Complexity Theory: What are "hard" problems?

Prepared by Eli on November 21, 2024

Instructor's Handout

This file contains solutions and notes.
Compile with the "nosolutions" flag before distributing.
Click `[here]` for the latest version of this handout.

Part 1: Basics: What is a 'language?'

Definition 1:
A *symbol* (or *character*) is our smallest building block and is simply any letter or picture.

An *alphabet* Σ is a **finite**, non-empty set of symbols.

A *string* w is a finite sequence of symbols from some alphabet. We denote the length of a string (the number of symbols in the string with repeats counted multiple times) by $|w|$.

For example, $\Sigma = \{0, 1\}$ is the alphabet of binary digits. $\Sigma = \{0, 1, \ldots, 9\}$ is the alphabet of decimal digits. $\Sigma = \{A, B, \ldots, Z\}$ is the alphabet of (capital) English letters. Note that the *empty string*, denoted ϵ, can be formed from any alphabet.

Problem 2:
Let $\Sigma = \{a, b, c\}$. How many strings of length at most n can be formed from characters in Σ?

Solution

Separating by length, the solution is $3^0 + 3^1 + 3^2 + \cdots + 3^n$. Students should note the existence of the empty string.

Problem 3:
Let $\Sigma = \{0, 1, \ldots, 9\}$. How many strings of length 4 can be formed from characters in Σ such that each constructed string has no repeated characters? Generalize this to strings of length k for $k \leq 10$. How would your solution change if our alphabet Σ had n unique characters for some positive integer n?

Solution

$10 \cdot 9 \cdot 8 \cdot 7$, or alternatively, $\frac{10!}{6!}$. In general, $\frac{10!}{(10-k)!}$, and furthermore with an alphabet of size n, $\frac{n!}{(n-k)!}$

Definition 4:
We denote by Σ^* the set of all possible strings (of any length) that can be formed from an alphabet Σ, including the empty string ϵ. We may refer to Σ^* as the *closure* of Σ.

Problem 5:
Let $\Sigma = \{a, b, c, 0, 1\}$. Give five examples of elements in Σ^*, the closure of Σ.

Solution

Any string formed from these characters is acceptable, including the empty string.

Problem 6:
Let Σ be an alphabet consisting of k symbols. How many strings exist in Σ^*, the closure of Σ?

Solution

This is a trick question. Since every alphabet is non-empty, and strings in Σ^* can be of arbitrary length, the closure Σ^* is of infinite size no matter the number of symbols k in Σ.

Definition 7:
A *language* formed over Σ is any subset of Σ^*. In other words, a language is a set of strings formed from Σ.

Problem 8:
Let $\Sigma = \{a, b, c, d, e\}$. We would like to form a language L over Σ^* such that L consists of all palindromes of length at most n for some positive integer n. How many different strings exist in L?

Solution

If n is even, this is simply equal to the number of strings of length $n/2$ formed over Σ, which is $5^{\frac{n}{2}}$. If n is odd, the solution is nearly the same, except we may choose the center symbol freely for another 5 possibilities, giving $5^{\frac{n}{2}+1}$.

Part 2: Turing Machines

As you saw in the last section, languages are used as mathematical representations of "problems." This is done, perhaps counterintuitively, by precisely describing the set of solutions to the problem. Then, given an input x which is an instance of the problem, the goal of the computation is to determine whether $x \in L$. That is, whether x is a solution to the problem that L represents. From a computational perspective, describing the exact set of solutions is the most straightforward way to formally represent a problem.

Our goal is to formally analyze the complexity of a particular problem. That is, how computationally difficult a particular problem is to solve. To do so, we must also define a mathematical representation of the tool we would use to solve such a problem: the computer. This topic is complex and we won't have time to explore it in detail, but for completeness, we will define a Turing machine and learn its basic properties.

