Problem Set 6

All parts are due Friday, October 25 at 6PM.

Name: Lydia Yu

Collaborators: Pranit Nanda, Isaac Redlon, David Magrefty

Problem 6-1.

(a) 4,2,1,0,3,5

The number of distinct topological orderings is 8. This is because there are three nodes with no outbound edges (5,3,1) that can all be the last node in the ordering and also interchanged with each other. 4 must always be the first node in the ordering because it is the only one with no inbound edges. We also know that 0 and 1 are interchangeable (to a certain extent that will be explained later) because they are not connected to each other and each have an inbound edge coming from the same node but no outbound edge that goes to the same node.

First, find the orderings with 5 as the last item. There are two different orderings for this cause since 1 and 0 are interchangeable. Then, for each of these 2 orderings, switch 5 and 3 so that 3 is now the last item in the ordering. Now we have 2*2=4 orderings. Next, we can make swaps in those 4 orderings of 1 and 3, but leave out the orderings where 3 ends up before 0 because 3 must come after 0 in the graph; this adds 2 to the total (since there were two of those orderings where 1 and 0 were swapped such that 1 comes before 0. 1 and 0 are not interchangeable if we end up with an ordering where a node connected to 0's outbound edge ends up before 0). Finally, swap 1 and 5 in those 4 orderings, but leave out the orderings where 5 ends up before 0 because 5 must come after 0 in the graph. This adds another 2 for a total of 8 different orderings.

(b) Add an edge from 3 to 2.

The number of distinct single edges that could be added to G to construct a simple graph with no topological ordering is just the number of nodes that have an edge going in and an edge going out of them, which is 3 in this graph, plus the number of existing edges in the graph, which is 7, plus the one cycle that can be formed with 4 nodes (by adding an edge from 5 to 4), for a total of 11 edges. This is because a simple graph with no topological ordering is just a simple graph with cycle(s) since you can't have a topological ordering if there is a cycle. If a node already has an inbound and outbound edge going to it, then that means that it is only one edge away from contributing to a cycle in the graph. To create the cycle of 3 nodes, just connect

the node that the original node's outbound edge is pointing to with a directed edge that points to the node that the original node's inbound edge is coming from, which will complete the cycle. Then, if there is already an existing edge between two nodes, a cycle of 2 nodes can be created by adding an edge pointing in the reversed direction.

Problem 6-2.

(a) When G is a tree, this means that there is only one possible path from any node to any other node in the graph (connected and acyclic graph). Having only one path from one node to another is what allows the DFS Relaxation algorithm to correctly compute the shortest path distances. If G were not a tree and instead had nodes that more than one path to other nodes in the graph (if cycles existed), the first path that the DFS algorithm finds to a certain node may not be the actual shortest path because there may be another path that is shorter that DFS does not cover. In a tree, however, there is only one path from each node to another, so whatever path the DFS algorithm finds must also be the shortest because there is no other path that exists.

(b) An undirected graph where the DFS Relaxation incorrectly computes to shortest path is one where the three vertices form a connected cycle (each vertex is connected to the other two).

(c) A possible undirected graph with k vertices that fails spectacularly is one where all k vertices are connected in a circle. If the vertices are connected in a circle, then the node next to the starting node that is the last one traversed in the DFS will end up with a d' value of k-1 even though its actual distance is 1; k-1 is always going to be greater than half the number of nodes times the actual distance when the number of nodes is greater than 3.

Problem 6-3. Create a directed graph where the vertices are landmarks, the edges are trails connecting the landmarks which will point in the direction from the landmark that's closer to the gorge to the landmark that's farther away (only include an edge if it goes to a landmark that's farther away because Bimsa wants to never return), and the weights on the edges are given by the length of the trail. We know that this graph will have no cycles because of the condition that Bimsa never returns as she traverses the trails: she will always be going to landmarks that are strictly farther away from the gorge than the current landmark she's at, so she will never end up at a landmark that she already visited. We will calculate a value that represents the euclidean distance a landmark is from the gorge by using the euclidean distance formula but without the square root step (just take the sum of the squares). This can be done in constant time and results in integers so that we don't have to end up with the complexity of dealing with floating point numbers, and still yields values that are proportional to the actual euclidean distances.

