**Assignment No.3 (c)**

**Single-Source Shortest Path Problem**

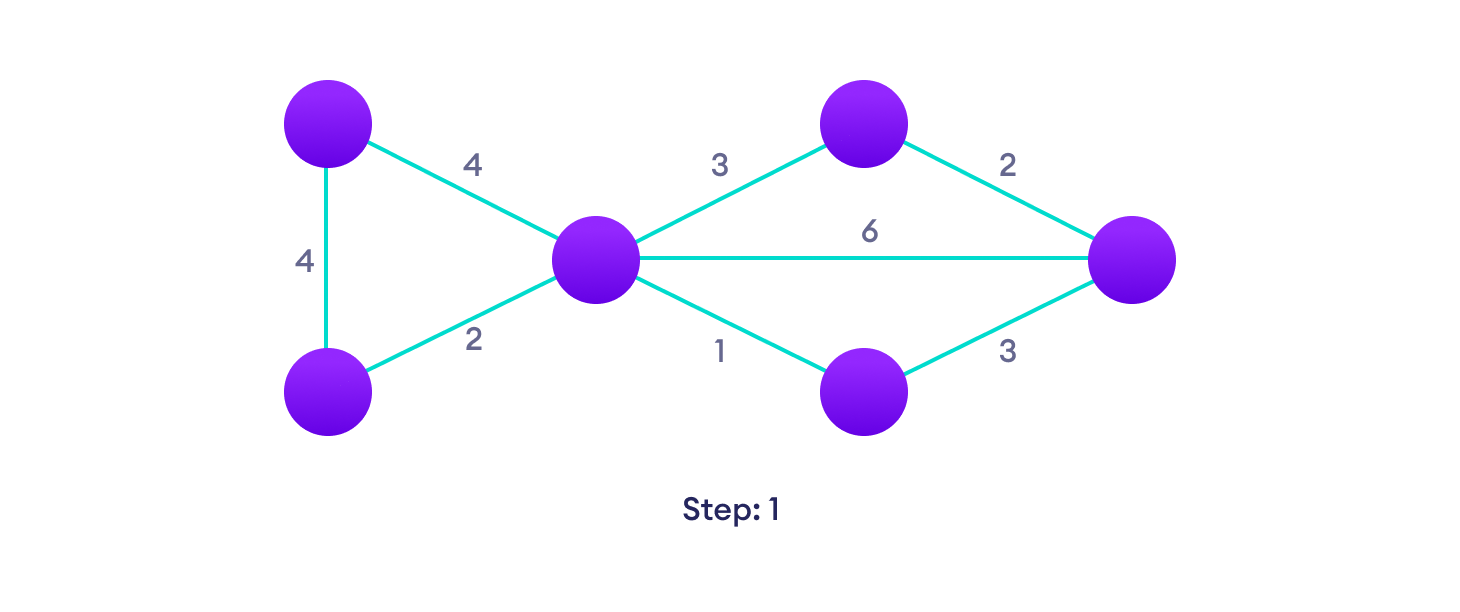