Definition 9:
A *Turing machine* consists of an infinitely long tape and a moving head. M performs computations by receiving the input x on the tape, performing various operations (according to its programming) using the tape as its "scratch paper", reaching either an accept or reject state signifying the output of the computation. Formally, M is defined as a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{accept}}, q_{\mathsf{reject}})$ where:

- Q is a finite set of states that M may exist in at any point of the computation.
- Σ is the input alphabet (a finite set of symbols excluding the blank symbol) which the input to M is written in.
- Γ is the tape alphabet used for internal computation, which contains all symbols from Σ as well as a special blank symbol $\perp$ used to represent empty tape cells.
- δ is the transition function which defines the behavior of the machine by dictating what operation is performed at any point during computation. An 'operation' consists of reading the current symbol (where the head is pointing), checking the current state, and using that

information to update the state, write a symbol to the tape, and then move the head either left or right. Formally, δ takes an input in $Q \times \Gamma$ (the input state and the current symbol) and gives an output in $Q \times \Gamma \times \{L, R\}$ (the output state, the symbol to write, and the direction to move the head).
- q_0 is the starting state of the machine.
- q_{accept} is the accept state, signifying the machine has finished computation and determined that $x \in L$.
- q_{reject} is the reject state, signifying the machine has finished computation and determined that $x \notin L$. Importantly, $q_{\mathsf{reject}} \neq q_{\mathsf{accept}}$.

Okay now take a deep breath! That was a lot of information, so let's do a few problems to explore how Turing machines work.

Problem 10:
Consider the following Turing machine M. Its set of possible states are $Q = \{q_0, q_{\mathsf{even}}, q_{\mathsf{odd}}, q_{\mathsf{accept}}, q_{\mathsf{reject}}\}$. Its input alphabet is $\Sigma = \{0, 1\}$. Its tape alphabet is $\Gamma = \{0, 1, \perp\}$. Its transition function δ behaves as follows: no matter what, the head always moves to the right. If the symbol at the current head position is a 0, the current state remains unchanged. If the symbol at the current head position is a 1 and the current state is q_{odd}, the new state is q_{even}. If the symbol at the current head position is a 1 and the current state is q_{even} or q_0, the new state is q_{odd}. If the symbol at the current head position is $\perp$ and the current state is q_{odd}, the new state is q_{accept}, and the machine halts. If the symbol at the current head position is $\perp$ and the current state is q_{even} or q_0, the new state is q_{reject}, and the machine halts. During all of these steps, the head writes to the tape the same symbol it reads (effectively not writing anything). Formally:

$$\delta(q, b) = \begin{cases} (q, b, R) & \text{if } b = 0, \\ (q_{\mathsf{even}}, b, R) & \text{if } b = 1 \text{ and } q = q_{\mathsf{odd}}, \\ (q_{\mathsf{odd}}, b, R) & \text{if } b = 1 \text{ and } q \in \{q_0, q_{\mathsf{even}}\}, \\ (q_{\mathsf{accept}}, b, R) & \text{if } b = \perp \text{ and } q = q_{\mathsf{odd}}, \\ (q_{\mathsf{reject}}, b, R) & \text{if } b = \perp \text{ and } q \in \{q_0, q_{\mathsf{even}}\} \end{cases}$$

Determine the output of M (accept or reject) for the following strings: 010, 11, 001, 1111.

Solution

010: accept. 11: reject. 001: accept. 1111: reject.

Problem 11:
Describe in your own words what the Turing machine M does.

Solution

M determines whether there are an odd number of 1's in the input.

Problem 12:
Using set notation, what language L is decided by the machine M?

Solution

$L = \{x \in \Sigma^* : \text{ the number of 1's appearing in } x \text{ is odd.}\}$

Problem 13:
We would like to construct a Turing machine M' such that M' accepts all inputs that M rejects, and M' rejects all inputs that M' accepts. Using our description of M as a starting point, how can we

modify M to get such a machine M'?

Solution

We need only change the transition function δ, and we simply swap all instances of q_{accept} and q_{reject}.

Definition 14:
For a particular Turing machine M, a *configuration* describes the combination of the current state, head position, and contents of its tape. Two configurations are *distinct* if they differ in any of these characteristics.

Problem 15:
Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ be a Turing machine that only uses N cells of its infinite tape. How many possible distinct configurations are there for M?

Solution

Each cell contains a symbol from Γ resulting in $|\Gamma|^N$ possibilities for the contents of the tape. The head could be in any of N positions, and there are $|Q|$ possible states the machine could be in. So the total number of distinct configurations is $|\Gamma|^N \cdot N \cdot |Q|$.

