#Assignment: asg8

#Name: Satish Dodda

#ULID: sdodda1

#Secret Directory: behappy

**Reading input:**

**From the command line argument, read the file name**

**Then open the file with the file name and store it in the file**

**By using for loop iterated over the file**

**If encounter \*\* in line, then split that line where ‘=’ and store the total vertex in the last vertex variable   
if E and {are encountered in line, then the test case starts, update the flag variable c with 1**

**If ‘--’ is not in line and ‘=’ is also not in line, that means there is a tuple in the line, using the eval function, I have converted the line into the tuple, reversed the tuple, and added in adjList and updated the shortest path map key’s as u,v vertices and values as list of empty string and 0 and updated the visited map with key ‘s as u,v vertex’s and value as 0, and updating the weight dictionary key as (u,v) and value as weight of the edge and using is\_negative variable to check the weather a negative weight in the graph or not if we found a negative weight in the graph then it will update the is\_negative with True if ‘}’ is an encounter in line that means that is the last tuple in the graph so I have removed the } from the line and updating the adjList with last tuple**

**If ‘--’ is encountered in a line that means the graph is ended so updating the c with 0 and incrementing the count by one means the nth graph is read successfully and if is\_negative is True then it will call the bellman\_ford() method otherwise it will calling the computershortestPath() method by passing the shortest path, adjList , visited,last\_vertix-1 and clearing the adjList and shortedPath maps visited maps so that while reading new graphs we don’t mess up**

**Finally closing the file**

**Methods for Dijkstra algorithm:**

**computeDijkstra(shortestPath, adjList, visited, last\_vertix)**

@shortestPath is a map, the key represents the vertex and value is the list of two items first indicate the target vertex and second value indicates the minimum cost to visit the edge

@adiList is a map, the key represents the vertex and values is a tuple of three values edge and weight of the edge

@visited is a map, the key represents the vertex and the value is either (0 or 1) if it is one then it is already visited, otherwise not visited

@last\_vertix , which holds the value of the last vertex

This method will compute the shortest from vertex 0 to n-1 and print the shortest path

**Algorithm Working:**

First, I tested for the edge case if 0 is not in adjList that means there is no edge from the vertex 0, which means we don’t have a path from 0 to n-1 th vertex

If 0 is in adjList then add all connected edges to 0 to the priority queue and mark vertex 0 is visited

Then, in the while loop checking condition if len(pq) priority queue is greater than 0 if greater than 0 then enter into loop

From the priority queue pick edge, which as less weight and remove the edge from the priority queue and check edge is present in the shorted path if present and the cost is 0 then update the cost with weight of the edge and mark the vertex as visited

Using a loop to traverse, the edges connect with the vertex that is marked as visited and if the connected vertex with the visited vertex is not visited then add that edge to priority queue and update the cost of the edge by adding shortest Path cost till that visited vertex

Repeat this until the priority queue is empty

Finally using a loop traverse from n-1 to 0 in shortestPath map and put the values in the stack and print the stack for the shortest path

**Data Structures used:**

**Stack=[], it is a list used to print the shortest path in correct order**

**Pq=[] it is list used for priority queue operation**

**adjList ={} it is a map it is used to vertex as key and values a list of tuple which contains edge and weight**

**visited={} it is map it it used to check a vertex is already visited or not**

**shortestPath={} it is map it is used to store the shortestPath for edges from 0 to n-1**

**Methods for BellmanFord Algorithm:**

**getCycleVertexs(cycle, current, track):**

the method will detect the cycle if any cycle is found it will store cycle vertices in the cycle list, if no cycle is found then it will remove the stored edges in the cycle list

@ cycle is a list to store the list of cycle vertices

@ current is the variable that contains the starting node to find the cycle

@ track is a map that will store the min in coming vertex u for every v

**detect\_negative\_cycle(edge\_weight,v,track,total\_vertices):**

first, it will find the cycle vertex in the track map if it finds the cycle vertex, then it will call getCycleVertexs method and store the vertex in the cycle list, print the negative cycle edges and break the program

@ track is a map that will store the min in coming vertex u for every v

@ weight\_by\_edges is a dict that will store the key as (u,v) and values as weight associated with edge (u,v)

@ total\_vertices is the total count of unique vertices in the graph

@ v is vertex v

**update\_dp\_table(weight,track,edges,weight\_by\_edges,total\_vertices):**

the method will check whether the current weight of v is greater than weight u+(u,v) if greater, then we will check for any negative cycle in the graph by calling detect\_negative\_cycle method, if any negative cycle is found, then it will print the negative cycle and stops the program. Otherwise, we will update the weight of v with u+weight(u,v)

@ weight it will store the weights of the edges

@ track is a map it will store the min in coming vertex u for every v

@ edges is the list that contains list of edges (u,v)