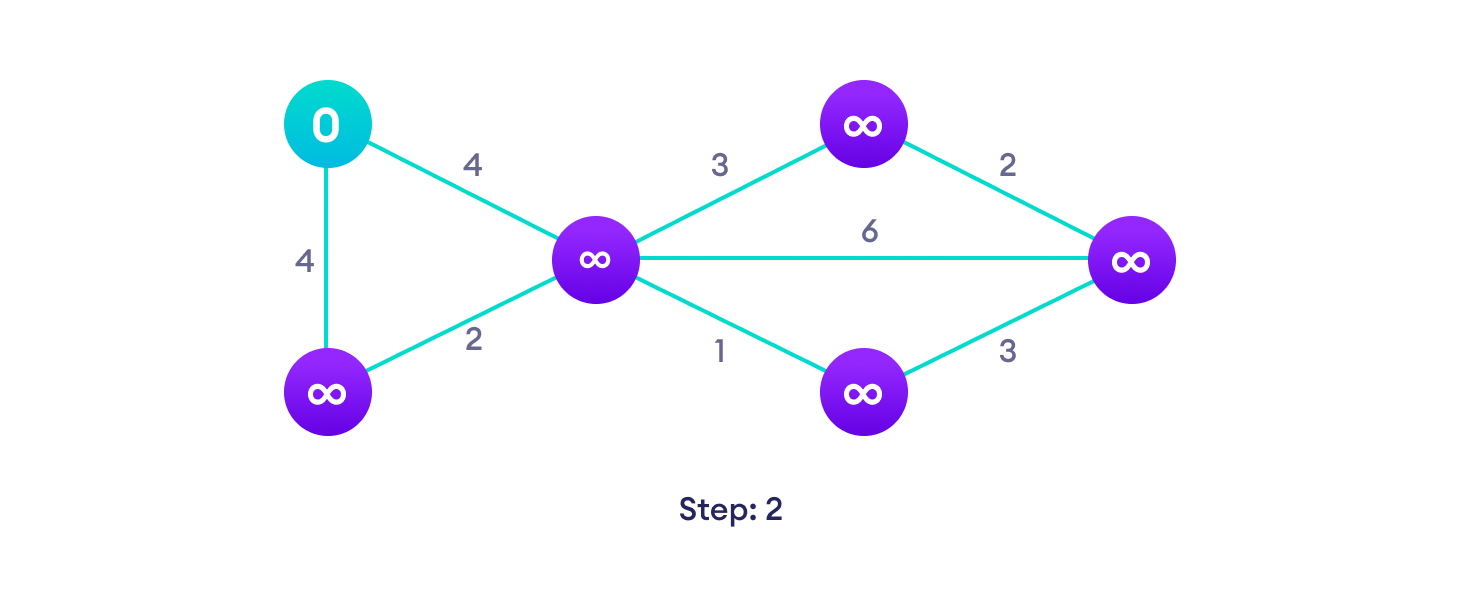
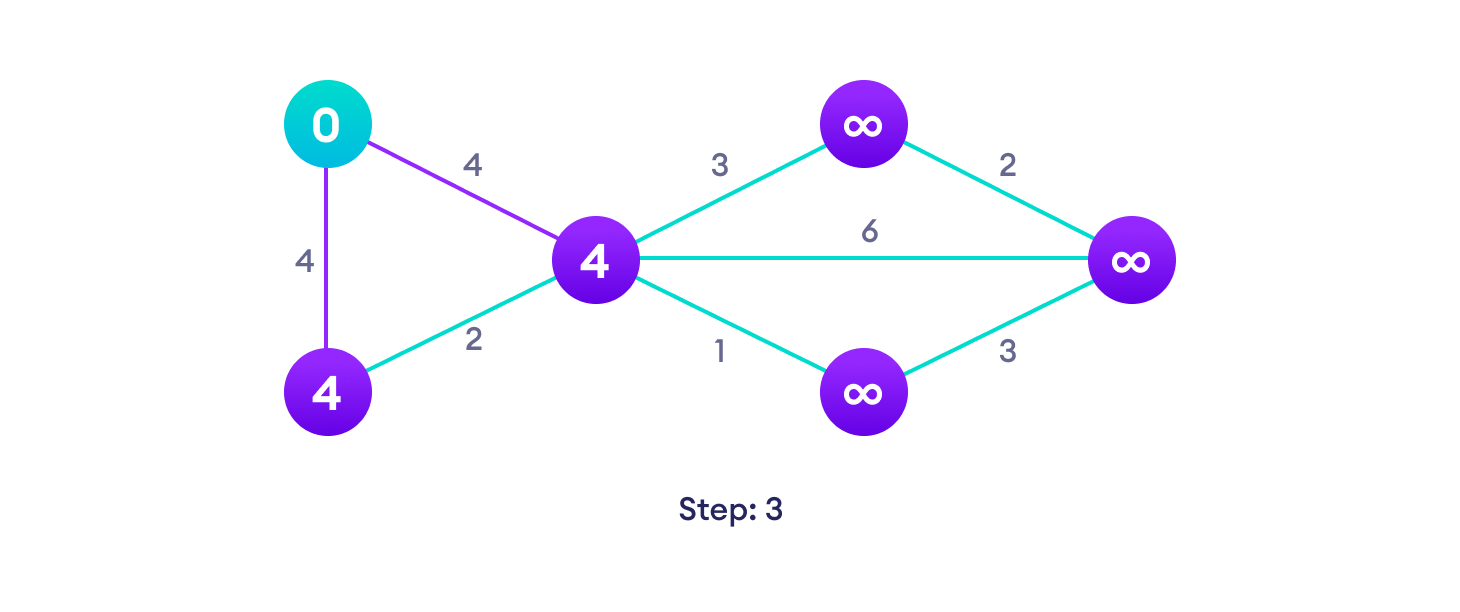
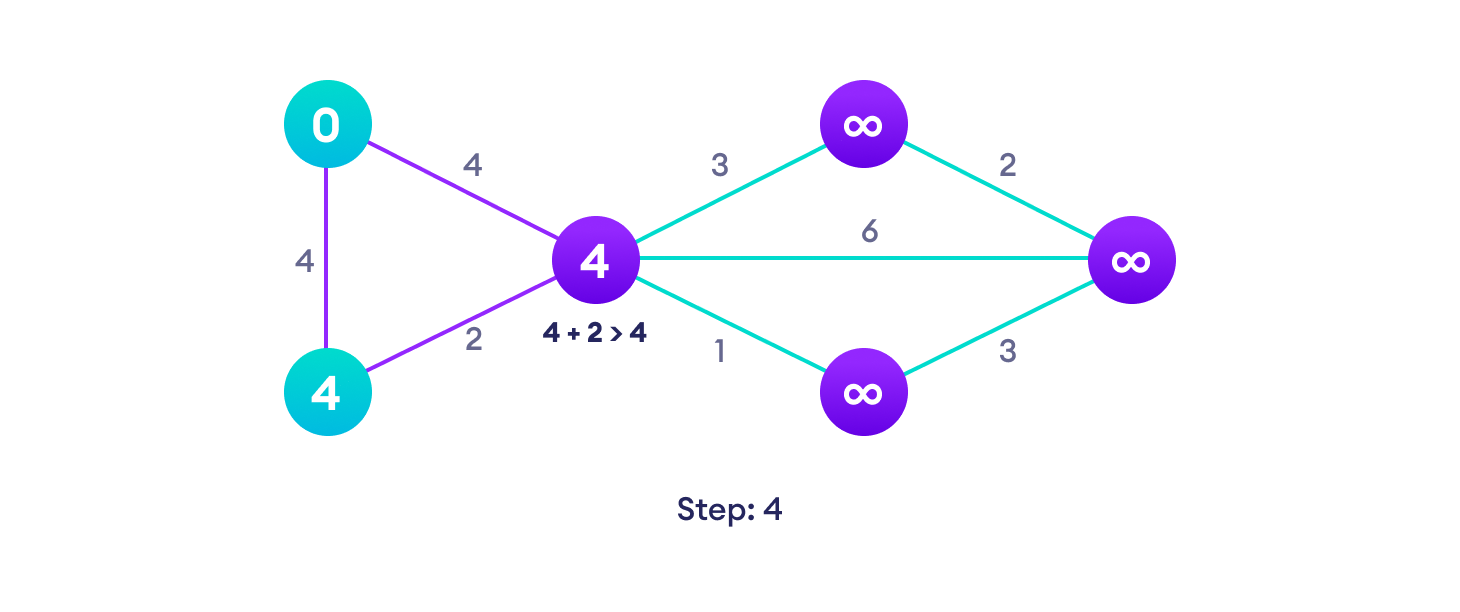
