Design analysis AND ALGORITHM

PRACTICAL NO 8

027\_Abhishek\_Ojha

**Experiment No - 8 Date of Experiment :- 23 October 2021**

**Program :-** Write a program to Implement Dijkstra’s algorithm.

**Algorithm**

**Dijkstra’s Algorithm**

1. Create cost matrix C[ ][ ] from adjacency matrix adj[ ][ ]. C[i][j] is the cost of going

from vertex i to vertex j. If there is no edge between vertices i and j then C[i][j] is

infinity.

2. Array visited[ ] is initialized to zero.

for(i=0;i<n;i++)

visited[i]=0;

3. If the vertex 0 is the source vertex then visited[0] is marked as 1.

4. Create the distance matrix, by storing the cost of vertices from vertex no. 0 to n-1

from the source vertex 0.

for(i=1;i<n;i++)

distance[i]=cost[0][i];

Initially, distance of source vertex is taken as 0. i.e. distance[0]=0;

5. for(i=1;i<n;i++)

– Choose a vertex w, such that distance[w] is minimum and visited[w] is 0. Mark

visited[w] as 1.

– Recalculate the shortest distance of remaining vertices from the source.

– Only, the vertices not marked as 1 in array visited[ ] should be considered for

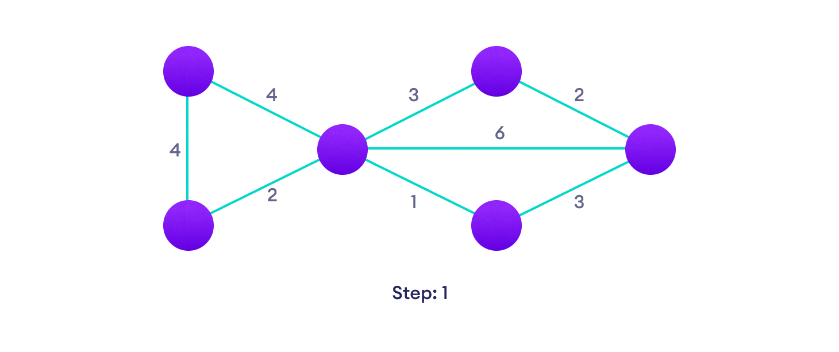
recalculation of distance. i.e. for each vertex v

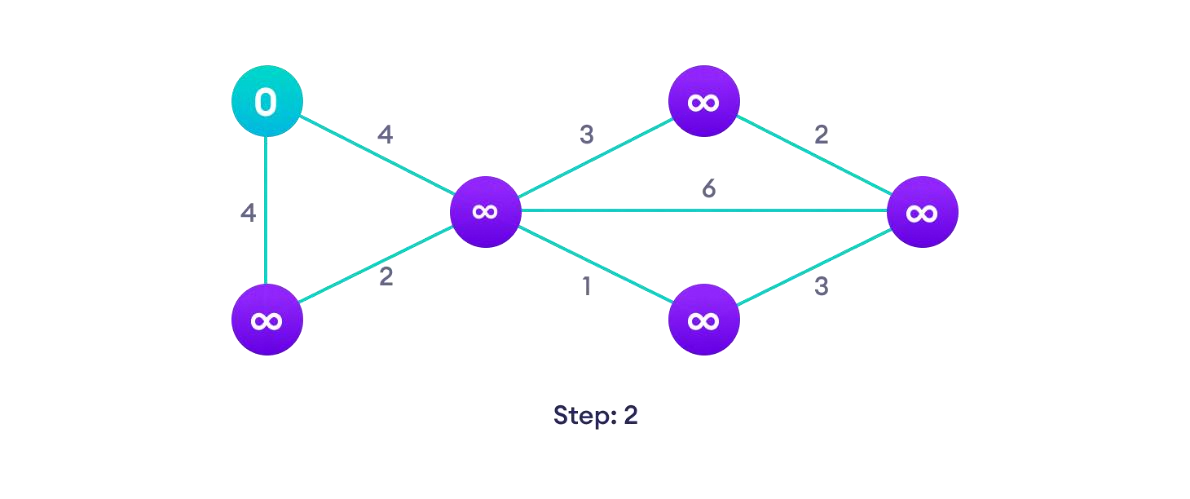
if(visited[v]==0)