Problem 16:
Let the input alphabet be $\Sigma = \{0, 1\}$ and let $L = \{x \in \Sigma^* : x \text{ consists of only 1's}\}$. Imagine that you are given an input $x \in \Sigma^*$ one symbol at a time. After reading each symbol, you may either move on and read the next symbol (which may be the empty symbol $\perp$ if you are currently reading the last symbol of x), or you may claim that $x \in L$, or you may claim that $x \notin L$. Describe what strategy you would use to correctly determine whether or not $x \in L$.

Solution

For each symbol, if it is a 1, continue to the next symbol. If it is a 0, reject x. If it is an empty symbol, accept x.

Problem 17:
How would you write a Turing machine M to determine if $x \in L$? Focus on defining the transition function δ and determining whether or not you need any additional states in Q besides the initial, accept, and reject states.

Solution

Our transition function starts on the first symbol of the input. Notably, our Turing machine doesn't need to do any writing, only reading, and does not need to perform any state transitions besides eventually accepting or rejecting. It simply scans each symbol of the input from left to right. If it encounters a 1, M stays in the initial state, writes the current symbol, and moves the head to the right. If it encounters a 0, M transitions to the reject state, writes the current symbol, moves either direction (lets say right), and halts. The only nuance here is that M can't "check ahead" to see if it's on the last symbol of the input. Thus we need one more rule: if it encounters a $\perp$, M transitions to the accept state, writes the current symbol,

Solution (continued)

moves either direction (lets say right), and halts. Formally:

$$\delta(q,b) = \begin{cases} (q,b,R) & \text{if } b = 1, \\ (q_{\text{reject}},b,R) & \text{if } b = 0, \\ (q_{\text{accept}},b,R) & \text{if } b = \perp . \end{cases}$$

Problem 18:
Let the input alphabet be $\Sigma = \{a,b\}$ and let $L = \{a^n b^n : n > 0\}$. That is, L consists of all strings consisting of some number of a symbols, followed by the same number of b symbols. Suppose you are given an input $x \in \Sigma^*$ written on a piece of paper with a pencil and your only tool is an eraser. How would you determine whether $x \in L$? Assume you do not have the ability to count the symbols of x.

Solution

Erase the leftmost a, then the leftmost b, then repeat until either the string is empty (in which case you accept) or you have run out of either a or b symbols (in which case you reject).

Problem 19:
For the language L given in Problem 18, describe a Turing machine M to determine if $x \in L$.

Solution

There are many solutions, so make sure to examine the student's solution carefully. One possibility is to use an additional symbol $X \in \Gamma$ to one by one 'delete' pairs of a, b. More specifically, M reads the first symbol, confirms that it is an a (otherwise it transitions to the reject state), then replaces the a with X. It then scans cell by cell until it reaches a b (if it reaches the end of the tape without seeing one, it rejects), and then replaces the first b with X as well. It then returns to the left until it reads an X (indicating it has returned to the most recently deleted a symbol). It moves one cell to the right, and if it reads an a, it repeats the above process. If it reads an X, that means all the a symbols at the start of the string have already been replaced by X. If $x \in L$, then all the b symbols should also have already been replaced by X, and if $x \notin L$, then there should be some leftover b symbols at the end of the tape (if there were fewer b than a, we would have already rejected due to reaching the end of the tape without seeing a b at some point in the above repeated process). Thus at this point M scans the remainder of the tape. If any cell contains the symbol b, M rejects. If it reaches the empty cell (containing $\perp$) at the end of the tape without rejecting, M accepts.

Definition 20:
We say a language L is *computable* if there exists a Turing machine M which accepts an input x if $x \in L$ and rejects otherwise. A language that is not computable is called *undecidable*.

Problem 21: Challenge
Since we can write down a description of a Turing machine M by writing down its transition function, states, etc., we can actually give that description of M as input to another Turing machine M'. Suppose we want to write a Turing machine M_H which takes as input a description of another Turing machine M and a string w. M_H should determine whether or not M halts (eventually either accepts or rejects) when given the input w, as opposed to getting stuck in some kind of infinite loop. Formally, M_H should decide the language $H = \{(M,w) : M \text{ halts on input } w\}$. Prove that no such M_H exists. In other words, prove that H is an undecidable language. (Hint 1: since M_H is a Turing machine, and M_H takes a Turing machine as input, M_H can also be used as an input to M_H). (Hint

2: the input w can be *any string*, including a Turing machine).