Run DAGSP on this acyclic graph starting at the gorge node. The algorithm can stop running as soon as a landmark that is outside of Honor Stone. DAGSP will give us the shortest path to this landmark outside of Honor Stone in O(n) time because DAGSP takes O($|V| + |E|$); in this case, we know that each landmark connects to 5 or less trails and that the graph is connected because every landmark can be reached through trails from the gorge, so $|E|$ is a constant multiple of $|V|$ and we get O($|V|$) $\rightarrow$ O(n) since there are n landmarks in Honor Stone, and we only go to one of the non-Honor Stone landmarks.

Problem 6-4. Create a directed graph with n nodes that represent the block types. The edges connect the different block types and represent a conversion from one block type to another. For each edge connecting block type b_i to block type b_j ($i, j \in 1...n, i \neq j$), it is weighted by the log of the ratio of the number of blocks of type b_j to the number of blocks of type b_i: $\log_2(b_j/b_i)$. Taking the log of the ratios allows us to add up the conversion factors in any set of conversions instead of needing to take the product of the blocks produced in the conversion because the log of the ratios is proportional to the product of numbers comprising the ratio. We know that we will have integers for the edge weights because all of the d_i are powers of 2. Also, since there are D blocks of each type where D is the product of all the d_i, whenever we make a conversion from one block to another by dividing by a d_i, we know that we will still end up with a number that is greater than 1 that makes more conversions possible. We multiply all of the weights by -1 so that we can find an infinite negative cycle. There are $\lfloor 1/5n^2 \rfloor$ edges total because we are looking at a set of $\lfloor 1/5n^2 \rfloor$ conversions.

We will also create a supernode that to all of the vertices in the graph (this takes O(n) to do since there are n vertices). This ensures that the graph is connected because if it were not, running a shortest-path algorithm from one node may not find a negative cycle if that cycle occurs in nodes that are not connected to the vertex that the algorithm started at. The weights of the edges coming from the supernode are 0. Finally, we run the bellman-ford algorithm starting from the supernode to determine if we end up with an infinite negative cycle. This takes O($|V||E|$) = O($n \cdot 1/5n^2$) $\rightarrow$ O(n^3).

Problem 6-5. Create an undirected graph with the vertices as the n towns and the edges as the roads connecting the towns. We have to generate 3 duplications of this graph since he only drinks at every 3 taverns. Each vertex has information on the town and the number of roads he's traversed since he last drank at a tavern (for example, t_{i1} means that town i is 1 road away from the last town he drank in); each town will thus have 3 different nodes representing the 3 different road traversal states he could be in when he visits that town. These vertices are connected such that towns 1 road away from the last drinking town must be connected with an edge to towns 2 roads away, towns 2 roads away must be connected to towns 3 roads away, and towns 3 roads away must be connected to towns 1 road away. This takes care of the condition that he only drinks at taverns every 3 towns.

The weight of each edge is the tax he collects by traversing along that road. If the edge is going from a town 2 roads away from a drinking town to a town 3 roads away, then the weight is the tax along that road minus the amount spent on drinks at the town 3 roads away because we know that he will lose gold every three towns he visits from drinking at the tavern. We will also multiply each of these weights by -1 so that we can find the path with the maximum amount of gold (shortest path algorithm will return the path with the most negative total weight), which will take O($|E|$).

We run the bellman-ford algorithm starting from the starting town Prince's Pier to the ending town Summerrise, assuming that he doesn't drink at the starting town (if he does, subtract the amount of gold he spends there from his total amount of gold), to find the shortest path, which in this case is the path with the highest absolute value of gold. The total amount of gold he has is equal to the sum of the weights of the roads he traverses along the found path times -1 (this number will be negative if he is in debt). If the algorithm finds an infinite cycle, then there is a path in the map that results in an infinite amount of money that he can collect. This takes O($|V||E|$); since each town is connected to at most 7 roads, we know that $|E|$ is a constant multiple of $|V|$, so we get a running-time of O(n^2).

Problem 6-6.