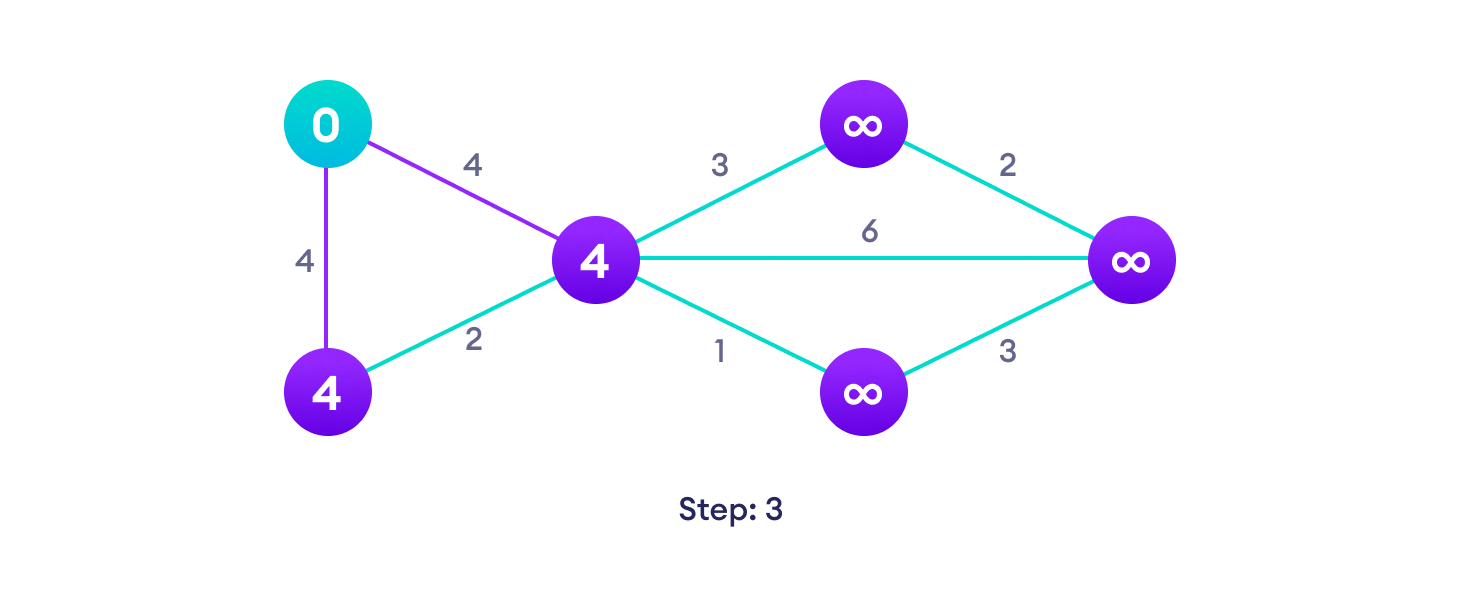
distance[v]=min(distance[v],

distance[w]+cost[w][v])