Solution

Suppose toward a contradiction that M_H exists. Define a new Turing machine D as follows:

- $D(M)$ invokes M_H to check whether a Turing machine M halts when given *itself* as an input.
- If $M_H(M, M)$ outputs **yes** (meaning M does indeed halt on itself), then $D(M)$ **enters an infinite loop**, and does not halt.
- If $M_H(M, M)$ outputs **no** (Meaning M enters an infinite loop when given itself as input), then $D(M)$ halts (either accepts or rejects, doesn't matter).

Now, examine what happens when we try to give D as input to D and compute $D(D)$. According to our construction, if $M_H(D, D)$ outputs yes (meaning D halts on itself), then $D(D)$ enters an infinite loop and does not halt, which is a contradiction. Similarly, if $M_H(D, D)$ outputs no (meaning $D(D)$ loops infinitely), then $D(D)$ halts, which is again a contradiction. Thus, in either case, we reach a contradiction, and can conclude that no such M_H exists that can decide the language H, and H is undecidable.

Problem 22:
Traditionally, we think of the tape of M being infinitely long in one direction (to the right). Suppose instead that we had Turing machine M' with a tape that is infinitely long in *both* directions. Prove or disprove the following claim: any language L which is computable using a traditional Turing machine M is also computable using a bidirectional Turing machine M'.

Solution

We can simulate the bidirectional infinite tape using the unidirectional infinite tape by splicing the two directions together. If we imagine labeling each index of the tape with an integer, then M has indices $\{0, 1, 2, \ldots\}$ and M' has indices $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$. We can map indices of M' to indices of M in a number of different ways, a simple one being $-x \to 2x - 1$, $x \to 2x$. Thus the indices of M' when reordered become $\{0, -1, 1, -2, 2, -3, 3, \ldots\}$. Reprogramming the transition function to operate on the reordered tapes is trivial, it just requires a lot more moving of the head without writing over anything or transitioning state.

Problem 23:
Can you think of any significant differences between the Turing machine model and a real life computer that might make it somewhat unrealistic when applied to real-life computations? If so, how would you modify the model to be more realistic?

Solution

The most obvious is that unlike real computers, Turing machines have infinite memory. This can be fixed by giving it a finitely sized tape rather than an infinite one. This makes the analysis much more complicated and removes the ability to analyze what is "theoretically computable" but does make the analysis more grounded in reality. I'm also interested to see what else the kids come up with. It's worth noting that although our description of Turing machines is limited to decision problems (yes/no), any search problem in NP can be reduced to a polynomial number of invocations of the relevant decision problem.

Part 3: Big O Notation

Now that we've defined our model and explored what is computable and what isn't, we'd like to narrow our focus on problems we can compute and identify which problems require more resources

and which problems require fewer resources. To do so, we'll need some more notation. Big O notation is used to describe an upper bound of an algorithm's runtime (or space requirements) in terms of the input size. It gives an asymptotic measure of how the runtime or space grows as the input size n becomes very large.

Definition 24:
Given two functions $f(n)$ and $g(n)$, we say that $f(n) = O(g(n))$ if and only if there exist positive constants C and n_0 such that for all $n \geq n_0$,

$$f(n) \leq C \cdot g(n)$$

In this context, $f(n)$ is the function we are trying to bound (e.g. our algorithm's runtime) and $g(n)$ is a simpler function providing an upper bound (e.g. some simple polynomial).

Informally speaking, saying $f(n)$ is $O(g(n))$ means you can find a positive scalar C which makes it so that $f(n)$ is less than $C \cdot g(n)$ when n is large.

Problem 25:
Let $f(n) = n$ and $g(n) = 2n + 5$. Show that $f(n) = O(g(n))$.

Solution

Straightforwardly, $C = \frac{1}{2}$ and $n_0 = 0$ (any value works for n_0 here).

Problem 26:
Let $f(n) = n^5 + 31$ and $g(n) = n^6$. Show that $f(n) = O(g(n))$.

Solution

$C = 1$ and $n_0 = 2$ (any larger value for n_0 also satisfies the definition).

