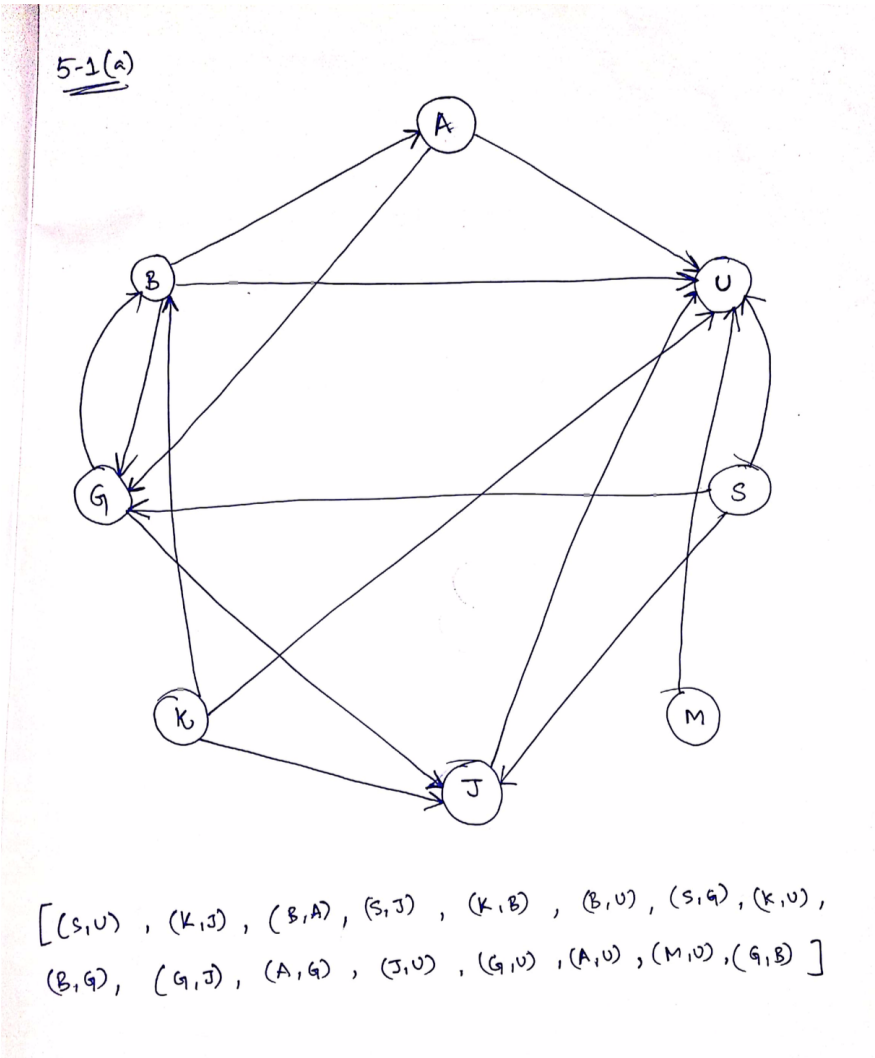
Problem Set 5

Name: Neeraj Pandey

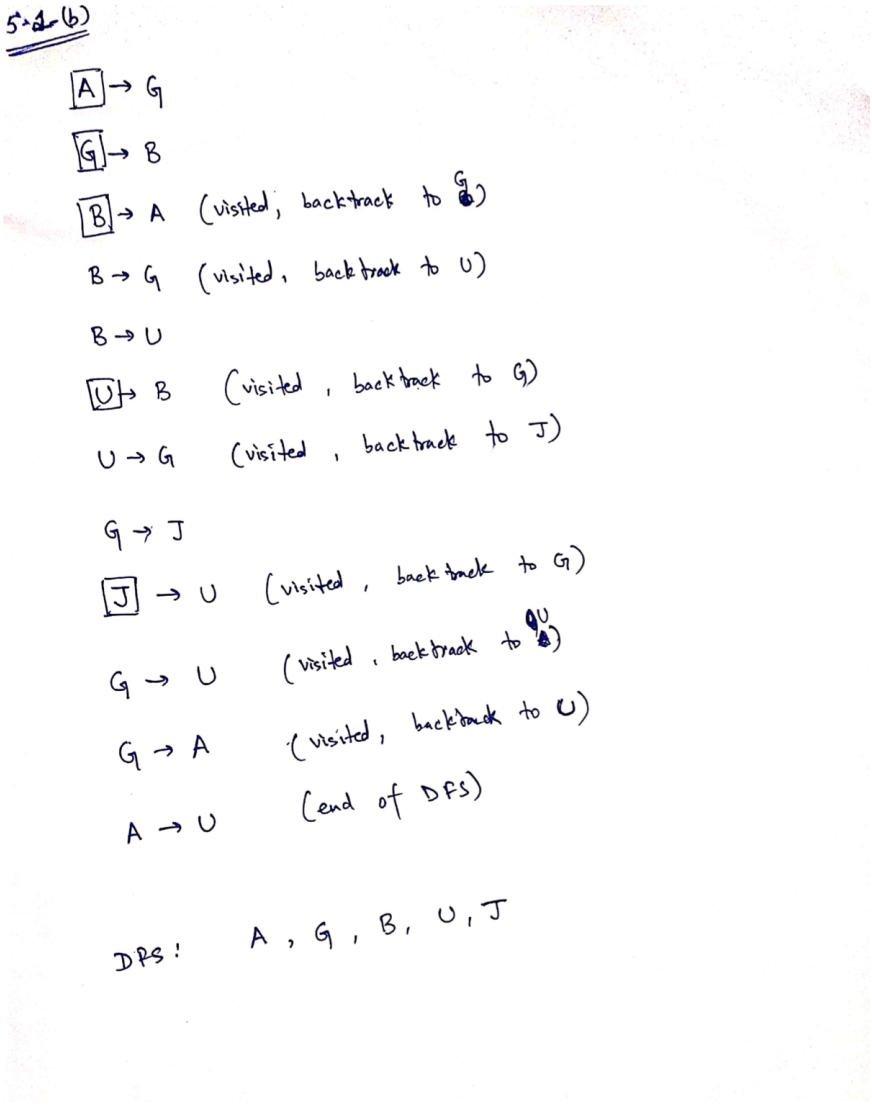
Collaborators: Raunak

Problem 5-1.

(a) Graph below:



(b) DFS: [A, G, B, U, J] in increasing order
and [J, U, B, G, A] in decreasing order As the graph is not a directed acyclic graph, topological sort cannot be performed. Therefore, returned list is not equal to the topological sort.

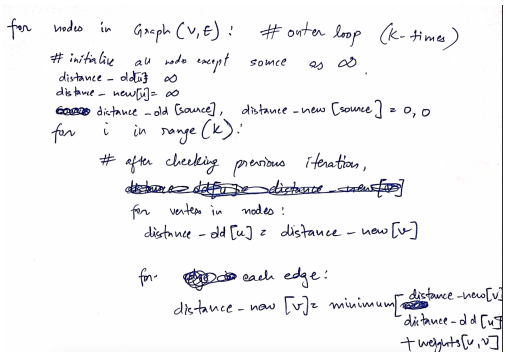


(c) Here, the list is not alphabetically sorted because the graph is a directed graph. Like A can choose B alphabetically but in this directed graph A is not directing towards B, so A cannot go to B. Also, though DFS is a topologically sorted but there is a cycle involved in the graph's reversed list. So, it is not topological.

(d)

Problem 5-2.

(a) To solve this problem, for the weighted directed graph we can run Bellman Ford Algorithm, but instead of running the algorithm for $vertices - 1$ iterations, we can run it till k times so that we can limit the Bellman-Ford such that the results computed in an iteration of the outer loop are not used to improve the distance estimates of the other vertices during the same iteration. This would provide the shortest distance for the k-edges which include some paths of edges, other than the k edges as well. For the distance calculated we keep track of the distances of other vertices in the previous iteration. So, during ith iteration, we will first check which edge needs to be relaxed but update it only after the next edges are checked and the relaxation will depend on the $i - 1$th iteration iteration. Therefore the time complexity will be O(V+KE) as initialization will take O(V) and checking and updating for k-iterations will be O(kE).



(b)

(c)

Problem 5-3.