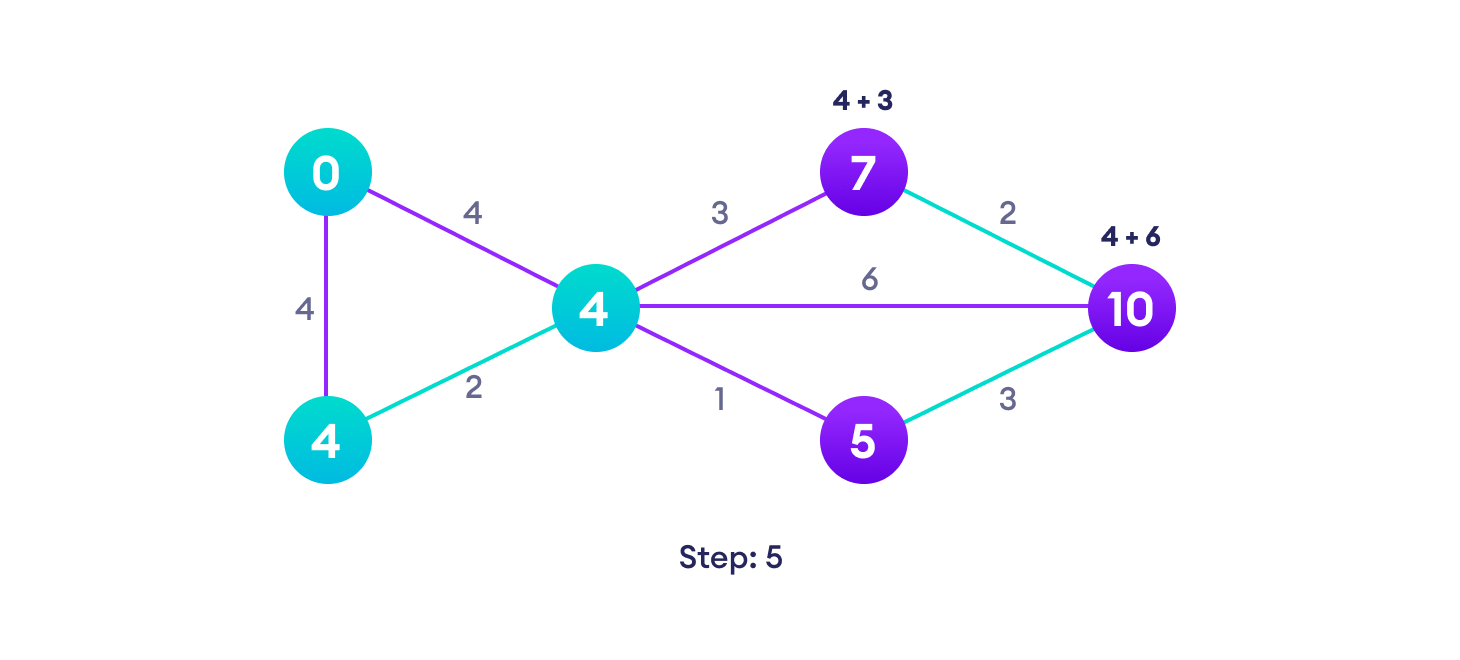
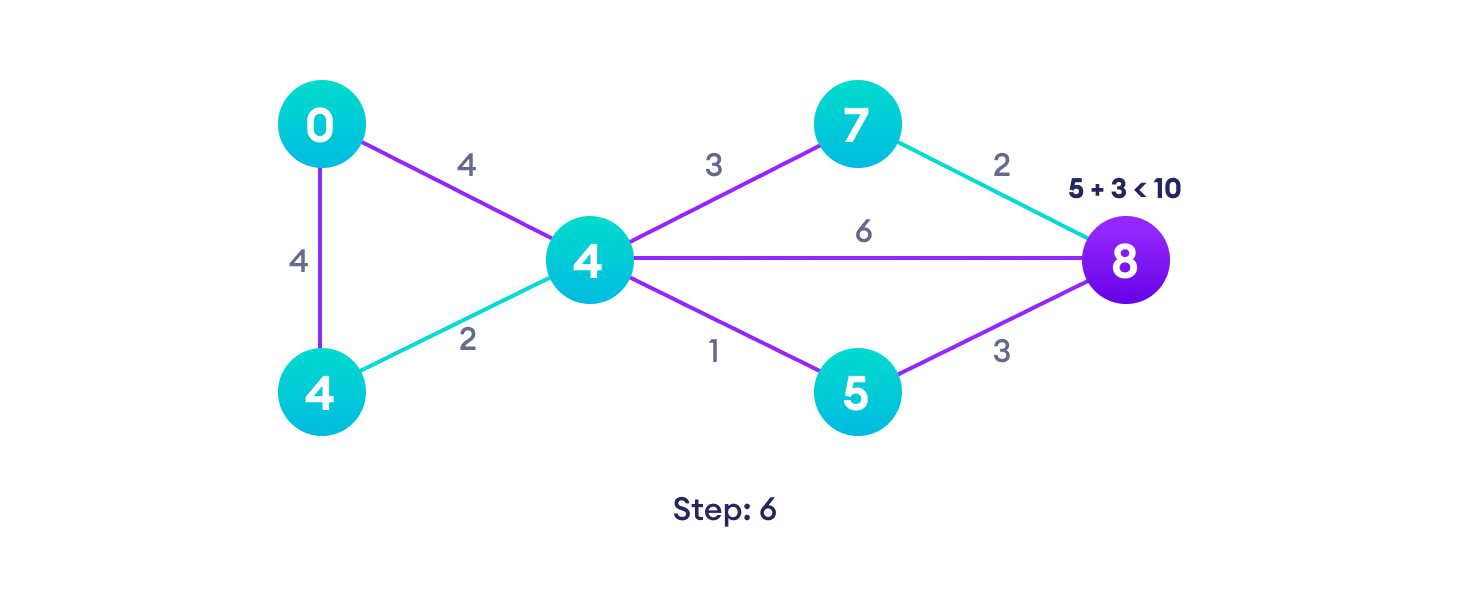
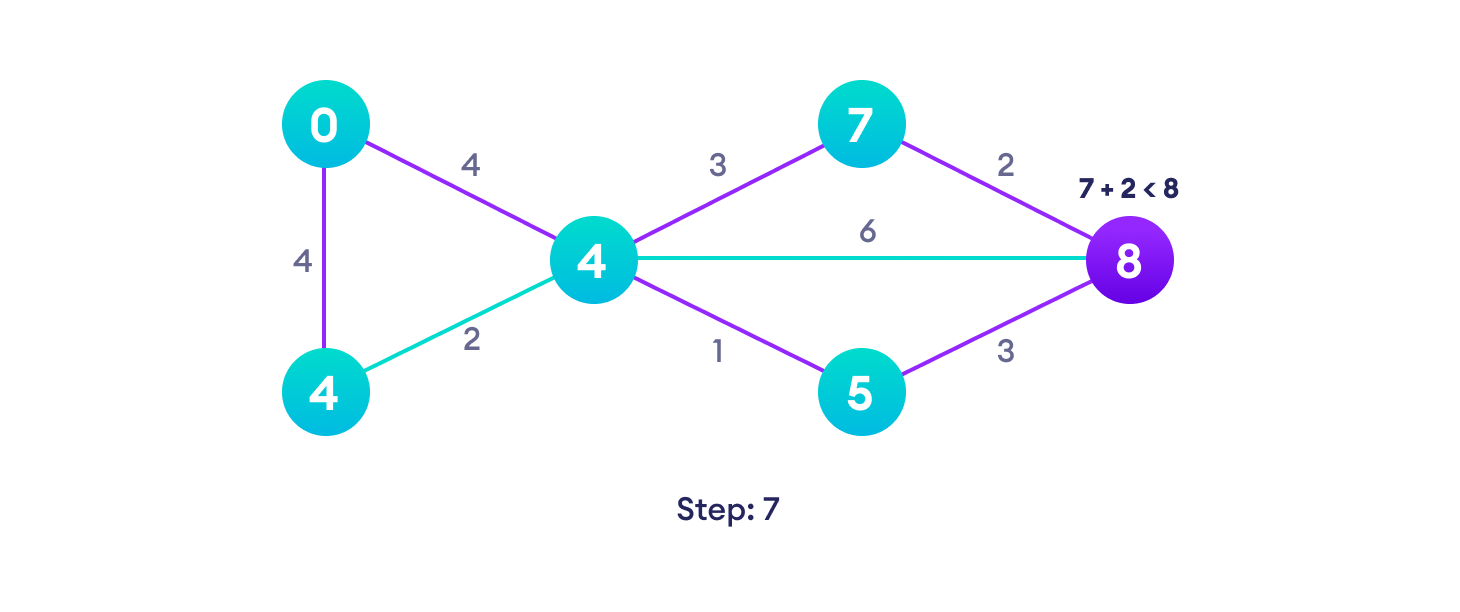