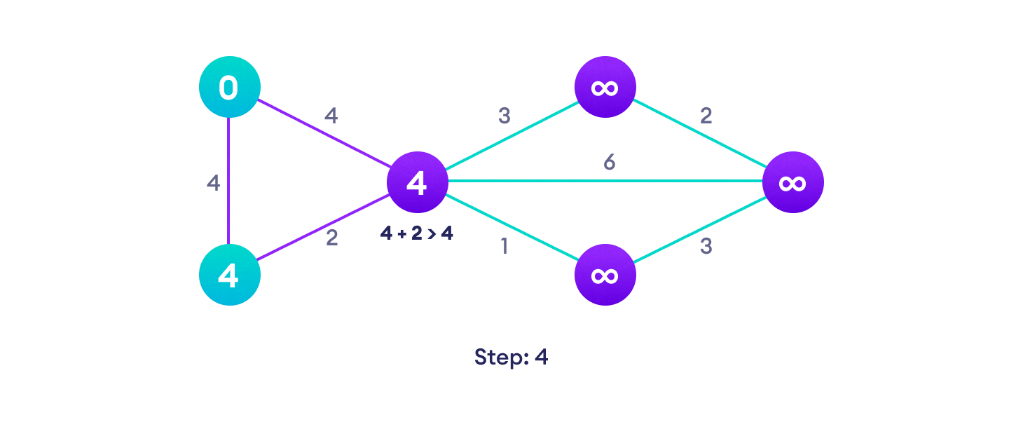
**Fig:**

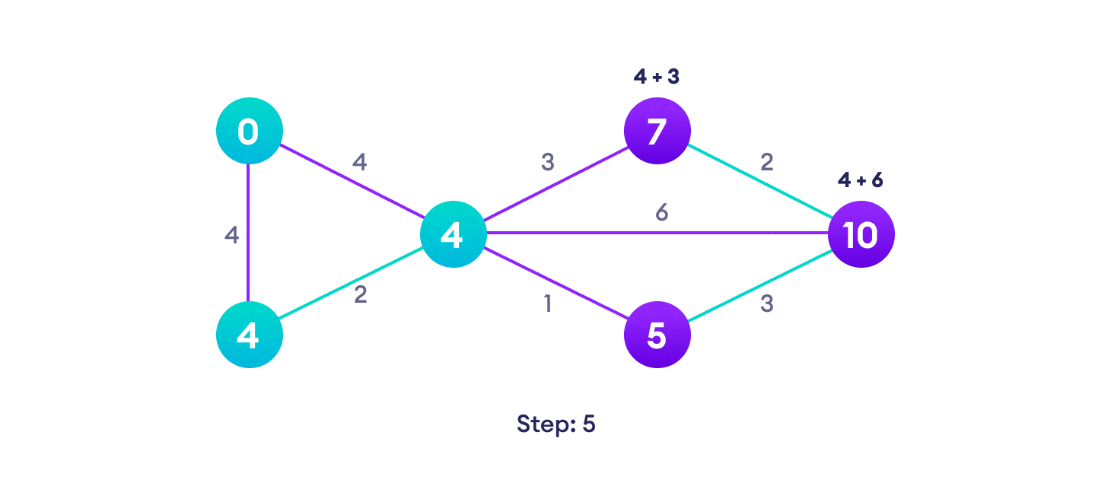


**Start with a weighted graph**

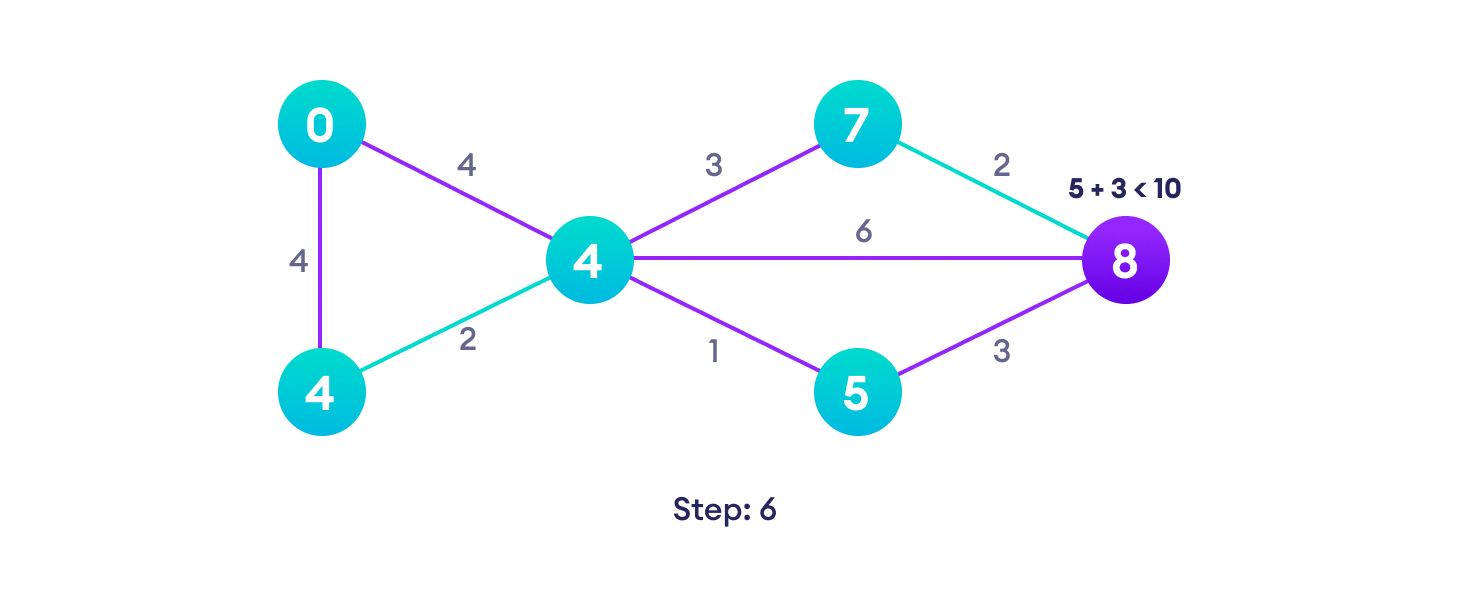
**Choose a starting vertex and assign infinity path values to all other devices**

