GRAPH ALGORITHMS…..

----DJKSTRA ALGO SINGLE SOURCE SHORTEST PATH

from collections import deque

class Solution:

# Function to return list containing vertices in Topological order.

def topoSort(self, V, adj):

indegree = [0] \* V

q = deque()

# Compute in-degrees of all vertices

for u in adj:

for v in u:

indegree[v] += 1

# Enqueue vertices with zero in-degree

for i in range(V):

if indegree[i] == 0:

q.append(i)

ans = []

while q:

d = q.popleft()

ans.append(d)

# Decrease the in-degree of neighboring vertices

for i in adj[d]:

indegree[i] -= 1

if indegree[i] == 0:

q.append(i)

return ans

|  |  |
| --- | --- |
| Time Complexity | O((V + E) log V) |
| Space Complexity | O(V) |

--------BELLMAN FORD ALGO SINGLE SOURCE SHORTEST FOR NEG CYCLE

class Solution:

# Function to construct and return cost of MST for a graph

# represented using adjacency matrix representation

'''

V: nodes in graph

edges: adjacency list for the graph

S: Source

'''

def bellman\_ford(self, V, edges, S):

#code here

import math

distance={}

n=False

for i in range(V):

distance[i]=int(1e8)

distance[S]=0

for i in range(V-1):

for u,v,w in edges:

if distance[u]!=int(1e8) and distance[v]>distance[u]+w:

distance[v]=distance[u]+w

for u,v,w in edges:

if distance[u]!=int(1e8) and distance[v]>distance[u]+w:

return [-1]

return [x for x in distance.values()]

| **Time Complexity** | **Space Complexity** |
| --- | --- |
| **Initialization** | O(V) | O(V) |
| **Relaxation** | O(V\*E) | O(1) |
| **Overall Complexity** | O(V\*E) | O(V) |

----PRIMS ALGO FOR MST…..

import heapq

class Solution:

#Function to find sum of weights of edges of the Minimum Spanning Tree.

def spanningTree(self, V, adj):

#code here

heap=[]

visited=[False] \*V

mst=0

heapq.heappush(heap,[0,0])

while heap:

top=heap[0]

heapq.heappop(heap)

node=top[1]

cost=top[0]

if visited[node]==True:

continue

mst+=cost

visited[node]=True

for i in adj[node]:

w=i[1]

n=i[0]

if visited[n]==False:

heapq.heappush(heap,[w,n])

return mst

| **Aspect** | **Complexity** |
| --- | --- |
| **Time Complexity** | O((V + E) log V) |
| **Space Complexity** | O(V + E) |

KRUSHKALS ALGORITHM….for mst

def find(i,parent):

if parent[i]!=i:

parent[i]=find(parent[i],parent)

return parent[i]

def union(u,v,parent,rank):

a=find(u,parent)

b=find(v,parent)

if a!=b:

if rank[a]<rank[b]:

parent[a]=b

elif rank[b]<rank[a]:

parent[b]=a

else:

parent[b]=a

rank[a]=rank[a]+1

def kruskalMST(n: int, edges: List[List[int]]) -> int:

# Write your code here

parent=[i for i in range(n+1)]

rank=[0] \*(n+1)

edges.sort(key=lambda x:x[2])

mst=0

li=[]

for i in edges:

u=i[0]

v=i[1]

if find(u,parent) != find(v,parent):

mst+=i[2]

union(u,v,parent,rank)

return mst

|  |  |
| --- | --- |
| **Time Complexity** | O(E log E) |
| **Space Complexity** | O(V + E) |

----TOPOLOGICAL SORTING

BFS

from collections import deque

class Solution:

#Function to return list containing vertices in Topological order.

def topoSort(self, V, adj):

indegree=[0]\*V

q=deque()

for u in adj:

for v in u:

indegree[v]=indegree[v]+1

for i in range(V):

if indegree[i]==0:

q.append(i)

ans=[]

while q:

d=q.popleft()

ans.append(d)

for i in adj[d]:

indegree[i]=indegree[i]-1

if indegree[i]==0:

q.append(i)

return ans

The time complexity for constructing the graph is O(V + E), where V is the number of vertices and E is the number of edges.

The time complexity for performing topological sorting using BFS is also O(V + E), where V is the number of vertices and E is the number of edges. This is because each vertex and each edge is visited once during the BFS traversal.

**Space Complexity:**

The space complexity for storing the graph using an adjacency list is O(V + E), where V is the number of vertices and E is the number of edges.

-------------USING DFS TOPOLOGICAL SORT-----

**def** topologicalSortUtil(v, adj, visited, stack):

*# Mark the current node as visited*

visited[v] = **True**

*# Recur for all adjacent vertices*

**for** i **in** adj[v]:

**if** **not** visited[i]:

topologicalSortUtil(i, adj, visited, stack)

*# Push current vertex to stack which stores the result*

stack.append(v)

*# Function to perform Topological Sort*

**def** topologicalSort(adj, V):

*# Stack to store the result*

stack = []

visited = [**False**] \* V

*# Call the recursive helper function to store*

*# Topological Sort starting from all vertices one by*

*# one*

**for** i **in** range(V):

**if** **not** visited[i]:

topologicalSortUtil(i, adj, visited, stack)

*# Print contents of stack*

print("Topological sorting of the graph:", end=" ")

**while** stack:

print(stack.pop(), end=" ")

-----KOSARAJU’S ALGO STRONGLY CONNECTED COMPONENTS

class Solution:

def dfsnew(self,s,adj,visited,stack):

visited[s]=True

stack.append(s)

for i in adj[s]:

if not visited[i]:

self.dfsnew(i,adj,visited,stack)

def dfs(self,s,visited,adj,stack):

visited[s]=True

for i in adj[s]:

if not visited[i]:

self.dfs(i,visited,adj,stack)

stack.append(s)

#Function to find number of strongly connected components in the graph.

def kosaraju(self, V, adj):

visited=[False]\*V

s=[]

for i in range(V):

if not visited[i]:

self.dfs(i,visited,adj,s)

new=[[] for i in range(V)]

for i in range(V):

for v in adj[i]:

new[v].append(i)

newvisited=[False]\*V

newstack=[]

c=0

while s:

index=s.pop()

if not newvisited[index]:

self.dfsnew(index,new,newvisited,newstack)

c+=1

return c

TIME COMPLEXITY:O(V+E)