Introduction to the Theory of Computation
Homework #5 Solutions

1. Let G be the graph with 8 vertices and 12 edges formed by the edges of a cube. How many Hamiltonian cycles does G have?

Answer: 6, unless you count the two possible orientations of each path separately, in which case there are 12.

2. (Exercise 7.8) Show that primality is in P if we give the input in unary rather than binary. In other words, show that $L = \{1^p : p \text{ is prime}\}$ is in P.

Answer: Just divide p by every integer from 2 to $p-1$ (or even just up to $\sqrt{p}$) and reject if any of them divide p evenly. This algorithm runs in poly(p) time; in fact, it runs in $O\left(\sqrt{p}\log^2 p\right)$ time, since we can easily divide one n-bit number by another in $O(n^2)$ time and $n = O(\log p)$. Thus L is in p.

Of course, this does not mean that primality is in P if we give the input in binary, which is what we would normally mean if we say that primality is in P. For this we need an algorithm that runs in time poly(n) = poly($\log p$), i.e. polynomial as a function of the number of digits of p rather than of p itself. Such an algorithm was discovered just last year.

3. (Exercise 7.11) Two graphs G and H are *isomorphic* if there is a permutation π of their vertices that makes G and H the same. Formally, two vertices u and v are connected in G if and only if $\pi(u)$ and $\pi(v)$ are connected in H. Prove that the property Graph Isomorphism (that is, the language $\{(G, H) : G \text{ and } H \text{ are isomorphic}\}$) is in NP. What about Graph Non-Isomorphism?

Answer: We just need to show that we can check that G and H are isomorphic in polynomial time given the right certificate. But this follows directly from the definition: the certificate is the permutation π, which is a list of n numbers from 1 to n. We first check that G and H have the same number of vertices and edges, and then check, for every edge of G, that π maps it to an edge of H. Since a graph with n vertices has at most n^2 edges this takes polynomial time. (We also need to check that π is indeed a permutation; we can do this in polynomial time by using an array to make sure that no numbers are repeated.)

Graph Non-Isomorphism, on the other hand, doesn't seem to be in NP; let me know if you see a kind of certificate that lets us check in polynomial time that G and H are non-isomorphic! Of course, we can check that a given permutation doesn't work, but I don't see how to check all $n!$ permutations in polynomial time.

4. (Problem 7.13) Recall that the star operation on a language gives $L^* = \cup_{i \geq 0} L^i$ where L^i is the concatenation of L with itself i times. Show that P is closed under this operation: that is, if $L \in$ P, then $L^* \in$ P. The tricky part is that if I give you an input in L^*, you don't know where the boundaries between words in L are. Hint: build a table of which substrings of the input are in L, and then use dynamic programming.

Answer: Given a word w of length n, let $L(i,j)$ be a Boolean variable which is true if the substring $w_i \cdots w_j$ is in L where $1 \leq i \leq j \leq n$. There are only $O(n^2)$ substrings, so we can calculate all the $L(i,j)$ in polynomial time.

Now let $L^*(i,j)$ be a Boolean variable which is true if $w_i \cdots w_j$ is in L^*. Our goal is to calculate $L^*(1,n)$, i.e. whether the entire input w is in L^*. We do this with dynamic programming, starting with the substrings of length 1 and building our way up. At each stage, a given substring is in L^* either if it is in L, or if it can be broken into two substrings, both of which are in L^*.

First, initialize $L^*(i,j) =$ false for all i,j. Then do:

```
for ℓ = 1 to n {
    for i = 1 to n − ℓ + 1 {
        j = i + ℓ − 1
        if L(i, j) then L*(i, j) = true
        else for k = i to j − 1
                if (L*(i, k) and L*(k + 1, j)) then L*(i, j) = true
    }
}
if L*(1, n), accept; else reject
```

This is just like the $O(n^3)$ algorithm we discussed in class to tell if a word is in a given context-free language. Note that a recursive top-down algorithm like

```
bool check(i, j) {
    if L(i, j) return true
    else for k = i to j − 1
            if (check(i, k) and check(k + 1, j)) return true
    return false
}
```

takes exponential time, since it checks the same substring many times.

5. (Problem 7.14) Show that if $L \in$ NP, then $L^* \in$ NP. Hint: this is easier than the previous problem.

Answer: In NP we can guess the boundaries between words in L, i.e. a list of integers $k_1, k_2, \ldots, k_t$ where we break the input w into substrings. We can then check in polynomial time that the substrings $w_1 \cdots w_{k_1}$, $w_{k_1+1} \cdots w_{k_2}$, $\ldots$, $w_{k_t+1} \cdots w_j$ are all in L.

6. (Problem 7.16) Consider two problems, Short Path and Long Path. Both take input (G, u, v, k) where G is a graph, u and v are vertices, and k is an integer. Short Path asks

whether there is a path in G from u to v of length *at most* k, and Long Path asks whether there is a path of length *at least* k. In both cases paths are not allowed to visit the same vertex twice.

Show that Short Path is in P, but Long Path is NP-complete.

Answer: Short Path is in P because polynomial-time algorithms exist (e.g. Dijkstra's) that computes the shortest path between u and v. Long Path is NP-complete because we can trivially reduce Hamiltonian Path to it: namely, to find if there is a Hamiltonian path, just ask if there is a path of length at least $n - 1$.

7. (Problem 7.22) Recall that Not-All-Equal Satisfiability (NAE-SAT) is like SAT, but where we demand that every clause contains at least one true literal and at least one false one. NAE-3-SAT is the special case where each clause has exactly 3 literals.

Show that NAE-3-SAT is NP-complete by reducing 3-SAT to it. Hint: it might be easier to first reduce 3-SAT to NAE-k-SAT for some $k > 3$ (i.e. allow longer clauses) and then show how to break up these clauses into 3 literals each. Use only variables (i.e. don't use constants like T or F).

