23CCE201 Data Structures

Name : Tarun Sri Vathsan K

Roll No : CB.EN.U4CCE23058

AIM :

- To implement various shortest path problems and solutions to weighted directed graphs.

LOGIC :

Dijkstra:

The logic of Dijkstra's algorithm revolves around a systematic approach to finding the shortest path in a weighted graph. It begins by initializing the distance to the source vertex as 0 while setting the distances to all other vertices as infinity. A priority queue is used to efficiently select the next vertex with the smallest known distance for processing. The algorithm then repeatedly extracts the vertex with the shortest distance from the queue, marks it as visited, and examines its unvisited neighbors. For each neighbor, it calculates a potential new path distance through the current vertex. If this new distance is shorter than the previously recorded distance for that neighbor, the algorithm updates the neighbor's distance and enqueues it with the updated value. This process repeats until all vertices are visited or the priority queue is empty, ensuring that the shortest distance to each vertex is finalized once it is visited.

Floyd:

The logic of Floyd's algorithm focuses on finding the shortest paths between all pairs of vertices in a weighted graph. It works by incrementally improving the distance matrix through a series of updates. Initially, the algorithm sets up a distance matrix where the entry at d[i][j] represents the direct distance between vertices i and j if an edge exists, or infinity if no edge connects them directly. The distance from any vertex to itself is initialized to 0. The algorithm then iteratively considers each vertex as an intermediate vertex and updates the distances between all pairs of vertices. For each pair of vertices (i,j), it checks if the path i→k→j (where k is the intermediate vertex being considered) provides a shorter path than the current known distance d[i][j]. If true, d[i][j] is updated to reflect the shortest path.

Bellman-Ford:

The logic of the Bellman-Ford algorithm is based on the principle of edge relaxation, which allows the algorithm to iteratively update the shortest path estimates from a source vertex to all other vertices in a weighted graph, even if negative weights are present. Initially, the algorithm sets the distance from the source vertex to itself as distance[source]=0. while all other vertices are initialized to infinity, indicating they are unreachable. The core operation involves examining each edge (u,v) with weight w and applying the relaxation formula

Distance[v] =min(distance[v],distance[u]+w)

This formula checks if the current known distance to vertex v can be improved by taking the edge from u to v. The relaxation process is repeated for **N - 1,** here N is the total no.of vertices.

ALGORITHM :

Dijkstra:

1. Get the total number of vertices, edges, neighbors, weights and source vertex from the user.
2. Initialize a dictionary shortest to store the shortest known distances from the source to each vertex, setting the source distance to 0 and all other vertices to infinity.
3. Create an empty set visited to keep track of nodes that have been processed.
4. Repeat the following until all vertices are visited:
5. Find the unvisited vertex with the smallest known distance (set it as curr).
6. If no such vertex exists, break out of the loop.

7. Add curr to the visited set.
8. Iterate through each neighbor of curr:
9. Calculate the new path distance by adding the weight of the edge from curr to the neighbor.
10. If this new path distance is less than the current recorded distance for the neighbor, update the shortest dictionary with the new distance.
11. Return the shortest dictionary containing the shortest paths from the source to all other vertices.

Floyd :

1. Get the total number of vertices, edges, neighbors, weights and source vertex from the user.
2. Initialize a distance matrix dist where each entry d[i][j] represents the distance from vertex i to vertex j. Set all entries to infinity to indicate no direct path exists, except for the diagonal entries d[i][i] which are set to 0, representing the distance from each vertex to itself.
3. Populate the distance matrix with initial distances based on the edges defined in the input graph. For each edge (u,v) with weight w, set d[u][v]=w.
4. Repeat the following steps for each vertex k in the graph, treating k as an intermediate vertex:
    a. For each vertex i in the graph:
        i. For each vertex j in the graph:
            1. Check if the distance from i to j can be improved by going through vertex k. If d[i][j]>d[i][k]+d[k][j], update d[i][j] to d[i][k]+d[k][j].
5. Continue the iteration until all vertices have been considered as intermediate vertices, ensuring that all possible paths have been evaluated.
6. Once all iterations are complete, the distance matrix will contain the shortest path distances between all pairs of vertices.