**Go to each vertex and update its path length**

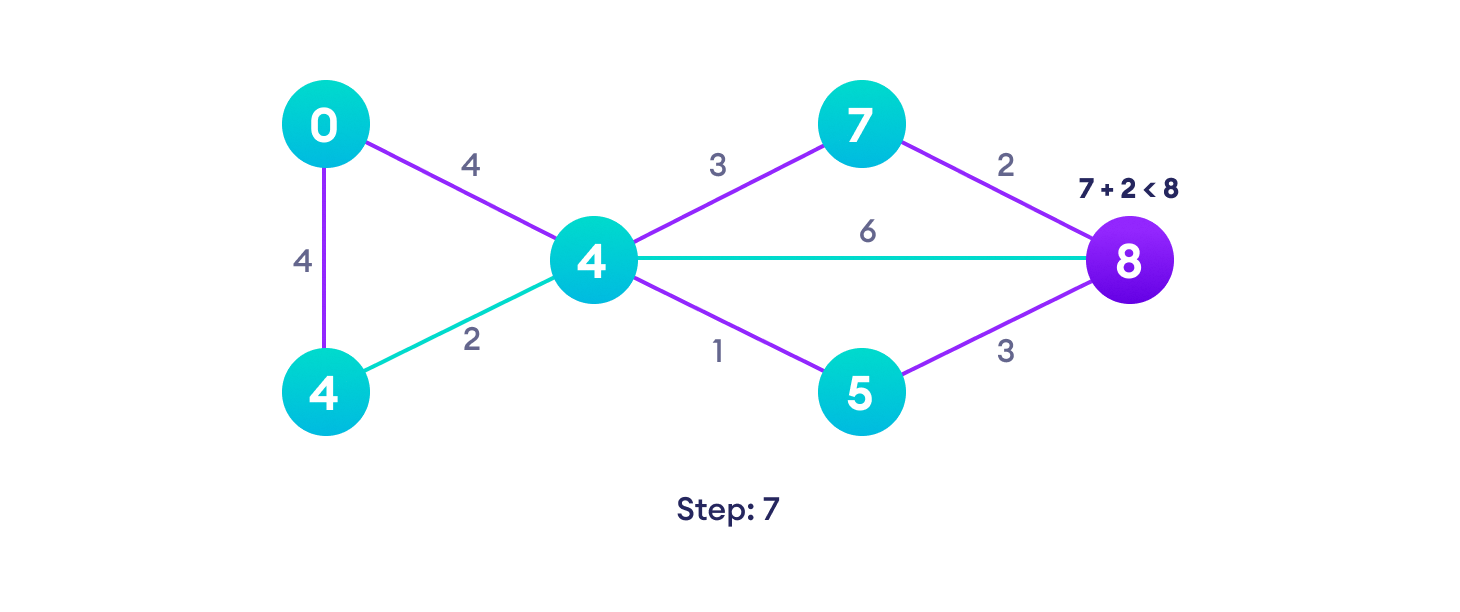


**If the path length of the adjacent vertex is lesser than new path length, don't update it**

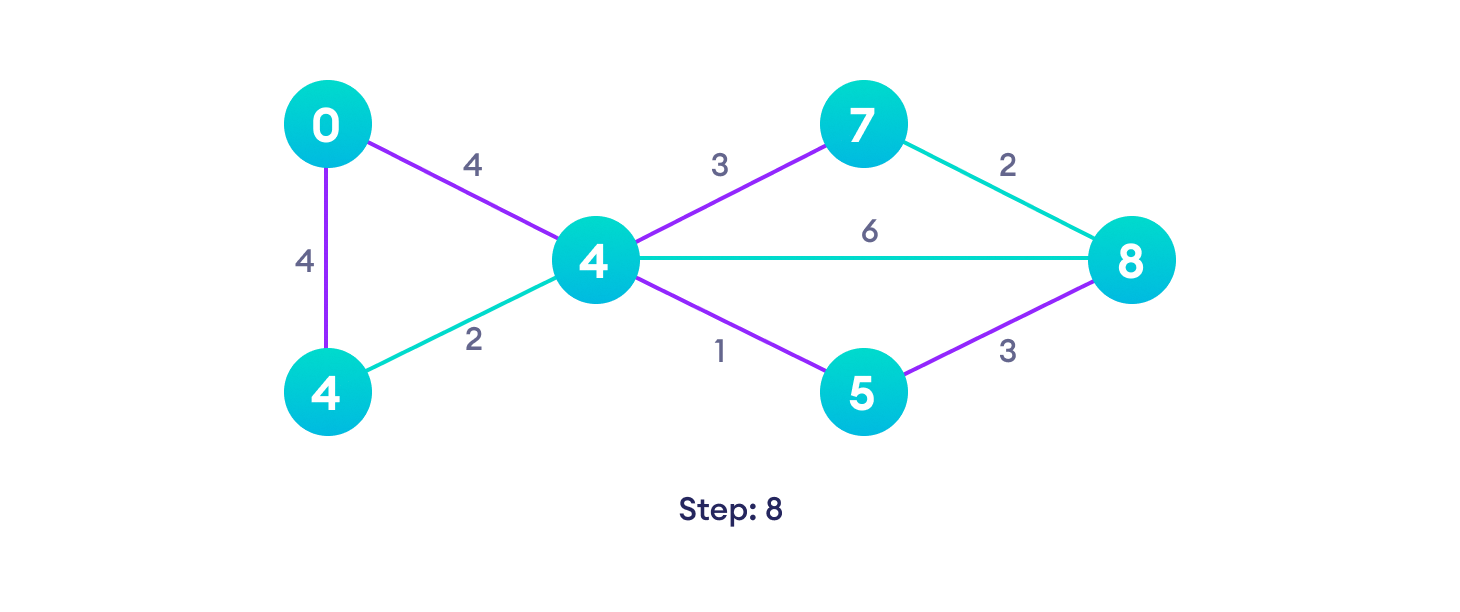
**Avoid updating path lengths of already visited vertices**



**After each iteration, we pick the unvisited vertex with the least path length. So**

**we choose 5 before 7**

**Notice how the rightmost vertex has its path length updated twice**



**Repeat until all the vertices have been visited**

**Practical Implementation Djikstra’s code algorithm**

import sys

*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 \tDistance from Source"

        for node in range(*self*.V):

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

*def* minDistance(*self*, *dist*, *sptSet*):

        min = sys.maxint

        for u in range(*self*.V):

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

                min = dist[u]

                min\_index = u

        return min\_index

*def* dijkstra(*self*, *src*):

        dist = [sys.maxint] \* *self*.V

        dist[src] = 0

        sptSet = [False] \* *self*.V

        for cout in range(*self*.V):

            x = *self*.minDistance(dist, sptSet)

            sptSet[x] = True

            for y in range(*self*.V):

                if *self*.graph[x][y] > 0 and sptSet[y] == False and \

                dist[y] > dist[x] + *self*.graph[x][y]:

                        dist[y] = dist[x] + *self*.graph[x][y]

*self*.printSolution(dist)

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)*;*

**Output:**

Vertex   Distance from Source

0                0

1                4

2                12

3                19

4                21

5                11

6                9

7                8

8                14

Time Complexity of the implementation is O(V^2). If the input graph is represented using adjacency list, it can be reduced to O(E log V) with the help of a binary heap.

**Conclusion:** Successfully Implemented the Dijkstra’s algorithm.