(a) Create a directed graph with the vertices as the code pairs and the edges as the dependencies in D: for each dependency in D (f_i, f_j), connect a directed edge from the code pair with filename f_i to the code pair with filename f_j for a total of $|D|$ edges. The weights of the edges are given by the time it takes the first filename in the dependency pair to be completed. We know that a job cannot be completed if there is a cycle in this graph because we will not know if a code pair in the cycle should be completed before or after another code pair in the cycle (two files cannot have dependencies going in both directions from one file to the other). We will also want to create a supernode that connects to all nodes in the graph without an incoming edge (all code pairs with filenames that appear as the first filename in any of the dependency pairs in D but never as the second filename; the code with a filename like this is only ever completed first and never completed after another piece of code) because we want the graph to be connected. The weights of the edges coming from the supernode are 0. Adding the supernode adds a maximum of $|C|$ edges to the graph (if all of the code pairs' files are not dependent on each other).

Run DFS starting from the supernode. If DFS runs to completion, then that means that there was no cycle in the graph and that the job can be completed because all of the dependencies are valid for the given code pairs. However, if DFS does not run to completion, then we know that there was a cycle in the graph and the job cannot be completed because the algorithm reached a group of code pairs where it doesn't know which code to complete first. We can know when a cycle occurs by keeping track of a list of nodes that we have already visited as we run DFS; if DFS tries to go to a node that is already in the list of visited nodes, then we know that there is a cycle. This takes O($|E| + |V|$)), but we know that the graph is connected so $|V| = |C| = |E| + 1$. Since the number of edges in the graph is on the order of O($|D|$), we know that this algorithm will run in O($|D|$).

(b) We will use the algorithm from part a) with a few modifications. First, we run what we have from part a) to check for cycles in the graph; if there is a cycle, we return that job can't be completed, but if there isn't a cycle and the DFS runs to completion, then we can move on. This algorithm from part a) will also allow us to obtain the topological order of the vertices, and using this information, we can later run DAGSP. We will modify our graph from part a) to include an end node to which all of the vertices that do not have outgoing edges (in other words, the code pairs with filenames that appear as the second filename in any of the dependency pairs but never as the first filename; code with these filenames is always completed after other code and never before other code) so that we have the specific node at which we want the algorithm to stop at that will be the end of the path and ensures that all jobs have been completed when we get to this end node. The weight of these edges to the end node will be the time it takes to complete the code from which the edge extends. We will have a maximum of $|C|$ edges added to the graph for the same logic as in part a) (if we have

only have filenames that are not dependent on each other, then we will add edges from every existing vertex to the end node), which is a constant multiple of $|D|$ because each dependency connects two filenames together. We will also multiply all of the edge weights by -1 because we want to find the minimum amount of time required to complete the job, which means that at each level of the graph, we want to travel down the path with a higher time and the shortest path algorithm will take the smaller (more negative) of the possible paths. This is necessary because if we did not multiply by -1 and let the algorithm find the shortest path, it would choose the edge at each level with the smaller completion time, which is not representative of how much time is required for the entire job to be completed because every code pair in that job must be completed for the job to be completed. This step takes O($|E|$), which we know is on the same order as O($|D|$) since the graph is connected and $|D|$ is the number of dependencies, because each multiplication is a constant operation.

Use the DAGSP algorithm to relax all of the edges in the topological order we found from the DFS from part a) and get the shortest distance from the supernode to the end node (which, in this case, represents the minimum time required to complete the job because the weights are negated so that the minimum path length that the algorithm finds is actually the sum of the longest time required at each level of the graph to complete a code). Essentially, we want to find the shortest path of the longest times it takes to complete the jobs at each level of the graph. In the end, we take the absolute value of the path length returned for the "shortest" path from the supernode to the end node to get the minimum time required to complete all of the code in the job. This algorithm takes O($|D|$) because we know from part a) that running DFS to check for cycles and obtain topological order takes O($|D|$), and DAGSP takes O($|V|+|E|$). The number of nodes in this graph is $|C|$, which we know from the previous section is on the same order as O($|D|$), and there are also O($|D|$) edges as shown above, so the overall running-time is O($|D|$).

(c) Submit your implementation to `alg.mit.edu`.