7. Return the completed distance matrix as the output, where each entry d[i][j] represents the shortest distance from vertex i to vertex j.

Bellman-Ford:

1. Get the total number of vertices, edges, neighbors, weights and source vertex from the user.
2. Initialize a distances dictionary with all vertices set to infinity, except the destination vertex, which is set to 0.
3. Repeat the following process for V−1 times, where V is the number of vertices:
4. For each vertex in the graph:
5. For each connected neighbor and its edge weight:
6. If the current distance to the vertex plus the edge weight is less than the known distance to the neighbor, update the distance to the neighbor.
7. Check for negative-weight cycles by iterating over all edges:
8. If any distance can still be relaxed (i.e., a shorter path is found), a negative-weight cycle exists.
9. Return or print the distances dictionary showing the shortest paths, or indicate if vertices remain unreachable or if a negative-weight cycle is present.

CODE :

1. Dijkstra

```
def dijkstra(graph,source):
    shortest = {vertex:float('inf') for vertex in graph}
    shortest[source] = 0
    visited = set()
    while len(visited) < len(graph):
        curr = None
        curr_distance = float('inf')
        for vertex in graph:
            if vertex not in visited and shortest[vertex] < curr_distance:
                curr = vertex
                curr_distance = shortest[vertex]
            if curr == None:
                 break

        visited.add(curr)
        for neighbour,weight in graph[curr]:
            distance = curr_distance + weight
            if distance < shortest[neighbour]:
                shortest[neighbour] = distance
        return shortest
def user_input():
    graph = {}
    n = int(input("Enter the number of vertices: "))
    for i in range(n):
        vertex = input("\nEnter the vertex: ")
        neighbours = []
        m = int(input(f"Enter the number of edges for {vertex}: "))
        for i in range(m):
            neighbour = input(f"Enter neighbour of the {vertex}: ")
            weight = int(input(f"Enter weight for edge {vertex} -> {neighbour}: "))
            neighbours.append((neighbour,weight))
        graph[vertex] = neighbours
     return graph
```

```
def print_graph(graph):
    print("\nGraph Representation:")
    for vertex, neighbours in graph.items():
        for neighbour, weight in neighbours:
            print(f"{vertex} -> {neighbour} (weight: {weight})")

graph = user_input()
print_graph(graph)
source = input("Enter the source node: ")
shortest = dijkstra(graph, source)
```

2.Floyd

```
def floyd_warshall(graph, vertices):
    # Initialize distance matrix with infinity
    dist = {v: {u: float('inf') for u in vertices} for v in vertices}

    # Set the distance from each vertex to itself as 0
    for v in vertices:
        dist[v][v] = 0

    # Populate initial distances based on the given graph edges
    for vertex, neighbors in graph.items():
        for neighbor, weight in neighbors:
            dist[vertex][neighbor] = weight
    # Floyd-Warshall algorithm
    for k in vertices:
        for i in vertices:
            for j in vertices:
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

def user_input():
    graph = {}
    n = int(input("Enter the number of vertices: "))
    vertices = []

    for i in range(n):
        vertex = input("\nEnter the vertex: ")
        vertices.append(vertex)
        neighbors = []
        m = int(input(f"Enter the number of edges for {vertex}: "))

        for j in range(m):
            neighbor = input(f"Enter neighbor of {vertex}: ")
            weight = int(input(f"Enter weight for edge {vertex} -> {neighbor}: "))
            neighbors.append((neighbor, weight))
        graph[vertex] = neighbors

    return graph, vertices

def print_distances(dist, vertices):
    print("\nAll-Pairs Shortest Paths:")
    for i in vertices:
        for j in vertices:
            if dist[i][j] == float('inf'):
                print(f"{i} -> {j}: No Path")
            else:
                print(f"{i} -> {j}: {dist[i][j]}")

graph, vertices = user_input()
distances = floyd_warshall(graph, vertices)
print_distances(distances, vertices)
```