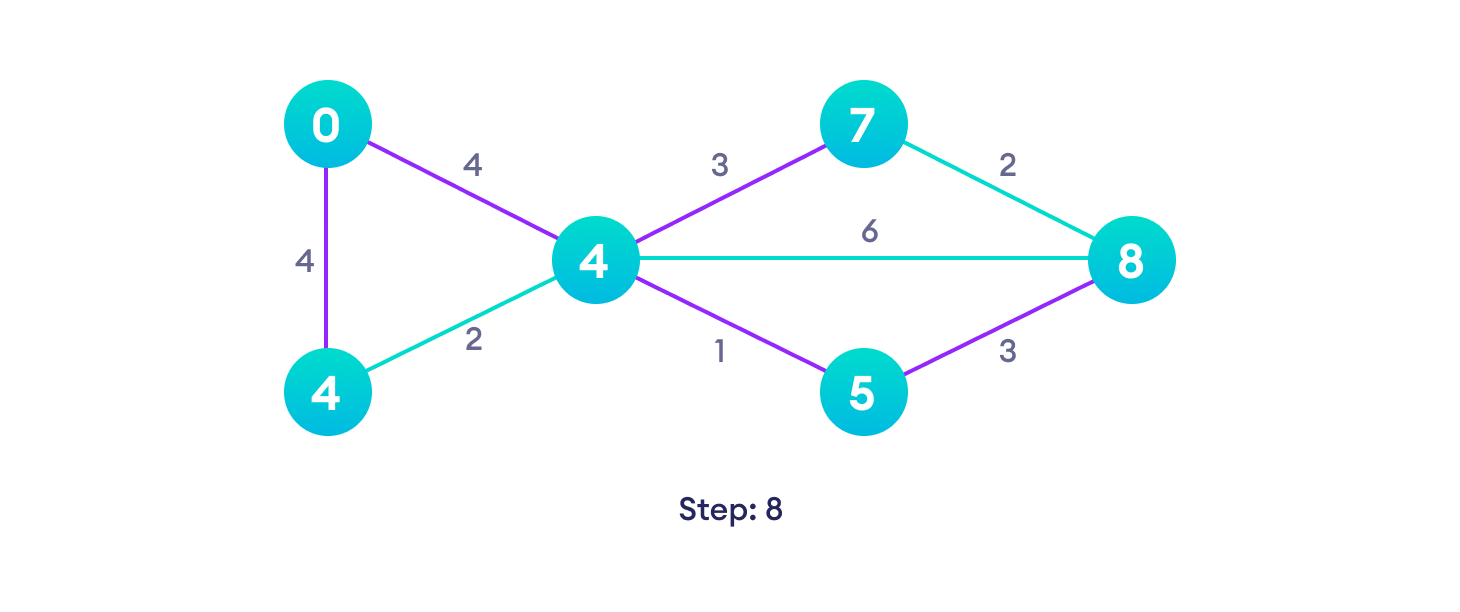
The Single-Source Shortest Path (SSSP) problem consists of finding the shortest paths between a given vertex v and all other vertices in the graph. This problem is mostly solved using Dijkstra, though in this case a single result is kept and other shortest paths are discarded.

