Problem Statement:

Consider two cities i.e Belgaum and Karwar or Goa as source and destination nodes.Propose shortest path or route using various Distance vector routing algorithms and also calculate the time complexity for all the algorithms used. Conclude the work by mentioning the best algorithm for the above problem defined with supported tabulation.Implementation can be done using any programming language. Results must beincorporated into Project report with various inputs.

Objectives:

1. Understand the working of different distance routing algorithms.
2. Learn to compute time complexities.
3. Learn to build graphs and equivalent Matrices.

Theory:

The Distance vector algorithm is iterative, asynchronous and distributed.

- ➢ **Distributed:** It is distributed in that each node receives information from one or more of its directly attached neighbors, performs calculation and then distributes the result back to its neighbors.
- ➢ **Iterative:** It is iterative in that its process continues until no more information is available to be exchanged between neighbors.
- ➢ **Asynchronous:** It does not require that all of its nodes operate in the lock step with each other.

The Distance vector algorithm is a dynamic algorithm. It is mainly used in ARPANET, and RIP. Each router maintains a distance table known as Vector.

- ❖ Three Keys to understand the working of Distance Vector Routing Algorithm:
- ☐ **Knowledge about the whole network:** Each router shares its knowledge through the entire network. The Router sends its collected knowledge about the network to its neighbors.
- ☐ **Routing only to neighbors:** The router sends its knowledge about the network to only those routers which have direct links. The router sends whatever it has about the network through the ports. The information is received by the router and uses the information to update its own routing table.
- ☐ **Information sharing at regular intervals:** Within 30 seconds, the router sends the information to the neighboring routers.

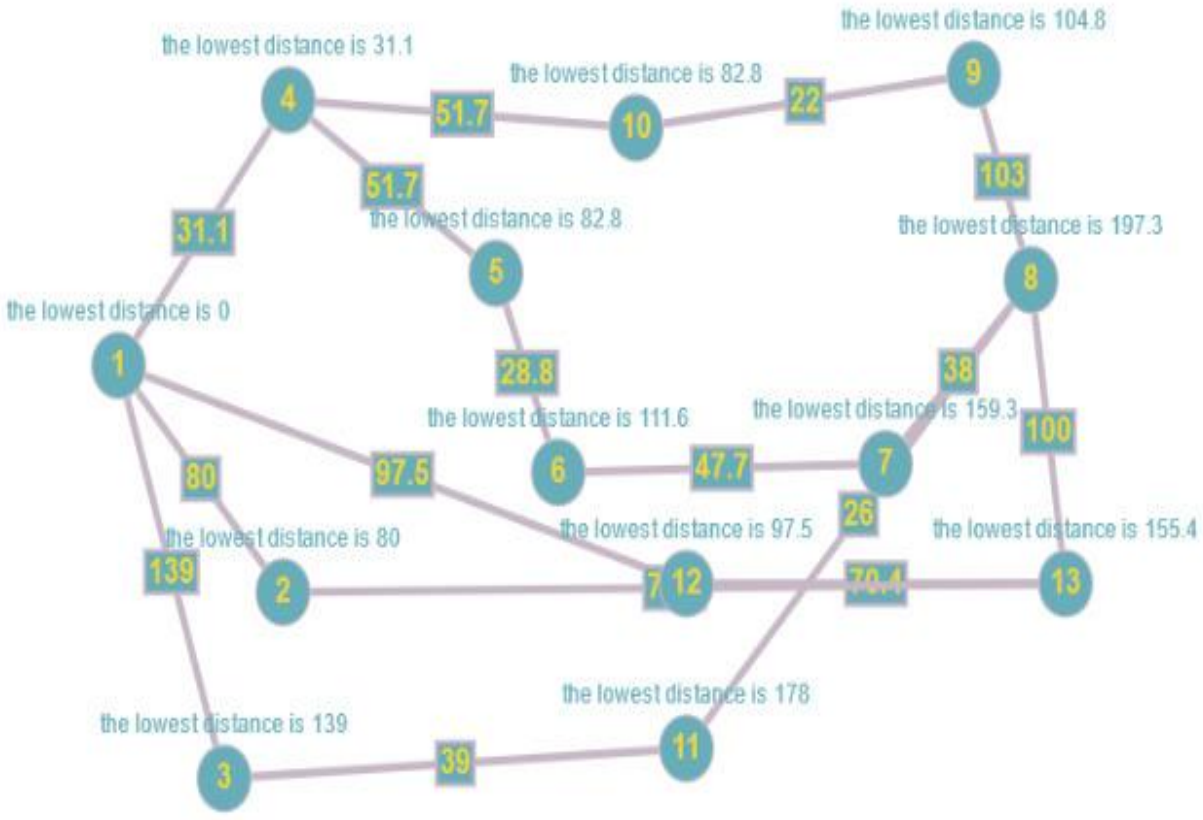
In this project, We'll be implementing different DVR algorithms like Floyd-Warshall, BellmanFord, Dijkstra's algorithm etc. for finding minimum/shortest path considering Source as **Belgaum** & Destination as **Karwar** and will compare the time efficiencies.