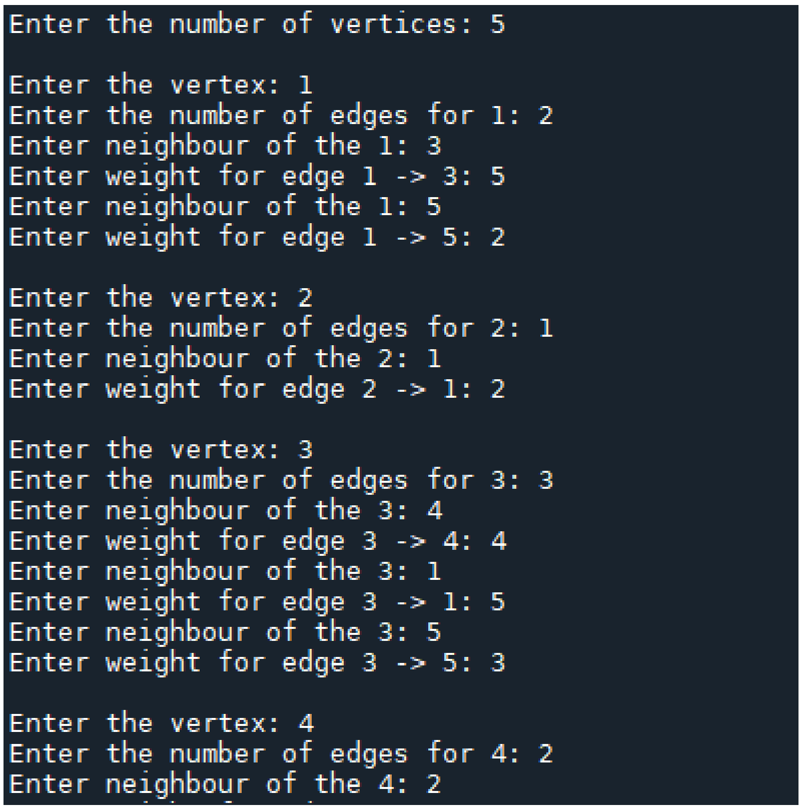
3.Bellman-Ford

```
from collections import defaultdict

def bellman_ford(graph, vertices, destination):
    distances = {vertex: float('inf') for vertex in vertices}
    distances[destination] = 0
    for _ in range(len(vertices) - 1):
        for vertex in graph:
            for neighbour, weight in graph[vertex]:
                if distances[vertex] != float('inf') and distances[vertex] + weight < 
distances[neighbour]:
```