**Dijkstra's Algorithm**

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

## **Example of Dijkstra's algorithm**

It is easier to start with an example and then think about the algorithm.

Start with a weighted graphChoose a starting vertex and assign infinity path values to all other devicesGo to each vertex and update its path lengthIf the path length of the adjacent vertex is lesser than new path length, don't update itAvoid updating path lengths of already visited verticesAfter each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7Notice how the rightmost vertex has its path length updated twiceRepeat until all the vertices have been visited

## **Djikstra's algorithm pseudocode :**

**function dijkstra(G, S)**

for each vertex V in G

distance[V] <- infinite

previous[V] <- NULL

If V != S, add V to Priority Queue Q

distance[S] <- 0

while Q IS NOT EMPTY

U <- Extract MIN from Q

for each unvisited neighbour V of U

tempDistance <- distance[U] + edge\_weight(U, V)

if tempDistance < distance[V]

distance[V] <- tempDistance

previous[V] <- U

return distance[], previous[]

## **Dijkstra's Algorithm Complexity :**

Time Complexity: O(E Log V)

where, E is the number of edges and V is the number of vertices.

Space Complexity: O(V)

## **Dijkstra's Algorithm Applications :**

* To find the shortest path
* In social networking applications
* In a telephone network
* To find the locations in the map

**Conclusion :**

**Implementation :**

# Dijkstra's Algorithm in Python

import sys

# Providing the graph

vertices = [[0, 0, 1, 1, 0, 0, 0],

[0, 0, 1, 0, 0, 1, 0],

[1, 1, 0, 1, 1, 0, 0],

[1, 0, 1, 0, 0, 0, 1],

[0, 0, 1, 0, 0, 1, 0],

[0, 1, 0, 0, 1, 0, 1],

[0, 0, 0, 1, 0, 1, 0]]

edges = [[0, 0, 1, 2, 0, 0, 0],

[0, 0, 2, 0, 0, 3, 0],

[1, 2, 0, 1, 3, 0, 0],

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

[0, 0, 3, 0, 0, 2, 0],

[0, 3, 0, 0, 2, 0, 1],

[0, 0, 0, 1, 0, 1, 0]]

# Find which vertex is to be visited next

def to\_be\_visited():

global visited\_and\_distance

v = -10

for index in range(num\_of\_vertices):

if visited\_and\_distance[index][0] == 0 \

and (v < 0 or visited\_and\_distance[index][1] <=

visited\_and\_distance[v][1]):

v = index

return v

num\_of\_vertices = len(vertices[0])

visited\_and\_distance = [[0, 0]]

for i in range(num\_of\_vertices-1):

visited\_and\_distance.append([0, sys.maxsize])

for vertex in range(num\_of\_vertices):

# Find next vertex to be visited

to\_visit = to\_be\_visited()

for neighbor\_index in range(num\_of\_vertices):

# Updating new distances

if vertices[to\_visit][neighbor\_index] == 1 and \

visited\_and\_distance[neighbor\_index][0] == 0:

new\_distance = visited\_and\_distance[to\_visit][1] \

+ edges[to\_visit][neighbor\_index]

if visited\_and\_distance[neighbor\_index][1] > new\_distance:

visited\_and\_distance[neighbor\_index][1] = new\_distance

visited\_and\_distance[to\_visit][0] = 1

i = 0

# Printing the distance

for distance in visited\_and\_distance:

print("Distance of ", chr(ord('a') + i),

" from source vertex: ", distance[1])

i = i + 1

**Output:**

Distance of a from source vertex: 0

Distance of b from source vertex: 3

Distance of c from source vertex: 1

Distance of d from source vertex: 2

Distance of e from source vertex: 4

Distance of f from source vertex: 4

Distance of g from source vertex: 3