@ weight\_by\_edges is a dict that will store key as (u,v) and values as weight associated with edge (u,v)

@ total\_vertices is the total count of unique vertices in the graph

**bellman\_ford(edges,total\_vertices, source,edge\_weight):**

method bellam\_ford method will call the update\_dp\_table method; it will update the dp table if it finds a valid path in the graph with no negative cycle in it After updating the table we are just extracting the path using the track table from last\_vertex to 0, which is source and printing it

@ edges is the list which contains list of edges (u,v)

@ weight\_by\_edges is a dict which will store key as (u,v) and values as weight associated with edge (u,v)

@ total\_vertices is the total count of unique vertices in the graph

@ source starting point which is 0

**BellmanFord algorithm**:

**Step**1: weight is a default dictionary whenever a new vertex is inserted into the default dictionary as a key, then it will update the value as the infinite value

The track is a default dictionary whenever a new vertex is inserted into the default dictionary as a key then it will update the values as None

We are finding the shortest path from 0, so the weight of the source is 0

**Step2**: calling update\_dp\_method by passing weight , track, edges,edge\_weight, last\_vertex as parameters to the method

In total, we have n vertices, from the source, we have to find a path to other n-1 vertex

My first while loop will iterate for n-1 time

For every iteration for loop which is inside this while loop will iterate through all the edges and every have a starting vertex u and ending vertex is v in the graph

**Step3:** if the weight u is not infinite and weight[u]+weight[(u,v)] edge is less than weight[v], this means there is a chance of having a negative cycle

**Step4:** then it will call the detect\_negative\_cycle method by passing edge\_weight, track, last\_vertex and vertex v , it will pass the current vertex to the track dictionary, and it will check the path from the vertex if a vertex is visited twice in the track dictionary , and visited vertex it is not None ,then it means it encounters a cycle

**Step5:** then it will call getCylceVertices by passing cycle, current cycle vertex, and track dictionary this method will find the cycle in the tracklist and place all the cycle vertex in the deque and update the cycle list with deque here, I used deque because it will insert the element in deque at the beginning at O(1) , if the negative cycle is found then it will print the negative cycle and it will stops the algorithms

**Step6:**  if no negative cycle is detected, then it will update the weight[v] with weight[u]+weight[(u,v)] edge, and this process will run until a negative cycle is detected or a valid path found from 0 to all vertex in the graph

**Step7: using the last vertex, backtrack the path in the track map if we have path from n to 0 it will print the path and weight associated with the path otherwise, it will print no path found from the source**

**DataStructures used:**

**Track is the dictionary to track the path from 0 to n**

**Weight is the dictionary to store the min\_cost to travel to that vertex**

**Edges is list which will store the list of edge which contains (u,v) values**

**Find\_cycle\_entry it is a set that is used to check the weather there is a negative cycle or not in a graph**

**Dq is a deque which is used to store the path from 0 to n , I have used this because it only cost o(1) for all operations**

**Edge\_weight It is a dictionary which is used to store the edge as a key and values as weight for that edge**

**Time Complexity :**

As we know for negative weights the Dijkstra algorithm will not , but for the positive weights it will the best algorithm to find the shortest path from source to all vertices in the graph

We have v vertex in the graph and e edges in the , we have to travel these e edges using v vertex and for finding the min\_weight path in graph we are using Priority queue the time complexity of priority queue is O(NlongN) here we have v vertex and e edges so it takes **O((v+E)logE) time complexity when we use the Dijkstra algorithm**

**But when we have a negative weight graph then Dijkstra algorithm will not work**

**So we have to use and dynamic programming algorithm which is BellmanFord algorithm and this will work for negative weight graph also**

**For updating the dp table values it will take O(V\*E) time complexity**

**For find negative cycle in graph it will take O(n)**

**For printing the path it will take O(n)**

**The maximum among this is O(V\*E)**

**When we use the bellman ford algorithm to find the shortest path total it will take O(V\*E) time complexity**

**When we calculate the overall time complexity of the algorithm**

**Max(O((V+E)logE), O(V+E)) , over all time complexity of the algorithm is O(V+E) when we use the bellman ford algorithm**

**Space complexity :**

**For storing edges it will take O(N)**

**For storing weight it will take O(N)**

**For storing cycle it will take O(N)**

**For storing path it will take O(N)**

**for storing data in Dp table it will take O(N)**

**for storing tracking data it will take O(N)**

**over all space complexity is O(V)**

**Difficulties faced :**

**I haven’t faced any difficulties while implementing the code because those two algorithms are easy for graph 21 my Dijkstra is not printing the path but it as path later on I figured out the issue it is caused by my shortest path dictionary, I forget to update the destination vertex in every edge in shortest path dictionary because that the algorithm assumes there is no path in the graph**