Graph Built:

1. Belgaum 2. Dharwad 3. Madgaon 4. Khanapur 5. Ramnagar 6. Joida 7. Kadra

8. Karwar 9. Dandeli 10. Haliyal 11. Canacona 12. Hubli 13. Yellapur.

🔵~> Node (city) 🟦~> weight (distance)



Matrix For Above Graph:

999 ~> INF (direct path not available)

```
#BUILDING MATRIX

Graph=[[0,80,139,31.1,999,999,999,999,999,999,999,97.5,999],
[80,0,999,999,999,999,999,999,999,999,999,999,75.4],
[139,999,0,999,999,999,999,999,999,999,39,999,999],
[31.1,999,999,0,51.7,999,999,999,999,51.7,999,999],
[999,999,999,51.7,0,28.8,999,999,999,999,999,999,999],
[999,999,999,999,28.8,0,47.7,999,999,999,999,999,999],
[999,999,999,999,999,47.7,0,38,999,999,999,999,999],
[999,999,999,999,999,999,38,0,109,999,26,999,100],
[999,999,999,999,999,999,999,103,0,22,999,999,999],
[99,999,999,51.7,999,999,999,999,22,0,999,999,999],
[999,999,39,999,999,999,999,26,999,999,0,999,999],
[97.5,999,999,999,999,999,999,999,999,999,0,70.4],
[999,75.4,999,999,999,999,999,100,999,999,999,70.4,0]]
```

Algorithms:

1. Bellman-Ford Algorithm:

Given a graph and a source vertex *src* in graph, find shortest paths from *src* to all vertices in the given graph. The graph may contain negative weight edges.

```
]: # Bellman Ford Algorithm


class BGraph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []
    # Add edges
    def add_edge(self, s, d, w):
        self.graph.append([s, d, w])

    def print_solution(self, dist):
        print("Vertex Distance from Source")
        for i in range(self.V):
            print("{0}\t\t{1}".format(i, dist[i]))
    def bellman_ford(self, src):
        dist = [999] * self.V
        dist[src] = 0
        for _ in range(self.V - 1):
            for s, d, w in self.graph:
                if dist[s] != 999 and dist[s] + w < dist[d]:
                    dist[d] = dist[s] + w
        for s, d, w in self.graph:
            if dist[s] != 999 and dist[s] + w < dist[d]:
                print("Graph contains negative weight cycle")
                return
        self.print_solution(dist)
G=BGraph(13)
for i in range(13):
    for j in range(i+1,12):
        Q=Graph[i][j]
        G.add_edge(i,j,Q)
G.bellman_ford(0)

#SOURCE~> Belgaum (0)
#DESTINATION~> Karwar (7)
```

```
Vertex Distance from Source
0               0
1               80
2               139
3               31.1
4               82.80000000000001
5               111.60000000000001
6               159.3
7               197.3
8               306.3
9               82.80000000000001
10              178
11              97.5
12              999
```

2. Dijkstra's Algorithm:

Dijkstra's algorithm is a single-source shortest path algorithm. Here, single-source means that only one source is given, and we have to find the shortest path from the source to all the nodes.

```
]: #Dijsktra's Algorithm

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+1, "\t", round(dist[node],2))
    def minDistance(self, dist, sptSet):
        min = 100000
        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 = [100000] * 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(12)
g.graph = Graph
g.dijkstra(0);

#SOURCE~> Belgaum (1)
#DESTINATION~> Karwar (8)
```

```
Vertex  Distance from Source
1        0
2        80
3        139
4        31.1
5        82.8
6        111.6
7        159.3
8        197.3
9        104.8
10       82.8
11       178
12       97.5
```

3. Floyd-Warshall's Algorithm:

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. Uses Dynamic Programming approach.

```
: #Floyd Warshal's Algorithm
INF=999
V = len(Graph)-1
def floydWarshall(graph,m,n):

    dist=graph
    for k in range(V):
        for i in range(V):
            for j in range(V):
                dist[i][j]=round(min(dist[i][j],dist[i][k]+dist[k][j]),2)
    source=m;dest=n
    print("{} ~>{} is => {}".format(m+1,n+1,dist[n][m]))
floydWarshall(Graph,0,7)

#SOURCE~> Belgaum (1)
#DESTINATION~> Karwar (8)
```