(a) Here we will be implementing a Bellman Ford algorithm. To peform this, we first take all the nodes(the points where the token jumps along the directed edge) and make an edge list of all the adjusts. In that lists, for each pair u, v, we will relax the pairs. Here relaxation increases the accuracy of the distance of the given pair of vertices. It works by continuously shortening the calculated distance between the vertices that distance with other known distances. Also, initialize the first vertex (here the initial integer score (s)) with 0 distance and all the other node as infinity.

The relaxation equation works in the following way:

$$if(Distance[u] + weight[u, v] < Distance[v] :$$

$$Distance[v] = Distance[u] + weight[u, v]$$

We need to relax the pair of vertices (vertices - 1) times. Now, use the relaxation equation and update the distance of each vertex and repeat it for (vertices - 1) times. At the end, each node will have a shortest distance. As we want to find if the token from position (s) could reach the vertex (t) in k jumps, we can compare the shortest distance from vertex (s) to vertex (t) with k. If it's equal, then yes it's possible. In best case, the time complexity for the given algorithm will be O(VE) for V number of vertices and E edges.

(b)

(c)

(d)

Problem 5-4.

(a) Here, we will consider the condition as a directed graph where Neil trade apples to get some fruit f_b. For every transaction we normalize a and b such that Neil gets how much of source fruit f_b for he has to pay to get 1 unit of destination fruit. Now, consider the source node of the graph as an apple, and Neil has to reach from apple to avocado and want this path that is shortest so that he can get as many avocados as possible by trading minimum number of apples. Shortest path here would mean the apples Neil would trade for 1 unit of avocado finally. To perform this, Dijkstra algorithm cab be implemented but considering edge weights can be negative, Dijkstra shortest path may not work here. So, we will use Bellman Ford algorithm here. It will work by overestimating the length of the path from the source (apple here) to all the other vertices (fruits that are being offered by other students). Iterating over, we will relax those estimates by finding a new path that is shorter than the previously overestimated path by using the relaxation : The relaxation equation works in the following way:

$$if(Distance[u] + weight[u, v] < Distance[v] :$$

$$Distance[v] = Distance[u] + weight[u, v]$$

We need to relax the pair of vertices (n-1) times. Now, use the relaxation equation and update the distance of each vertex and repeat it for (n-1) times. At the end, each node will have a shortest distance. The time complexity for the given algorithm will be O(mn) as it take (n-1) relaxations for n responses Neil received and m as the weights.

(b)

(c)

(d)

Problem 5-5.

(a) Here, we have to find the shortest distance which is of course less than m. To solve this problem, we can do it using Dijkstra algorithm but the time complexity will be O(knlogn). But, we need to get the time complexity under O(kn). To do so, taking into account that the Eueclidean distance is irrelevant, we can use Bellman Ford algorithm. Take take all the nodes (the destinationss) and make a edge list of all the adjusts. In that lists, for each pair u, v, we will relax the pairs. Here relaxation increases the accuracy of the distance of the given pair of vertices. It works by continuously shortening the calculated distance between the vertices that distance with other known distances. Also, initialize the first vertex (here, the Boston node) with 0 distance and all the other node as infinity. The relaxation equation works in the following way:

$$if(Distance[u] + weight[u, v] < Distance[v] :$$

$$Distance[v] = Distance[u] + weight[u, v]$$

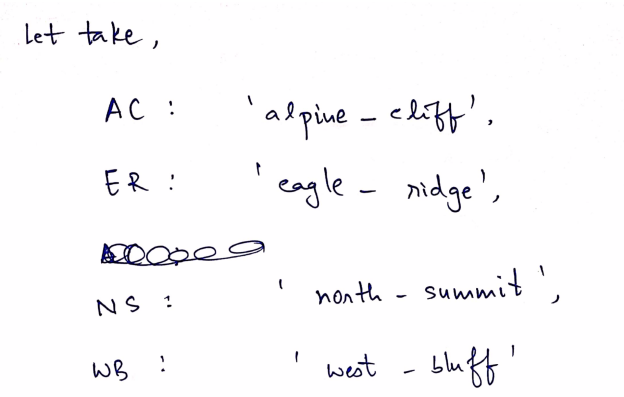
Here weight is the price of a specific route. We need to relax the pair of vertices (n-1) times. Now, use the relaxation equation and update the distance of each vertex and repeat it for (n-1) times. At the end, each node will have a shortest distance (here, the route as K). **So, applying bellman ford algrithm from source to the destination station would give us all the destination station, which would give us distances to all the intermediate stations, and check adjacent stations should have distance** $<=$ **m.** In best case, the time complexity for the given algorithm will be O(Kn).