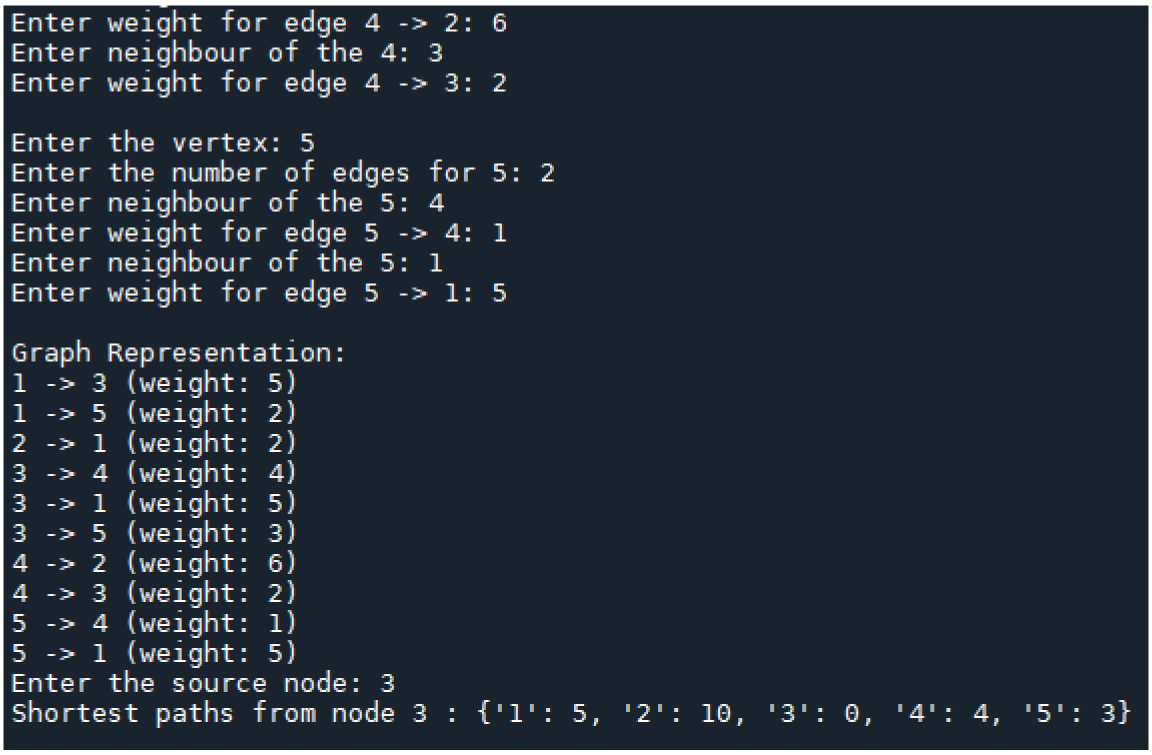
```
                    distances[neighbour] = distances[vertex] + weight

    for vertex in vertices:
        if distances[vertex] == float('inf'):
            print(f"The vertex {vertex} is unreachable from the destination vertex {destination}.")

    return distances

def reverse_graph(graph):
    reversed_graph = defaultdict(list)
    for vertex, neighbours in graph.items():
        for neighbour, weight in neighbours:
            reversed_graph[neighbour].append((vertex, weight))
    return reversed_graph

def user_input():
    graph = defaultdict(list)
    vertices = set()
    n = int(input("Enter the number of vertices: "))
    for _ in range(n):
        vertex = input("Enter the vertex: ")
        vertices.add(vertex)
        m = int(input(f"Enter the number of edges for vertex {vertex}: "))
        for _ in range(m):
            neighbour = input(f"Enter the neighbour for vertex {vertex}: ")
            weight = int(input(f"Enter the weight for the edge {vertex} -> {neighbour}: "))
            graph[vertex].append((neighbour, weight))
            vertices.add(neighbour)
    return graph, vertices


def display_distances(distances, destination):
    print(f"\nShortest distances to the destination ({destination}):")
    for vertex, distance in distances.items():
        if distance == float('inf'):
            print(f"Vertex {vertex} is unreachable.")
        else:
            print(f"Distance to vertex {vertex}: {distance}")


graph, vertices = user_input()
reversed_graph = reverse_graph(graph)
destination = input("\nEnter the destination vertex: ")
distances = bellman_ford(reversed_graph, vertices, destination)
display_distances(distances, destination)
```

RESULTS :