Problem 27:
Let $f_1(n) = 2^n + 312, f_2(n) = \frac{n^n}{2^3 2}, f_3(n) = n!$. Order f_1, f_2, f_3 such that if f_a appears before f_b then $f_a = O(f_b)$.

Solution

(f_1, f_3, f_2) due to simple replacement tricks within the inequalities after ignoring constant terms. Some care must be taken to properly choose C and n_0 to be formal.

Problem 28:
Write an algorithm for primality testing that requires at most $O(\sqrt{k})$ arithmetic operations (additions, subtractions, multiplications, or divisions), where k is the integer we wish to determine the primality of. In other words, given an integer k, your algorithm should determine whether it is prime "efficiently", i.e. in $O(\sqrt{k})$ operations.

Solution

For each integer $m \in \{1, \ldots, \lceil \sqrt{k} \rceil\}$, check whether $m|k$. If every check fails, then k is prime. Otherwise, k is composite. The algorithm is correct because for any pair of factors p, q such that $pq = k$, we must have either $p \leq \sqrt{k}$ or $q \leq \sqrt{k}$. This algorithm requires exactly $\sqrt{k}$ operations which is trivially $O(\sqrt{k})$ with $C = 1, n_0 = 1$. Notably one can optimize this and only check prime values for m but that requires prerequisite knowledge of the primality of all integers less than k.

Part 4: Complexity Classes

Definition 29:
A *complexity class* is a set of languages grouped together based on the resources required to solve them such as time or space. As a reminder, a language represents a problem with a particular set of solutions which are the elements of the language, so you can think of a complexity class as a set of problems.

Earlier we found that there exists a non-empty set U consisting of undecidable languages which cannot be computed even given an infinite amount of computational resources. Although this is similar to a complexity class, we consider it outside the hierarchy of complexity classes.

Definition 30:
The class P consists of all languages which can be solved in an amount of time which is bounded by some polynomial function of the input length. Formally, $L \in \mathsf{P}$ if and only if there exists a Turing machine M that decides L in $O(n^k)$ steps for some constant k, where n is the **length** of the input.

Problem 31:
Revisit your solutions to Problem 17 and Problem 19. Do the Turing machines you constructed solve the problem in polynomial time? Give an upper bound on the time required for each of your solutions using Big O notation.

Solution

Depends on their solution, but most likely yes. Our solutions require $O(n)$ time for Problem 9 and $O(n^2)$ time for Problem 10.

Definition 32:
The class PSPACE consists of all languages which can be solved in an amount of *space* which is bounded by some polynomial function of the input length. Formally, $L \in \mathsf{PSPACE}$ if and only if there exists a Turing machine M that decides L using $O(n^k)$ space (cells of the infinite tape) for some constant k, where n is the length of the input.

Problem 33:
Prove that $\mathsf{P} \subseteq \mathsf{PSPACE}$.

Solution

A Turing machine that takes at most $t(n)$ steps uses at most $t(n)$ cells of the infinite tape, since the head moves at most one cell during each step.

Definition 34:
The class $\mathsf{EXPTIME}$ consists of all languages which can be solved in an amount of time which is bounded by some *exponential* function of the input length. Formally, $L \in \mathsf{EXPTIME}$ if and only if there exists a Turing machine M that decides L using $O(2^{p(n)})$ steps for some polynomial function p, where n is the length of the input.

Problem 35:
Prove that $\mathsf{PSPACE} \subseteq \mathsf{EXPTIME}$. (Hint: revisit your solution to Problem 8).

Solution

If a Turing machine repeats the same configuration, it will necessarily get stuck in an infinite loop. Therefore, the running time for a Turing machine is bounded from above by the number of distinct configurations. For a PSPACE machine, this number of configurations is exponential

Solution (continued)

(just replace N from Problem 8 with $p(n)$ for some polynomial p).

Definition 36:
The class NP consists of all languages for which a solution can be *verified* in a polynomial amount of time. Formally, $L \in \mathsf{NP}$ if and only if there exists a Turing machine V which runs in polynomial time and behaves as follows: given an input x, if $x \in L$ then there exists a witness w (which you can think of as the solution to x that V is verifying) such that $V(x, w)$ accepts, and if $x \notin L$ then for all possible w, $V(x, w)$ rejects.
As an analogy, think of x as the starting configuration for a Sudoku puzzle. Then w would be the completed Sudoku puzzle, and V would check to make sure that x and w match on overlapping cells and that w does not violate the rules of a Sudoku solution.

