \text{if } G \text{ has a Hamiltonian path} \\ 0 & \text{otherwise.} \end{cases}$$

Theorem 1. HPATH *is* **NP***-complete.*

Proof Given a path in the graph, one can check in polynomial time whether or not it is a Hamiltonian path. Thus $\mathsf{HPATH} \in \mathbf{NP}$ using the path as a witness. Next we show that you can reduce 3SAT to HPATH, proving that HPATH is **NP**-hard.

Suppose the formula has n variables and m clauses. We shall construct a graph on $(2m+2)n+2$ vertices that encodes assignments to the formulas as follows. We start by constructing a graph that will contain $(2m+2)n$ vertices named $v_{i,j}$, where $i \in \{1, 2, \ldots, n\}$ and $j \in$. For every i and $1 \leq j < j+1 \leq k$, we have the edges $(v_{i,j}, v_{i,j+1})$ and $(v_{i,j+1}, v_{i,j})$. Thus these vertices can be thought of as arranged in n rows, where in each row the path can go left or right. For every $1 \leq i < i+1 \leq n$, we add the edges

$$(v_{i,1}, v_{i+1,1}), (v_{i,1}, v_{i+1,n}), (v_{i,n}, v_{i+1,1}), (v_{i,n}, v_{i+1,1}).$$

Finally we add two special vertices s, t, with edges

$$(s, v_{1,1}), (s, v_{1,n}), (v_{n,1}, t), (v_{n,n}, t).$$

By construction, every Hamiltonian path in the graph must start at s and end at t, and must traverse each row in order. Each row can be traversed in either left to right or right to left fashion. We shall imagine that traversing the row left to right corresponds to

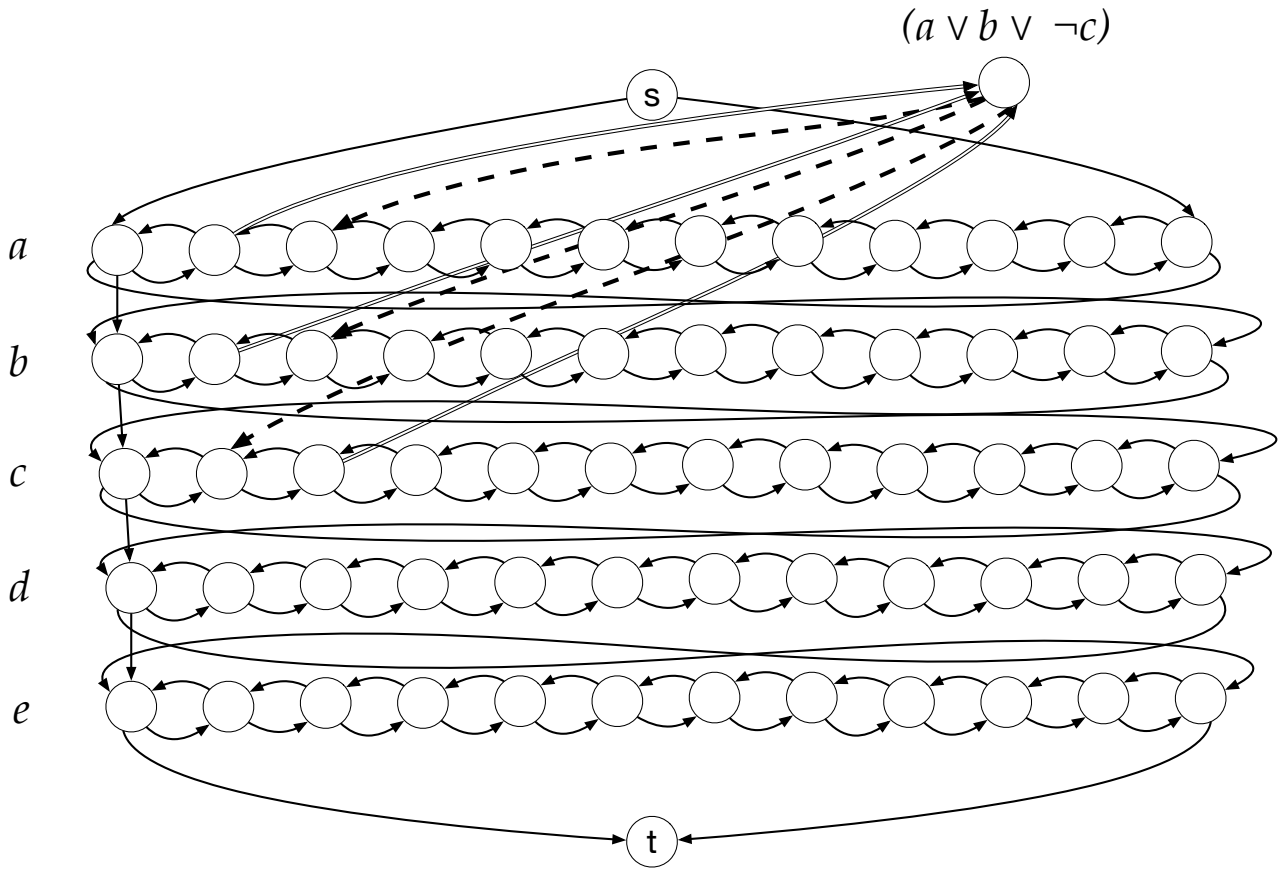

Figure 1: An example showing how to generate a directed graph for the Hamiltonian path problem using a single clause from the formula.

assigning the i'th variable the value 0, and traversing it the other way corresponds to assigning the value 1.

