Assignment: asg3

Name: satish Dodda

ULID: 809961786

Secret Directory: behappy

**For 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 total number of a vertex in the TotalVertex 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 list and added that into the edge list, and added the vertex in edge list to the vertex set and added the edge and weight of the edge to weight map**

**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 calling the find\_arborescences() method by passing the edge, vertex, weight and after passing clearing the edge and vertex weight maps so that while reading new graphs we don’t mess up**

**Finally closing the file**

**Methods:**

**find\_Indegree(min\_indegree,edge,weight):**

#@min\_indegree map it will store the min\_indegree values for every vertex expect root

# @edge is a list that will store a list of edges

# @weight is a map that will store the weights of the edge

#find\_indegree method finds the min\_indegree for every vertex in the edge list, except the root

Using for loop iterate over the edges list if edge[i][1] is not in min\_indegree map, then for edge[i][1] put edge in the map

If suppose edge[i][1] is already in min\_indegree then find the min edge among then and updating the min\_indegree

after identifying the minimum incoming edge for each vertex, subtract the weight of the minimum incoming edge from every other incoming edge to that vertex

from the above step, we will get, min\_indegree edges with modified weight zero

**is\_cycle(min\_indegree,vertex):**

# @min\_indegree is map it will store the min\_indegree weight for every vertex

# @vertex is set which will store the set of vertex

# is\_cycle method find cycle if any cycle is found in min\_indegree map

Using for loop iterate over the set of vertex

If a vertex is not root and the vertex is not in the visited set, and it is in min\_Indegree map

Then, take the next vertex from the min\_indegree map and continue this process until the v is in the visited set or v is not in min\_indegree map

If v is already visited that means we have a cycle

Then using the while loop I stored all cycle vertex in the cycle list and returned it

**find\_arboresence(edge,vertex,weight,count):**

@ edge is a list that contains a list of edges

# @ vertex is a set that contains a set of vertex

# @ weight is a map that contains the weight of the edge

# @ count is a number to uniquely name the cycles

# This method finds the minimum cost arborescence, and it will return it

**Algorithm Working**:

By using the find\_Indegree method finding the min indegree for every vertex after identifying the minimum incoming edge for each vertex, subtract the weight of the minimum incoming edge from every other incoming edge to that vertex

By using the is\_cycle method, find whether we have a cycle or not in the min\_indegree map if we have a cycle then the method will return the cycle list otherwise empty list

If we don’t have a cycle, then returning the min\_indegree edges

If we have a cycle then :

Storing all non cycle vertex to the no\_cycle\_vertex set

And add the supper\_node to the no\_cycle\_vertex set

And taking edge1 list to store new graph

And track map to track the original values of (u,v)

Creating a super node with name s+count , here count represents the number of the super node

new\_weight map to store the new graph weights

Using for loop iterate over the edge list (u,v) let u is the source vertex and v is the destination vertex

Graph contraction

**If the source vertex is in the cycle and the destination vertex is not in the cycle :**

**Then create a new edge by using the (cycle\_node, v) and name it e**

**If e is already present in new\_weight map then compare the weights of the current previous edges if the current edge is less than the previous one then update the weight otherwise skip the edge**

**Update the track map with the key as e and values as tuple of (u,v)**

**And add the edge e to the new edge1 list**

**Update the weight [e] with the weight of the edge**

**If the source vertex is not in the cycle and the destination vertex is in the cycle :**

**Then create a new edge by using the (u, cycle\_node) and name it e**

**If e is already present in new\_weight map then compare the weights of the current previous edges if the current edge is less than the previous one then update the weight otherwise skip the edge**

**Update the track map with the key as e and values as tuple of (u,v)**

**And add the edge e to new edge1 list**

**Update the weight [e] with the weight of the edge**

**If u and u is not in cycle list then:**

**Update the track map key as e and value as (u,v)**

**Add the edge e to new edge1 list**

**Update the weight[e] with edge weight**

**In the above, contraction removes the cycle edges from the graph and replaces the cycle edges with the supper node**

**And calling the find\_ arboresence by passing new graph edge1 new\_weight map and new vertex list**

**This process will continue until no cycle is left in the graph**

**Expansion :**

Using for loop iterate over the graph if destination vertex equal to super \_Node :

By using the track map by passing (u,v) to it, getting the previous value of u,v, we will get the original values of (u,v) now, bypassing v to min\_indegree map, I find the incoming edge of the v , here I found the edge that I have to delete

Extract the original values by using the track map by passing (u,v ) from the graph

Now using for loop iterate over the cycle

By passing v to the min\_indegree map, we find the edge for vertex v

If the edge is equal to the deleted edge then skip the edge other wise add the edge to expand list and return the list

End of the algorithm, it will return the cost of Arborescence

**Data Structure used :**

Edge is a list that will store the list of edges

Vertex is a set that will store the set of vertex

Weight is a map, it will store the edge as key and values as weight

Cycle is list it will store the cycle list

Min\_Indegree is a map for every vertex it will store the min incoming edge

Visited is a set that is used to check whether a cycle exists in min\_indegree map

The track is a map that uses (super\_node, (u or v)) as the key and values are original values of (u,v)

No\_cycle\_vertex is a set that will store the vertex with no cycles

New\_weight is a map which will store edge a key and value as weight of the edge

**Time Complexity:**

To find the minimum in-degree edge for each vertex, which will cost O(E ) because, in total, we have an E edge in the graph

To find the cycle in the worst case, it might include all the vertex so it will take O(V)

Once the cycle is detected the algorithm contracts the cycle into super\_node and recursively calls the algorithm

In each at least the size of the graph decreases by one vertex b because of the contraction in the worst case this will result in V recursive calls

For each recursive call the algorithm must process all edges and update the weight it will take O(E)

Finding the minimum coming edges it takes O(E)

For the detecting cycle, it will take O(V)

Recursion and edge contraction each recursive call needs O(E) to adjust weight and O(V) to find the cycle in worst-case

**Total Time complexity will be O(V\*E)**

**Space complexity :**

For adjusting the edge weight in it will take O(E) and for storing v vertex it will take O(V)

**Total space complexity will O(E+V)**

**Difficulties faced:**

I faced one difficulty in the expansion while expanding the graph, I was stuck with how I could get the original values for the data in the super node after spending some time with it, I came up with the solution to use a map to keep track of the vertex’s of edges before removing edge and compressing it into super node it helped me to expand the graph easily, and that solved my problem