Problem 37:
Prove that $\mathsf{P} \subseteq \mathsf{NP}$

Solution

If a Turing machine M can simply solve the problem x in polynomial time then there is no need for a witness at all, $M(x)$ will simply tell us directly whether $x \in L$ so we can construct an NP machine M' by simply running M and ignoring any witness w.

Problem 38:
Try to come up with an example of a problem that is easy to verify, but hard to solve. Can you prove that your problem exists in NP but not in P? If you can, congratulations! You just solved one of the Millennium problems of mathematics and earned a prize of one million dollars.

Solution

A simple example is Sudoku, but obviously no proof exists that it is actually hard to solve.

Part 5: Reductions and NP-Completeness

Now that we've explored some classic complexity classes, you might have realized that proving which problems are 'hard' is not such an easy task. All the inclusions we proved above have the potential to actually be set equalities, and we have no way of knowing whether these sets exist in a complicated hierarchy, or whether they all 'collapse down' into a single class, or whether the reality is somewhere in between. Since we haven't figured out any absolute ordering within these complexity classes, we can't prove claims about the hardness of particular problems in isolation. the best we can do is to prove how hard problems are *relative to each other*. To do this, we use a technique called a reduction.

Definition 39:
Let A and B be languages over Σ^*. We say that A is polynomial-time reducible to B (written $A \leq_p B$) if, given a polynomial-time procedure M_B that decides B, we can construct a polynomial-time procedure M_A that decides A by calling M_B as a subroutine.

Problem 40:
Let p be prime, and let $A = \{a, b, c \in \mathbb{N} : abc \text{ is a multiple of } p\}$. Let $B = \{k \in \mathbb{N} : k \text{ is a multiple of } p\}$. Show that $A \leq_P B$.

Solution

Given M_B which decides B, we construct M_A as follows: simply evaluate $M_B(a), M_B(b)$, and $M_B(c)$. If any of these evaluations accept, then M_A accepts. This works because for prime p, $p|abc \iff p|a \vee p|b \vee p|c$.

Definition 41:
A directed graph $G = (V, E)$ is a set of vertices and edges such that $e = (v_1, v_2) \in E$ represents a directed edge from source vertex v_1 to destination vertex v_2. A path in G is a series of vertices $(v_1, v_2, \ldots, v_n)$ such that $(v_i, v_{i+1}) \in E$ for each $i \in [1, n-1]$. A cycle in G is a path whose starting and ending node are adjacent. That is, $(v_n, v_1) \in E$. A Hamiltonian path in G is a path that visits every vertex of G exactly once. Similarly, a Hamiltonian cycle in G is a cycle that visits every vertex of G exactly once.

Problem 42:
Draw a directed graph with at least 5 nodes and 7 edges that contains a Hamiltonian path, but no Hamiltonian cycle.

Solution

The easiest solution is a ring of 5 nodes $(v_0, v_1, v_2, v_3, v_4)$ connected in a chain by 4 edges $((v_0, v_1), (v_1, v_2), (v_2, v_3), (v_3, v_4))$ and then 3 additional edges which do not connect v_4 to v_0. For example, $(v_0, v_2), (v_1, v_3), (v_0, v_3)$.

Problem 43:
Does there exist a graph G containing a Hamiltonian cycle but no Hamiltonian path? Justify your answer.

Solution

No, a Hamiltonian cycle is by definition a Hamiltonian path with the additional property that it is a cycle.

Problem 44:
Let $\mathsf{HAMPATH} = \{G = (V, E) : G \text{ contains a Hamiltonian path}\}$ and $\mathsf{HAMCYCLE} = \{G = (V, E) : G \text{ contains a Hamiltonian cycle}\}$. Prove that $\mathsf{HAMPATH} \leq_p \mathsf{HAMCYCLE}$.