Next we add some vertices to encode the constraints given by the clauses. Without loss of generality we assume that each clause contains a variable at most once (since we can always reduce the formula to this case). For the j'th clause C_j, we add the vertex c_j. For every variable x_i that the clause contains unnegated, we add the edges $(v_{i,2j}, c_j), (c_j, v_{i,2j-1})$. For every variable x_j that is contained in the clause as $\neg x_j$, we add the edges $(v_{i,2j-1}, c_j), (c_j, v_{i,2j})$. By construction, any Hamiltonian path that takes the edge $(v_{i,2j}, c_j)$, must take $(c_j, v_{i,2j-1})$ next, or $v_{i,2j-1}$ will never be visited. Similarly, any Hamiltonian path that takes the edge $(v_{i,2j}, c_j)$ must take $(c_j, v_{i,2j-1})$ next. We claim that the graph has a Hamiltonian path if and only if the formula is satisfiable.

Indeed, if the formula is satisfiable, then traverse each row in the direction corresponding to the satisfying assignment. Since each clause is satisfied by some variable, we can visit the vertex for the

clause when we traverse the first variable that satisfies it. Conversely, if there is a Hamiltonian path, then the construction ensures that this path corresponds to an assignment to the variables, and this path must visit every clause vertex, which guarantees that each clause vertex is satisfied by some variable. ■

Subset Sum

In the subset sum problem, the input is a collection of numbers $a_1, \ldots, a_k$, as well as a target number t. The goal is compute whether or not some subset of the numbers $a_1, \ldots, a_k$ sums to t.

$$\mathsf{SubSum}(a_1, \ldots, a_k, t) = \begin{cases} 1 & \text{if there is a subset } S \subseteq \{1, 2, \ldots, n\} \text{ such that } \sum_{i \in S} a_i = t, \\ 0 & \text{otherwise.} \end{cases}$$

Theorem 2. SubSum *is* **NP***-complete.*

We sketch the proof. SubSum is in **NP**, since there is an obvious polynomial time computable verifier for the problem. The witness is a subset S, and the verifier simply checks that $\sum_{i \in S} a_i = t$, which can be done in polynomial time.

To show that SubSum is **NP**-hard, we shall show that

$$\mathsf{3SAT} \leq_P \mathsf{SubSum}.$$

We describe the polynomial time reduction next. Given a 3-sat formula ϕ, our algorithm needs to output numbers $a_1, \ldots, a_k$ and t such that $\mathsf{SubSum}(a_1, \ldots, a_k, t) = 1$ if and only if ϕ is satisfiable.

Suppose ϕ has n variables and m clauses. Then, we will have $k = 2n + 2m$, and all of the numbers $a_1, \ldots, a_k$ and t will be $n + m$ digit numbers, written in base 10. Moreover, all the digits of $a_1, \ldots, a_k$ will be either 0 or 1, and the numbers will be chosen in such a way that adding any subset of $a_1, \ldots, a_k$ will never produce a carry.

For each variable x_i of the formula ϕ, we shall have two numbers: t_i and f_i. The i'th digit of t_i and f_i will be set to 1 and all of the remaining $n - 1$ digits in the first n digits will be set to 0. Meanwhile, in the target number t, all of the first n digits will be set to 1. This choice ensures that choosing any subset of $t_1, f_1, \ldots, t_n, f_n$ that sums to t corresponds to choosing either t_i or f_i to be included in the set, for each i. In other words, a subset of these numbers that sums to t corresponds to a truth assignment to the variables $x_1, \ldots, x_n$.

Next, we need to add more digits to ensure that this truth assignment satisfies all the clauses. For every i, j, if x_i occurs in the j'th

Example: suppose we are given the formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_1, \vee \neg x_3 \vee \neg x_4) \vee (\neg x_2, \vee \neg x_3 \vee x_4)$. There are 4 variabels and 4 clauses, so the polynomial time reduction will generate 16 numbers, each with 8-digits, and a target number with 8-digits:

$t_1 = 10001000$
$f_1 = 10000010$
$t_2 = 01000000$
$f_2 = 01001101$
$t_3 = 00101100$
$f_3 = 00100011$
$t_4 = 00010101$
$f_4 = 00010010$
$b_1 = 00001000$
$c_1 = 00001000$
$b_2 = 00000100$
$c_2 = 00000100$
$b_3 = 00000010$
$c_3 = 00000010$
$b_4 = 00000001$
$c_4 = 00000001$

The target number will be:

$t = 11113333.$

clause, we make the $n + j$'th digit of t_i 1. If $\neg x_i$ occurs in the j'th clause, we make the $n + j$'th digit of f_i 1. All other digits (upto the $n + m$'th digit) of t_i, f_i are set to 0. This choice ensures that if the subset chosen satisfies the j'th clause, then the j'th digit of the sum will be either 1, 2 or 3. Finally, we add two numbers b_j, c_j, which are 0 in all digits, except for the j'th digit. The j'th digit of both numbers is 1. This ensures that if the j'th clause is satisfied by the assignment, then one can pick 0, 1 or 2 elements of $\{b_j, c_j\}$ to add to the subset, so that the sum of the j'th digits is 3.

