Colleen Minor
HW4
Analysis of Algorithms

1. *Let X and Y be two decision problems. Suppose we know that X reduces to Y in polynomial time. Which of the following can we infer? Explain*

a) If Y is NP-complete, then so is X

False. X reduces to Y, which means that it is *no harder* than Y, but it doesn't mean that X is *as hard* as Y, so it might not be NP-complete.

b) if X is NP-complete, then so is Y

False. We know by the fact that X is NP-complete and that it reduces to Y that Y is in NP-Hard, but would still need to prove a solution can be verified in poly time.

c) if Y is NP-complete and X is in NP then X is NP-complete

False. All NP problems can be reduced to any NP-complete problem, but the reverse is not true, so the fact that Y is NP-complete does not mean that X also is.

d) if X is NP-complete and Y is in NP then Y is NP-complete

True. We see that Y is in NP (solution is verified in poly time), and it must be at least as hard as X (which is NP-hard by part of the NP-complete criteria), making is NP-complete.

e) X and Y can't both be NP-complete

False. All NP-complete problems can be reduced to each other, so X being NP-complete and reducing to Y doesn't mean that Y can't also be.

f) If X is in P, then Y is in P

False. X reduces to Y, which means that it is *no harder* than Y, but how hard X is doesn't say anything about how hard Y is, except that it is not easier than X.

g) If Y is in P, then X is in P

True. Because X reduces to Y, it is not harder than Y, and Y, being in P, can be solved in polynomial time. That means that X can be too, making it also $\in$ P.

2. *Consider the problem COMPOSITE: given an integer y, does y have any factors other than one and itself? For this exercise, you may assume that COMPOSITE is in NP, and you will be comparing it to the well-known NP-complete problem SUBSET-SUM: given a set S of n integers and an integer target t, is there a subset of S whose sum is exactly t?*

Clearly explain whether or not each of the following statements follows from that fact that COMPOSITE is in NP and SUBSET-SUM is NP-complete:

a. SUBSET-SUM ≤p COMPOSITE.

No. We know that SUBSET-SUM is NP-complete, but composite might not be. We only know that it is NP, but if it is not also NP-hard, it is not NP-complete, so we cannot say that SUBSET-SUM will reduce to COMPOSITE.

b. If there is an O(n^3) algorithm for SUBSET-SUM, then there is a polynomial time algorithm for COMPOSITE.

Yes. In theory, if any NP-complete problem can be solved in polynomial time, so can any NP-problem, so this is true.

c. If there is a polynomial algorithm for COMPOSITE, then P = NP.

No. I believe that the idea is that if a single *NP-complete* problem has a polynomial-time solution, then all NP problems do, but this is not the case with solving an NP problem which is not NP-complete. We only know COMPOSITE to be in NP, but not in NP-complete.

*d. If P != NP, then **no** problem in NP can be solved in polynomial time.*

No. All of the problems in P, which is the set of problems which can be solved in polynomial time, are also in NP; and NP problems are all the problems that have a verifiable proof, regardless of whether they can be solved efficiently. Thus if P != NP, then there exist problems in NP that can be verified in polynomial time, but not solvable in polynomial time.

3. *Two well-known NP-complete problems are 3-SAT and TSP, the traveling salesman problem. The 2-SAT problem is a SAT variant in which each clause contains at most two literals. 2-SAT is known to have a polynomial-time algorithm. Is each of the following statements true or false? Justify your answer.*

a. 3-SAT ≤p TSP.

True. It is stated that 3-SAT and TSP are both NP-complete problems, and, extrapolating from the definition that “any NP problem can be reduced to an NP-complete problem in polynomial time" (by logic of NP-problems being both NP-hard, meaning that any NP problem can be reduced to them, and NP-complete problems also being in NP).

b. If P != NP, then 3-SAT ≤p 2-SAT.

False. If P != NP, than 3-SAT, by being in NP-hard (since it is an NP-complete problem), does not have a polynomial-time algorithm. Since 2-SAT is known to have a polynomial-time algorithm, we know that it is in p. If 3-SAT reduced to 2-SAT, there would need to be a polynomial-time algorithm for 3-SAT. But 3-SAT being in NP-hard (meaning that it is as hard to solve as the hardest NP problems, which, if they are distinct from P problems, cannot be solved in poly time), cannot, thus showing that 3-SAT cannot be reduced to 2-SAT if P != NP.

c. If P != NP, then no NP-complete problem can be solved in polynomial time.

True. I believe that the distinction between P and NP (assuming they are different) is that NP problems cannot be solved in polynomial time, and since NP-complete problems are in NP-hard, they cannot be solved in polynomial time.

(Different wording for personal notes): True. I believe that the distinction between P and NP (assuming they are different) is that NP problems which are not in P cannot be solved in polynomial time, and since NP-complete problems are as hard to solve as the hardest problems in NP (since they are NP hard problems, which are problems that all problems in NP can be reduced to) they cannot be solved in polynomial time.

***4.** LONG-PATH is the problem of, given (G, u, v, k) where G is a graph, u and v vertices and k an integer, determining if there is a simple path in G from u to v of length at least k. Show that LONG-PATH is NP-complete.*

First, we will show that LONG-PATH is in NP. If LONG-PATH is in NP, that means that it's answer can be verified in poly steps. A certificate of this problem is the sequence of edges and vertices in the path. We can check in polynomial time that each vertex/edge is only in the path once, and that the length of the edges add up to a length of at least k, by iterating through the the path once in O(|G|) time.