Dijkstra

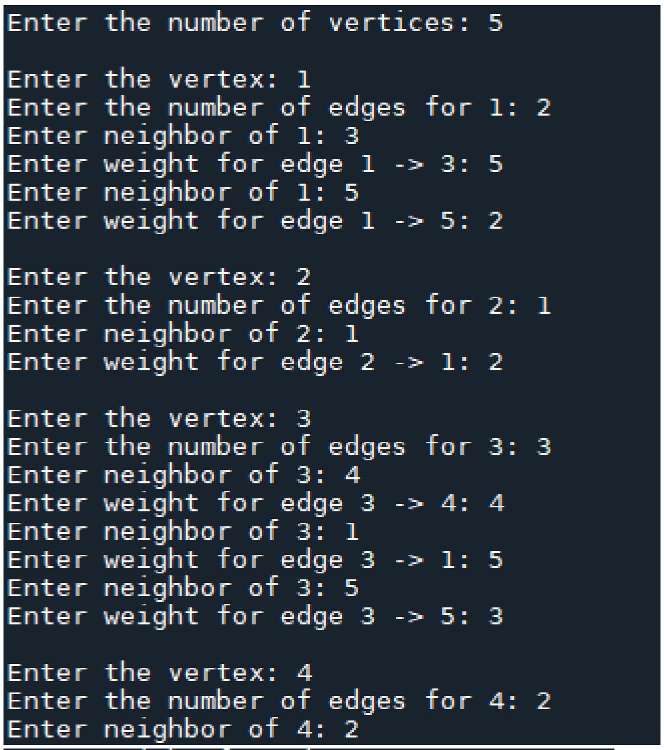


```
Enter the number of vertices: 5

Enter the vertex: 1
Enter the number of edges for 1: 2
Enter neighbour of the 1: 3
Enter weight for edge 1 -> 3: 5
Enter neighbour of the 1: 5
Enter weight for edge 1 -> 5: 2

Enter the vertex: 2
Enter the number of edges for 2: 1
Enter neighbour of the 2: 1
Enter weight for edge 2 -> 1: 2

Enter the vertex: 3
Enter the number of edges for 3: 3
Enter neighbour of the 3: 4
Enter weight for edge 3 -> 4: 4
Enter neighbour of the 3: 1
Enter weight for edge 3 -> 1: 5
Enter neighbour of the 3: 5
Enter weight for edge 3 -> 5: 3

Enter the vertex: 4
Enter the number of edges for 4: 2
Enter neighbour of the 4: 2
```

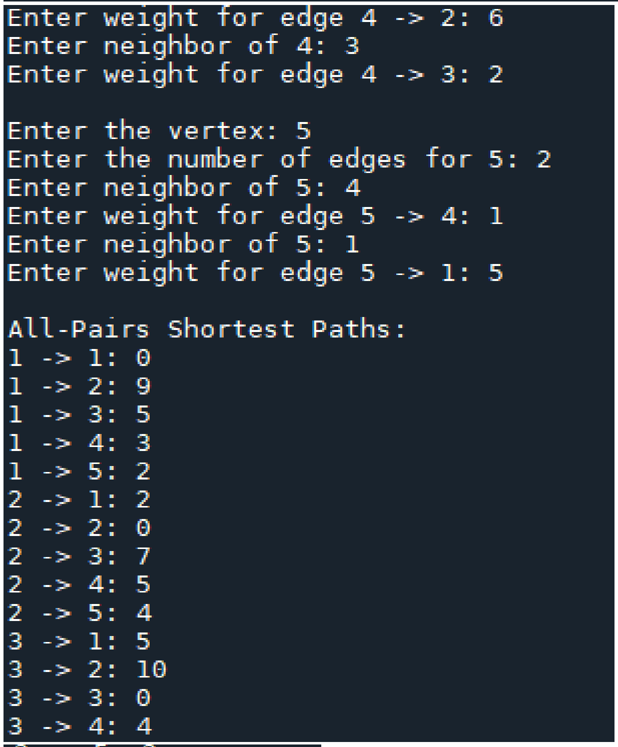


```
Enter weight for edge 4 -> 2: 6
Enter neighbour of the 4: 3
Enter weight for edge 4 -> 3: 2

Enter the vertex: 5
Enter the number of edges for 5: 2
Enter neighbour of the 5: 4
Enter weight for edge 5 -> 4: 1
Enter neighbour of the 5: 1
Enter weight for edge 5 -> 1: 5

Graph Representation:
1 -> 3 (weight: 5)
1 -> 5 (weight: 2)
2 -> 1 (weight: 2)
3 -> 4 (weight: 4)
3 -> 1 (weight: 5)
3 -> 5 (weight: 3)
4 -> 2 (weight: 6)
4 -> 3 (weight: 2)
5 -> 4 (weight: 1)
5 -> 1 (weight: 5)
Enter the source node: 3
Shortest paths from node 3 : {'1': 5, '2': 10, '3': 0, '4': 4, '5': 3}
```

