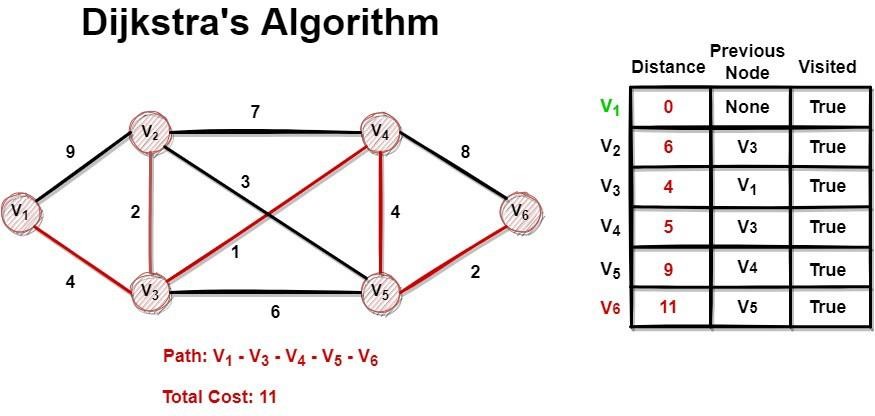
## Practical 8

### **Aim:** Write a program to implement Dijkstra’s algorithm

**Theory:**

1. Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.
2. It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.
3. Dijkstra algorithm is a single-source shortest path algorithm. Here, singlesource means that only one source is given, and we have to find the shortest path from the source to all the nodes.

**Example:**



**Algorithm:**

DIJKSTRA(G, w, s)

1. INITIALIZE-SINGLE-SOURCE (G,s)
2. S = Not Null
3. Q = G.V
4. while Q ≠ Not Null
5. u = EXTRACT-MIN(Q)
6. S = S ∪ {u}
7. for each vertex v ∈ G.Adj[u]
8. RELAX(u,v,w)

**Code:**

class Graph():

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = [[0 for column in range(vertices)]

for row in range(vertices)]

def printSolution(self, dist):

print("Vertex \t Distance from Source")

for node in range(self.V):

print(node, "\t\t", dist[node])

# A utility function to find the vertex with

# minimum distance value, from the set of vertices

# not yet included in shortest path tree

def minDistance(self, dist, sptSet):

# Initialize minimum distance for next node

min = 1e7

# Search not nearest vertex not in the

# shortest path tree

for v in range(self.V):

if dist[v] < min and sptSet[v] == False:

min = dist[v]

min\_index = v

return min\_index

# Function that implements Dijkstra's single source

# shortest path algorithm for a graph represented

# using adjacency matrix representation

def dijkstra(self, src):

dist = [1e7] \* self.V

dist[src] = 0

sptSet = [False] \* self.V

for cout in range(self.V):

# Pick the minimum distance vertex from

# the set of vertices not yet processed.

# u is always equal to src in first iteration

u = self.minDistance(dist, sptSet)

# Put the minimum distance vertex in the

# shortest path tree

sptSet[u] = True

# Update dist value of the adjacent vertices

# of the picked vertex only if the current

# distance is greater than new distance and

# the vertex in not in the shortest path tree

for v in range(self.V):

if (self.graph[u][v] > 0 and

sptSet[v] == False and

dist[v] > dist[u] + self.graph[u][v]):

dist[v] = dist[u] + self.graph[u][v]

self.printSolution(dist)

# Driver program

g = Graph(9)

g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],

[4, 0, 8, 0, 0, 0, 0, 11, 0],

[0, 8, 0, 7, 0, 4, 0, 0, 2],

[0, 0, 7, 0, 9, 14, 0, 0, 0],

[0, 0, 0, 9, 0, 10, 0, 0, 0],

[0, 0, 4, 14, 10, 0, 2, 0, 0],

[0, 0, 0, 0, 0, 2, 0, 1, 6],

[8, 11, 0, 0, 0, 0, 1, 0, 7],

[0, 0, 2, 0, 0, 0, 6, 7, 0]

]

g.dijkstra(0)

print("Neeraj Appari 021")