Solution

Given M_B that solves HAMCYCLE, we must construct an M_A that solves HAMPATH. To do so, take the graph $G = (V, E)$ for which we want to determine if $G \in$ HAMPATH. Add a vertex u to G such that $(v, u) \in E$ and $(u, v) \in E$ for all $v \in V$. It is then simple to show that if this new graph G' contains a Hamiltonian cycle $(v_0, \ldots, v_n, u)$, then $(v_0, \ldots, v_n)$ is a Hamiltonian path in G. Similarly, if $(v_0, \ldots, v_n)$ is a Hamiltonian path in G, then $(v_0, \ldots, v_n, u)$ is a Hamiltonian cycle in G'.

Problem 45:
So far we have only considered the 'decision' version of problems, e.g. 'does there exist a Hamiltonian path in this graph?' But what if we actually want to find the Hamiltonian path in the graph? Luckily, for many useful problems, the search version of the problem is polynomial-time reducible to the decision version of the problem. Show that given an efficient procedure M_H for solving the

decision version of HAMPATH (telling us whether a graph G contains a Hamiltonian path), we can construct an efficient procedure that actually *finds* the Hamiltonian path.

Solution

Given G, compute $M_H(G)$. If it rejects, there is no Hamiltonian path. If it accepts, remove an edge from G to get G'. Now, compute $M_H(G')$. If it rejects, then the edge we removed must have been part of the Hamiltonian cycle, so add it back and remove a different edge. Otherwise, if $M_H(G')$ accepts, then G' still contains a Hamiltonian cycle, so remove another edge. Continue this process until all that remains is the Hamiltonian cycle we are looking for.

Problem 46:
Suppose there existed a problem described by a language L which satisfied the following properties:

- $L \in \mathsf{NP}$
- For all $L' \in \mathsf{NP}$, $L' \leq_p L$

Now, suppose through hard work and determination you discovered an efficient procedure for solving L. In other words, you proved that $L \in \mathsf{P}$. What would this imply about whether or not $\mathsf{P} = \mathsf{NP}$?

Solution

This would imply that $\mathsf{P} = \mathsf{NP}$ because given any $L' \in \mathsf{NP}$, we can solve it in polynomial time by efficiently reducing it to L and then using our efficient procedure to solve that instance of L.

Definition 47:
Believe it or not, such a language actually exists. In fact many of them do, and we call such languages NP-Complete. The requirements are exactly as stated: an NP-Complete language L is one such that $L \in \mathsf{NP}$ and for all $L' \in \mathsf{NP}$, $L' \leq_p L$. As a matter of fact, we have already been introduced to two NP-Complete problems: HAMPATH and HAMCYCLE. In order to prove that another problem is NP-Complete, the standard method is to show that it is in NP, and then show that it is polynomial-time reducible to some other NP-Complete problem.

Definition 48:
A directed weighted graph $G = (V, E, \omega)$ is a directed graph such that each edge $e \in E$ has an associated weight $\omega(e) \in \mathbb{R}$. The Traveling Salesman Problem asks the following question: can we visit each node of G in such a way that the total weight along our path is at most k? Formally, $\mathsf{TSP} = \{G = ((V, E, \omega), C) : G \text{ contains a Hamiltonian path } (v_0, \ldots, v_n) \text{ of total weight } w \leq C\}$.

Problem 49:
Prove that TSP is NP-Complete. You may assume that HAMPATH and HAMCYCLE are NP-Complete.

Solution

Clearly $\mathsf{TSP} \in \mathsf{NP}$ since given a solution, we can easily check that it traverses every node and that the sum of the weights on the edges is at most C. To show that it is NP-Complete, we can show that an efficient procedure for solving TSP can be used as a subroutine to solve HAMPATH. Given a graph $G = (V, E)$, we construct a new graph $G' = (V, E' = V \times V)$ which has an edge between every pair of vertices. We weight the edges according to the following rule: for $e \in E'$, if $e \in E$, then $\omega(e) = 0$. Otherwise, if $e \notin E'$, then $\omega(e) = 1$. We now use our efficient TSP procedure on G' with cost bound $C = 0$. If G contained a Hamiltonian path, then G' will contain a Hamiltonian path with cost 0, and our TSP procedure will accept. Otherwise, the minimum weight Hamiltonian path in G' will have weight at least 1, and our TSP procedure will reject.