Answer: One way to do this is to add a single dummy variable b to every clause, turning the 3-SAT clause $(x \vee y \vee z)$ to the NAE-4-SAT clause (x, y, z, b). If b is false, at least one of x, y, and z must be true; if b is true, then at least one must be false. The latter case is equivalent to flipping every variable in the original 3-SAT formula, and the resulting formula is satisfiable if and only if the original formula is. Thus the choice of b "breaks the symmetry" between true and false, and sets a compass by which the rest of the variables can be judged. (Note the similarity to Graph 3-Colorability, where the choice of color for a single vertex determines which two colors will represent the variables' values.)

Another (much less efficient) reduction is to represent each variable x with two variables x_1 and x_2, where $x = x_1 \oplus x_2$ ($\oplus$ is a common symbol for XOR). That is, x is true if $x_1 \neq x_2$ and false if $x_1 = x_2$. Note that this definition of truth is preserved if we flip all the variables. Then the 3-SAT clause $(x, \overline{y}, z)$, for instance, becomes the four NAE-6-SAT clauses:

$$(x_1, x_2, y_1, \overline{y_2}, z_1, z_2)$$
$$(x_1, x_2, y_1, \overline{y_2}, \overline{z_1}, \overline{z_2})$$
$$(x_1, x_2, \overline{y_1}, y_2, z_1, z_2)$$
$$(x_1, x_2, \overline{y_1}, y_2, \overline{z_1}, \overline{z_2})$$

A little reflection will show that the only way for all four clauses to contain at least one true literal and at least one false one is if $x_1 \neq x_2$, $y_1 = y_2$, or $z_1 \neq z_2$: that is, if x is true, or y is false, or z is true.

To reduce from NAE-4-SAT (or NAE-6-SAT) to NAE-3-SAT, we use exactly the same trick we used to reduce CNF-SAT to 3-SAT. We break clauses into chains of 3-clauses, connected by dummy variables: for instance (x, y, z, b) becomes $(x, y, t) \wedge (\overline{t}, z, b)$.

8. (Problem 7.23) Given a graph G with vertices V, a *cut* is a subset $S \subset V$. The size of the cut is the number of edges with one end in S and the other end in $\overline{S}$. MAX-CUT is the following problem: given a graph G and a number k, does G have a cut of size k or more?

Show that MAX-CUT is NP-complete by reducing NAE-3-SAT to it. Hint: if the NAE-3-SAT formula has k clauses, represent each variable x with $3k$ vertices labelled x and another $3k$ vertices labelled $\overline{x}$, with $(3k)^2$ edges connecting all the x vertices to all the $\overline{x}$ vertices. Then represent each clause with a triangle connecting an appropriate triple of vertices. Figure out how large the cut corresponding to a solution to the NAE-3-SAT formula is, and prove that a cut of this size only exists if the formula is satisfiable.

Answer: If there is an NAE-satisfying assignment, then two out of three edges of every triangle will be in the cut. If the formula has n variables and k clauses, the size of the cut is then $(3k)^2 n + 2k$.

Conversely, suppose a cut of this size exists. We first need to prove that for every variable x, all the x vertices are in S and all the $\overline{x}$ vertices are in $\overline{S}$, or vice versa. If not, i.e. if S or $\overline{S}$ contains two vertices of opposite type, then at least $3k$ of the $(3k)^2$ edges among the x and $\overline{x}$ vertices are not in the cut. (Do you see how to prove this? Hint: no loop can contain an odd number of edges in the cut.) Then, even if we could magically put all three edges of each clause triangle in the cut, the size of the cut would still be only at most $(3k)^2 n - 3k + 3k = (3k)^2 n < (3k)^2 n + 2k$.

Thus S contains either all x vertices or all $\overline{x}$ vertices, representing a consistent choice of x's truth value, and that it contains all $(3k)^2 n$ edges connecting vertices of opposite type. To get the remaining $2k$ vertices, we need to get two out of three vertices of each clause triangle, since we can't have all three. This corresponds to setting one literal in each clause different from the other two, so such a cut corresponds to solving the NAE-3-SAT formula.

9. (Problem 7.27) A *hypergraph* is a set of vertices V, and a set of "edges" E, where each edge is a non-empty subset of V. (A normal graph is one where every edge has size 2.) A hypergraph is 2-colorable if there is a way to color the vertices black and white so that every edge contains at least one black vertex and at least one white vertex.

Show that Hypergraph 2-Colorability is NP-complete. Hint: this is just like NAE-3-SAT, except 1) the "clauses" can have any number of variables, and 2) the variables are never negated. You might want to represent each variable with a pair of vertices.

Answer: Represent each variable x with a pair of vertices x and $\overline{x}$, and include the pair $(x, \overline{x})$ in the list of edges E. Since every edge must contain both colors, one of these will be white and the other black. We use this to choose whether x is true or false.

Then, for each NAE-clause such as $(x, \overline{y}, z)$, we add a hyper-edge to E, namely the triplets $(x, \overline{y}, z)$. Since this must contain both colors, we force at least one of these literals to be true and one to be false.

10. (Problem 7.33) Point out the fallacy in the following "proof" that P $\neq$ NP: "To see if a 3-SAT formula is satisfiable, we need to look at 2^n possible truth assignments. This takes exponential time, so 3-SAT is not in P. But it is in NP, so P $\neq$ NP."

Answer: Just because one algorithm takes exponential time doesn't mean that there isn't a faster one! There may indeed be a much smarter way of telling whether a 3-SAT formula is satisfiable than doing an exhaustive search on all its truth assignments. P $\neq$ NP only holds if *all* algorithms for 3-SAT take more than polynomial time.