**Output**:

Vertex Distance from Source

0 0

1 4

2 12

3 19

4 21

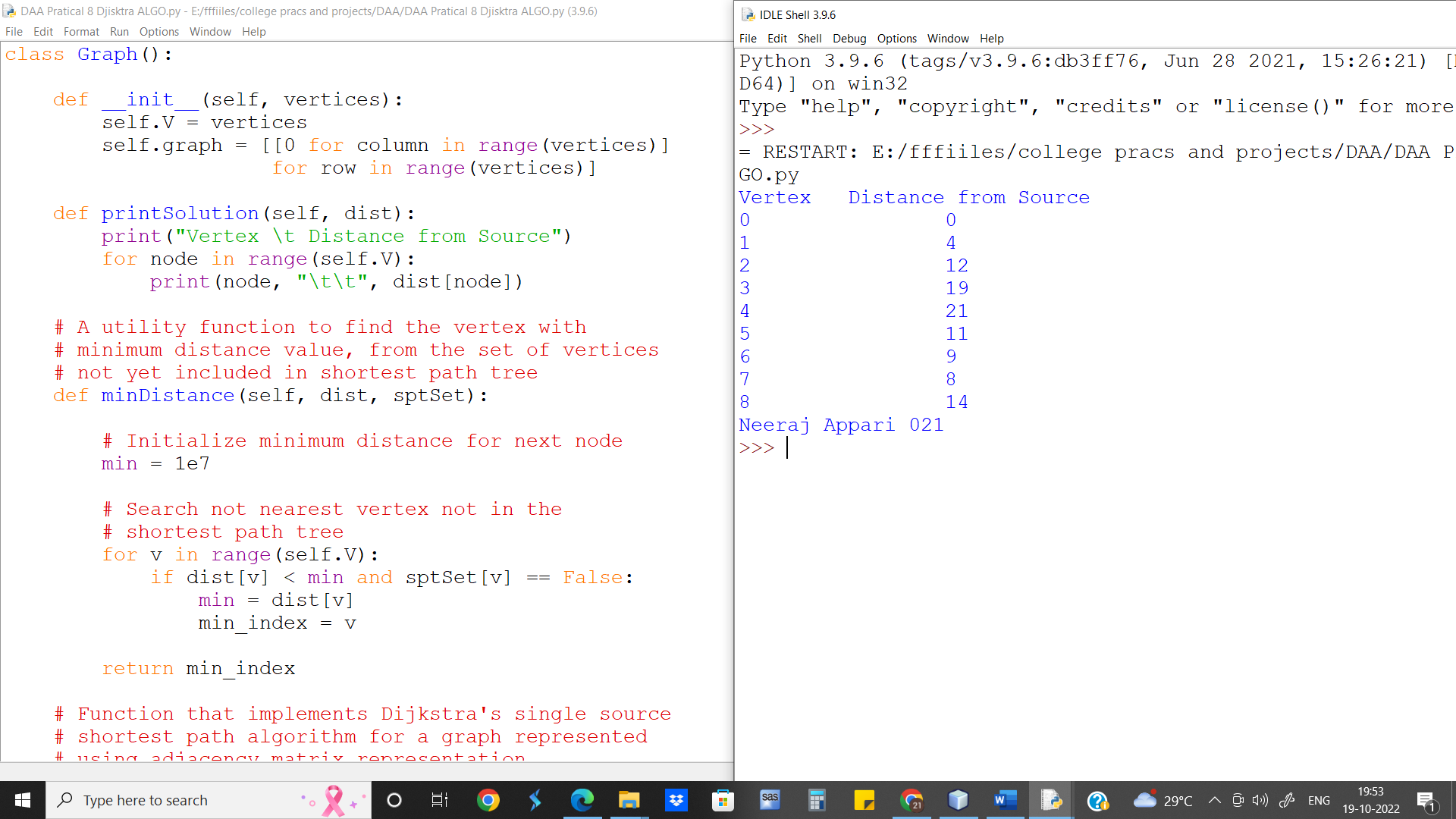
5 11

6 9

7 8

8 14

Neeraj Appari 021



**Complexity Analysis:-** We go through each edge once which gives us O(E), where E is the number of edges.

Also we visit each node/ vertex once which gives us O(V), where V is the number of vertices.

Since we use priority queue we take O(V log|V|) time to sort the nodes and O(1) time to find the closest neighbour.

Discarding O(1) as the lower order complexity we get O( E + V log|V|) as the time complexity for Dijkstra algorithm.

**Conclusion:-** We implement Dijkstra algorithm and find the time complexity of the same