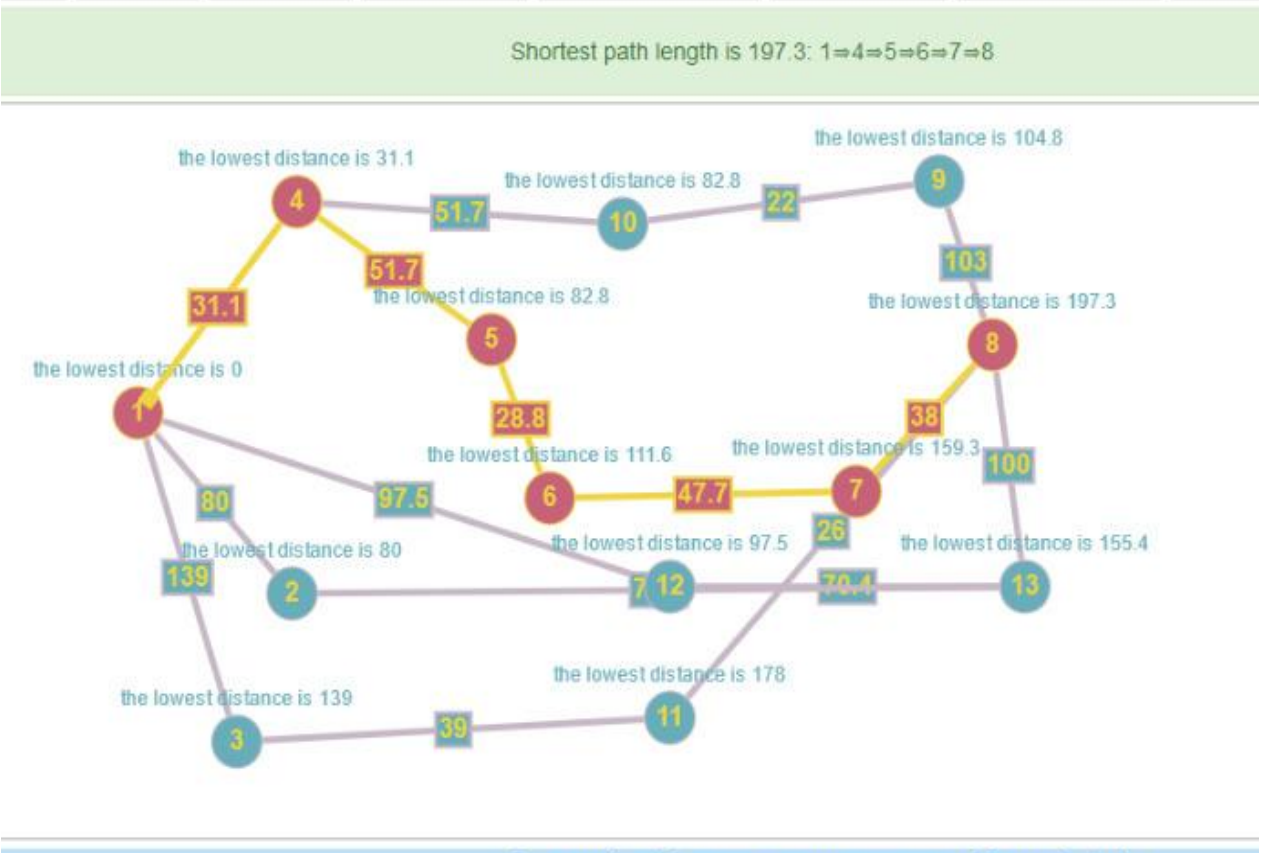
```
1 ~>8 is => 197.3
```

Complexity of Algorithm:

For a given graph (Graph) with Vertices V=13 & Edges E=16

ALGORITHM	TIME COMPLEXITY	SPACE COMPLEXITY
Bellman Ford	O(V*E) ~ 208	O(E) ~ 16
Dijkstra's Algorithm	***O(V^2) ~ 169***	***O(E) ~ 16***
Floyd-Warshal	O(V^3) ~ 2169	O(V^2) ~ 169

Path Visualization from Dijkstra's Algorithm:



Outcomes:

From this project, we understood the working of some of the routing algorithms, their time complexities and learnt to build distance graphs and convert it to matrices.

Conclusion:

Following the Complexity table, We conclude that - Dijkstra's Algorithm is the most efficient algorithm for solving the given problem.

References:

1. https://www.programiz.com/dsa/bellman-ford-algorithm
2. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
3. https://www.javatpoint.com/distance-vector-routing-algorithm