We also must prove that this problem is NP-hard. To do that we will show that the known np-complete problem called HAMPATH(G,u,v k) ≤p LONG-PATH(G', u', v' k'). Given a SUBSET-SUM(G, u, v, k) instance, we can construct a LONG-PATH(G', u', v', k') instance as follows. The SUBSET-SUM list G contains the numbers to be summed, and u and v are the . For each $g_i \in$ G, we add an edge, $g_{i,}$ from v to u with weight g_i. We set the target sum of LONG-PATH, k', to the target sum of SUBSET-SUM, k.

***4.** LONG-PATH is the problem of, given (G, u, v, k) where G is a graph, u and v vertices and k an integer, determining if there is a simple path in G from u to v of length at least k. Show that LONG-PATH is NP-complete.*

We will now show that LONG-PATH is NP-complete, which means that it is both NP and NP-hard.

First, we will show that LONG-PATH is in NP. If LONG-PATH is in NP, that means that it's answer can be verified in poly steps. A certificate of this problem is the sequence of edges and vertices in the path. We can check in polynomial time that each vertex/edge is only in the path once, and that the length of the edges add up to a length of at least k, by iterating through the the path once in O(|G|) time.

We also must prove that this problem is NP-hard. To do that we will show that the known np-complete problem called SUBSET-SUM(G, k) ≤p LONG-PATH(G', k'). Given a SUBSET-SUM(G, k) instance, we can construct a LONG-PATH(G', k') instance as follows. The SUBSET-SUM list G contains a list of numbers to be summed, k is the target number, and the output is *u and v*, which are the starting and ending indexes of the numbers in G that sum to at least k . We make the graph G' by using each $g_i \in G$ between indexes u and v, as the vertices, and we use the numerical value of each g_i as the length of the edge between each g'_i vertex and the next vertex in the path. This makes the graph G' a list of vertices that are traversed, and the edge lengths between them. We set the target sum of LONG-PATH, k', to the target sum of SUBSET-SUM, k.

i. Show that if G, k is a "yes" solution to SUBSET-SUM, then G', u, v, k' is a "yes" solution to LONG-PATH:

If (G, k) ∈ SUBSET-SUM, then there exists a subset of elements $\{g_{i1}…. g_{in}\}$ that sum to at least k. The path in G' is the vertices between u and v, $\{g'_{i1}…g'_{in}\}$, traversing each vertices exactly once. This is a simple path that will have a length >= k, because we assigned the lengths of the edges between each vertex as the numerical values of each index between the u and v that SUBSET-SUM found summed to >= k.

ii. Show that if G', u, v, k' is a "yes" solution to LONG-PATH, G, k is a "yes" solution to SUBSET-SUM:

Say that there exists a simple path, p, in G' with length k. The sequence of vertices in this path is $\{g'_{i1}…g'_{in}\}$. Let G be the list of $\{g_{i1}…g_{in}\}$ corresponding to the lengths of the edges of p. For each i in $\{g'_{i1}…g'_{in}\}$, there is a corresponding element $g_i \in G$, who's value is the length of the edge between g'_i and $g'_{i+1.}$ Obviously, the sum of the values in G will be k, because we are adding up a list of the same numbers. Therefor, (G, k) ∈ SUBSET-SUM.

5. *Graph-Coloring. Mapmakers try to use as few colors as possible when coloring countries on a map, as long as no two countries that share a border have the same color. We can model this problem with an undirected graph G = (V,E) in which each vertex represents a country and vertices whose respective countries share a border are adjacent. A k-coloring is a function c: V -> {1, 2, … , k} such that c(u) != c(v) for every edge (u,v) ∈ E. In other words the number 1, 2, .., k represent the k colors and adjacent vertices must have different colors. The graph-coloring problem is to determine the minimum number of colors needed to color a given graph.*

a. *State the graph-coloring problem as a decision problem K-COLOR. Show that your decision problem is solvable in polynomial time if and only of the graph-coloring problem is solvable in polynomial time.*

Stated as a decision problem:
K-COLOR :
input: *G=(V,E)*, an undirected graph where each vertex represents a country and vertices whose respective countries share a border are adjacent; *k* maximum the number of colors we want to use.
output: *True* if we can use k colors to color each vertex in V such that each vertex is colored, and no vertex will be the same color as that of its adjacent vertices…. otherwise, *False*.

Why this is solvable in poly time:
This is only solvable in polynomial time if the graph-coloring problem is by effect of the fact that the graph-coloring problem is a sub-routine of K-COLOR. In K-COLOR, once we get the minimum number of colors that we need by sending our graph into the graph-coloring problem, we can just compare that number of colors to the number of vertices in the graph to tell you if the answer is yes or no.

Obviously if graphColoringProb can run in poly time, so can K-COLOR, because all we have to do is call it once to retrieve the minimum number of colors needed, and then make one comparison of that number to k, and that one comparison runs in O(1) time.

b. It has been proven that 3-COLOR is NP-complete by using a reduction from SAT. Use the fact that 3-COLOR is NP-complete to prove that 4-COLOR is NP-complete.

Step 1) Show that 4-COLOR belongs to NP:
Given an instance of the problem, the certificate is the coloring (a list of vertices and colors) in the graph. A certifier (verification algorithm) could work by checking that no more than four colors are used, this list contains each vertex exactly once, and that all vertices that share an edge with whatever vertex is being checked are of a different color from that vertex.

Step 2) Show that 4-COLOR belongs to NP-Complete by showing that 3-COLOR $\leq_p$ FOUR-COLOR where 3-COLOR $\in$ NP-complete:

Give a graph G $\in$ 3-COLOR, we can map G to a graph, G', and then to G' we add one more vertex which is attached to every other vertex in G'.

Because G is 3-colorable, G' must be 4-colorable, with the additional color coloring only the vertex that shares an edge with every other vertex. The fact that this extra vertex touches every other vertex means that it cannot be colored with any of those 3 colors, and we must use the additional color.

This reduction takes linear time to add a single node and G and edges.