(b)

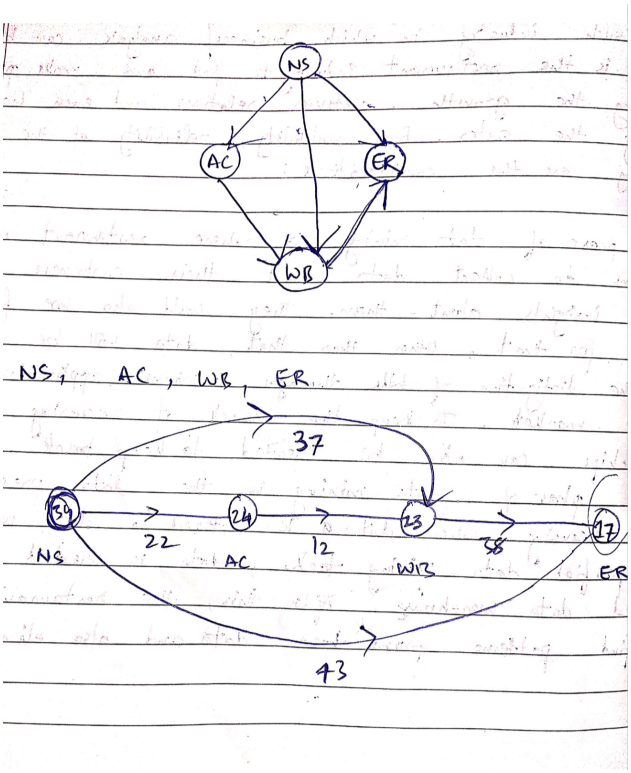
(c)

Problem 5-6.

(a) Intially,



Then, we will perform topological sort (below is the output of topological sort):



From the above, we can conclude 3 best most awesome downhill courses as:

- "north_summit" $\Longrightarrow$ "$alpine\_cliff$" $\Longrightarrow$ "$west\_bluff$" $\Longrightarrow$ "$eagle\_ridge$" : 72
  "$north\_summit$" $\Longrightarrow$ "$west\_bluff$" $\Longrightarrow$ "$eagle\_ridge$" : 75

- "north_ summit" $\implies$ "$eagle\_ridge$" : 43

(b) To compute this, we will create a graph, take each checkpoint as a node. As the graph will be a directed acyclic graph, we can compute the longest distance here using the -G transformation to shortest path of finding the longest simple path. As there will be a downhill course, we will compute the max of c_1 and c_2 and mark the direction from greater to smaller. Here, we will sorting the vertices of the DAG $v_1, v_2....v_n$ such that there is an edge directed towards vertex v_j from v_i then v_i comes before v_j. We can do this either by DFS or BFS traversal. Let's do this with a DFS traversal. In DFS, we print the vertex and then recursively call DFS for adjacent vertices. Hers, we create a stack where keep the vertices. To do that. we recursively call the topological sorting for all the adjacent vertices to a given vertex and then push it into the stack. Once it's done, we print the contents of the stack. So, we will iterate the nodes of the graph(stack) in topological order. For each node v which is a neighbour of u, we have to relax $distance[v]$ using the following equation:

$$distance[v] = min(distance[v], distance[u] + weight(u, v))$$

This will give us the shortest path in a DAG, but as the graph is a directed weighted acyclic graph there is no cycles involved in it, so we can use use topological sort in linear time $O(V + E)$ to solve it which is similar to finding the shortest path but here we use negative weights and take the maximum value as we relax the path. Finding the topological sort will run in $O(V + E)$ time and finding the longest path will run the loop for all adjacent vertices. As total vertices is $O(E)$. So, the inner loop takes $O(V + E)$. Total time taken is $2O(V + E)$ which is $O(V + E)$.

(c) Code has been attached on dropbox.