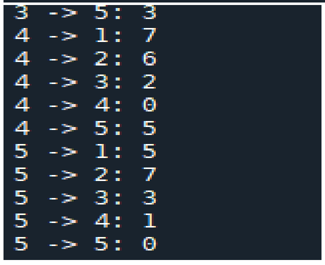
Floyd



```
Enter the number of vertices: 5

Enter the vertex: 1
Enter the number of edges for 1: 2
Enter neighbor of 1: 3
Enter weight for edge 1 -> 3: 5
Enter neighbor of 1: 5
Enter weight for edge 1 -> 5: 2

Enter the vertex: 2
Enter the number of edges for 2: 1
Enter neighbor of 2: 1
Enter weight for edge 2 -> 1: 2

Enter the vertex: 3
Enter the number of edges for 3: 3
Enter neighbor of 3: 4
Enter weight for edge 3 -> 4: 4
Enter neighbor of 3: 1
Enter weight for edge 3 -> 1: 5
Enter neighbor of 3: 5
Enter weight for edge 3 -> 5: 3

Enter the vertex: 4
Enter the number of edges for 4: 2
Enter neighbor of 4: 2
```

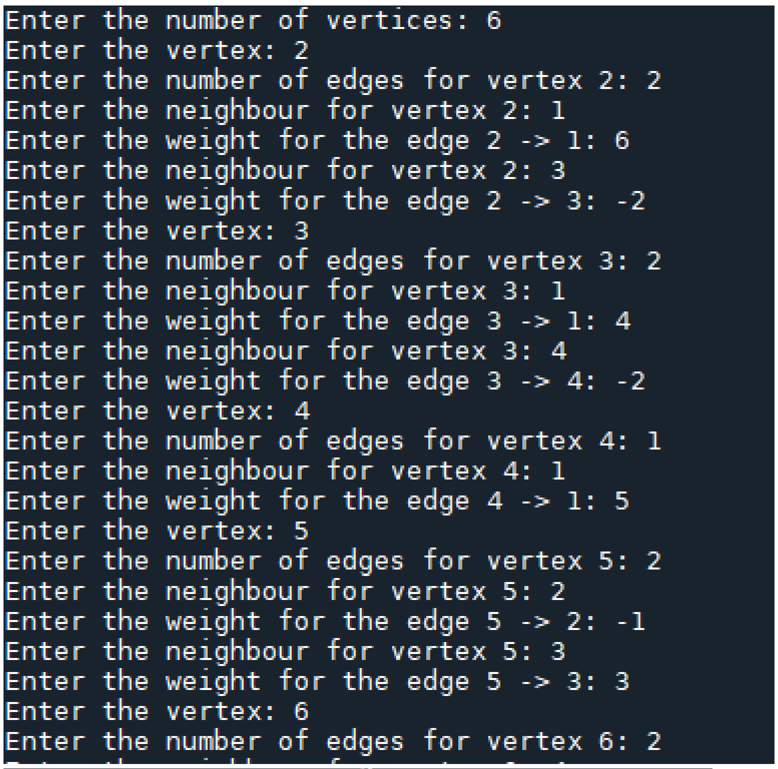


```
Enter weight for edge 4 -> 2: 6
Enter neighbor of 4: 3
Enter weight for edge 4 -> 3: 2

Enter the vertex: 5
Enter the number of edges for 5: 2
Enter neighbor of 5: 4
Enter weight for edge 5 -> 4: 1
Enter neighbor of 5: 1
Enter weight for edge 5 -> 1: 5

All-Pairs Shortest Paths:
1 -> 1: 0
1 -> 2: 9
1 -> 3: 5
1 -> 4: 3
1 -> 5: 2
2 -> 1: 2
2 -> 2: 0
2 -> 3: 7
2 -> 4: 5
2 -> 5: 4
3 -> 1: 5
3 -> 2: 10
3 -> 3: 0
3 -> 4: 4
```