The problem with diagonalization

THE ONLY WAY WE KNOW how to prove lower bounds on the running time of Turing Machines is via diagonalization. Can we hope to show that $\mathbf{P} \neq \mathbf{NP}$ by some kind of diagonalization argument? In this lecture, we discuss an issue that is an obstacle to finding such a proof.

Definition 3 (Oracle Machines)**.** *Given a function $O : \{0,1\}^* \to \{0,1\}$, an* oracle-machine *is a Turing Machine that is allowed to use a special oracle tape to make queries to O. Each query to O takes unit time.*

We can define $\mathbf{P}^O$, $\mathbf{NP}^O$ as functions computable in poly time (resp nondeterministic poly time) with oracle access to O.

Then we have the following theorem:

Theorem 4. *There exists an oracle A such that $\mathbf{P}^A = \mathbf{NP}^A$, and an oracle B such that $\mathbf{P}^B \neq \mathbf{NP}^B$.*

The theorem gives a hint about one of the ways in which it will be hard to determine whether or not $\mathbf{P} = \mathbf{NP}$. Any such proof must not work in the *relativized* worlds where access to A, B is permitted. On the other hand, the kinds of proofs that we have seen using diagonalization *do relativize*—the same argument would work even if the machines have oracle access to some oracle O.

Proof Let A be the function that on input α, x outputs 1 if and only if $M_\alpha(x)$ outputs 1 in $2^{|x|}$ steps. Then $\mathbf{P}^A = \mathbf{EXP}$, since every exponential time computation can be simulated with access to A, and every query to A can be simulated in exponential time. Also $\mathbf{NP}^A = \mathbf{EXP}$, since in exponential time we can simulate all queries to A and simulate all nondeterministic choices.

To simulate a machine M_α, that runs in time 2^{n^c}, we first create a new machine M'_α that runs M_α on the first $n^{1/c}$ bits of its input. Then we call the oracle on $M_{\alpha'}(y)$, where y is the input of length n^c with x as the first $n^{1/c}$ bits of y.

The second part is more interesting. We shall define an oracle $B : \{0,1\}^* \to \{0,1\}$ and a function $f \in \mathbf{NP}^B$ such that $f \notin \mathbf{P}^B$. f is

defined in terms of B as follows:

$$f(x) = \begin{cases} 1 & \text{if there exists } y \text{ such that } |y| = |x| \text{ and } B(y) = 1, \\ 0 & \text{else.} \end{cases}$$

We first show that $f \in \mathbf{NP}^B$: a non-deterministic machine can guess y of the same length as x, and make a single query to verify that $B(y) = 1$.

To define B, we shall use diagonalization. Let $M_1, M_2, \ldots, M_i, \ldots$, be an enumeration of all machines that query B, with the feature that every machine occurs infinitely often in the sequence. (Such an enumeration exists if we allow our programming language to have redundant lines). Our goal is to make sure that the $i'th$ machine fails to compute the correct value of $f(x)$ in time $2^{n/10}$, for some n where $n = |x|$. To do this we define the value of B gradually. We define the value of B in phases. After each phase, we shall have defined the value of B on a finite set of strings.

In Phase i, let t be so large that the value of B is not yet defined on each string of length t. Then run the i'th machine $M_i(1^t)$ for $2^{t/10}$ steps. Each time M_i queries a string of B whose value has not yet been defined, return 0 and define the value of B on that string to be 0. If M_i halts with value 1, then set B to be 0 on all strings of length t. If M_i halts with value 0, then pick a string y of length t that $M_i(1^t)$ did not query (note that such a string always exists since there are 2^t binary stings of length t and M_i did not take more than $2^{t/10}$ steps), and set $B(y) = 1$.

Set the value of B on strings that are not defined by the above process to be 0.

Suppose for the sake of contradiction that $f \in \mathbf{P}^B$. Then consider the machine M that computes f. Since redundant lines of code do not change the computation of the machine, M actually shows up an infinite number of times in the enumeration over all machines.

Let i be the index such that the i'th machine in the enumeration is equivalent to M and t be such that $M_i(1^t)$ was used to define B on strings of length t during the i'th phase, with $2^{t/10}$ bigger than the running time of M on inputs of length t. Because $f(1^t) \neq M(1^t)$, M does not compute f. ■