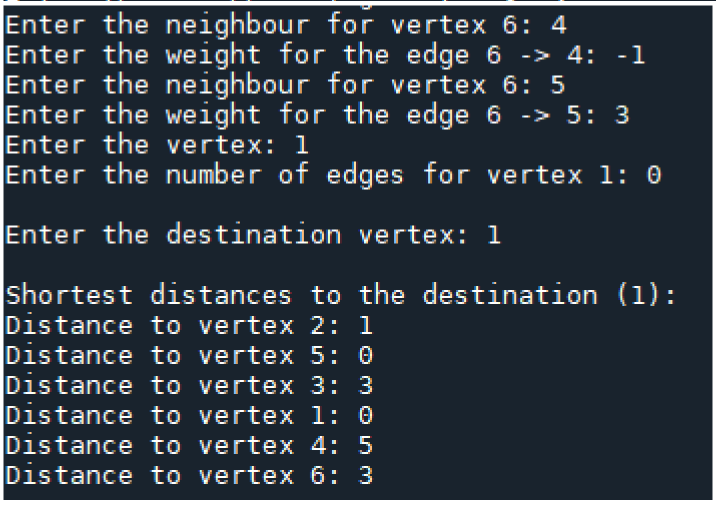


```
3 -> 5: 3
4 -> 1: 7
4 -> 2: 6
4 -> 3: 2
4 -> 4: 0
4 -> 5: 5
5 -> 1: 5
5 -> 2: 7
5 -> 3: 3
5 -> 4: 1
5 -> 5: 0
```

Bellman-Ford



```
Enter the number of vertices: 6
Enter the vertex: 2
Enter the number of edges for vertex 2: 2
Enter the neighbour for vertex 2: 1
Enter the weight for the edge 2 -> 1: 6
Enter the neighbour for vertex 2: 3
Enter the weight for the edge 2 -> 3: -2
Enter the vertex: 3
Enter the number of edges for vertex 3: 2
Enter the neighbour for vertex 3: 1
Enter the weight for the edge 3 -> 1: 4
Enter the neighbour for vertex 3: 4
Enter the weight for the edge 3 -> 4: -2
Enter the vertex: 4
Enter the number of edges for vertex 4: 1
Enter the neighbour for vertex 4: 1
Enter the weight for the edge 4 -> 1: 5
Enter the vertex: 5
Enter the number of edges for vertex 5: 2
Enter the neighbour for vertex 5: 2
Enter the weight for the edge 5 -> 2: -1
Enter the neighbour for vertex 5: 3
Enter the weight for the edge 5 -> 3: 3
Enter the vertex: 6
Enter the number of edges for vertex 6: 2
```



```
Enter the neighbour for vertex 6: 4
Enter the weight for the edge 6 -> 4: -1
Enter the neighbour for vertex 6: 5
Enter the weight for the edge 6 -> 5: 3
Enter the vertex: 1
Enter the number of edges for vertex 1: 0

Enter the destination vertex: 1

Shortest distances to the destination (1):
Distance to vertex 2: 1
Distance to vertex 5: 0
Distance to vertex 3: 3
Distance to vertex 1: 0
Distance to vertex 4: 5
Distance to vertex 6: 3
```

INFERENCES :

The Bellman-Ford, Dijkstra’s, and Floyd-Warshall algorithms each have their strengths for finding shortest paths in different types of graphs. Dijkstra’s algorithm is highly efficient for finding the shortest path in graphs with non-negative weights. However, it cannot handle graphs with negative weights, as this would lead to incorrect results. The Floyd-Warshall algorithm, which computes shortest paths between all pairs of vertices, is ideal when a complete set of shortest paths is needed for smaller or dense graphs. It can handle graphs with negative weights, as long as there are no negative-weight cycles. Bellman-Ford is particularly useful when dealing with graphs that may contain negative-weight edges because it can detect negative-weight cycles, making it reliable for scenarios where cycle detection is important.