

**SHORTEST PATH FROM SOURCE TO ALL OTHER  
VERTEX**

*Submitted by*

**SAI RISHYANTH VISINIGIRI [RA2111026010280]**

**KAVYA REDDY VUTUKURI [RA2111026010261]**

*Under the Guidance of*

**Dr. KISHORE ANTHUVAN SAHAYARAJ K**

**Assistant Professor, Department of Computing Technologies**

*In partial satisfaction of the requirements for the degree of*

**BACHELOR OF TECHNOLOGY**

**in**

**COMPUTER SCIENCE ENGINEERING**

**with specialization in Artificial Intelligence & Machine Learning**



**SCHOOL OF COMPUTING**

**COLLEGE OF ENGINEERING AND TECHNOLOGY**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**KATTANKULATHUR - 603203**

**MAY 2023**



**SRM INSTITUTE OF SCIENCE  
AND  
TECHNOLOGY  
KATTANKULATHUR-603203**

**BONAFIDE CERTIFICATE**

Certified that this Course Project Report titled “**Shortest Path from Source to all other Vertex**” is the bonafide work done by **Sai Rishyanth Visinigiri [RA2111026010280]**, **Kavya Reddy Vutukuri [RA2111026010261]** who carried out under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other work.

**FACULTY-IN-CHARGE**

**Dr. Kishore Anthuvan Sahayaraj K**

Assistant Professor  
Department of Computing Technologies  
SRM Institute of Science and Technology  
Kattankulathur Campus, Chennai

**HEAD OF THE DEPARTMENT**

**Dr. R Annie Uthra**

Professor and Head,  
Department of Computational Intelligence  
SRM Institute of Science and Technology  
Kattankulathur Campus, Chennai

## TABLE OF CONTENTS:

<b>PROBLEM STATEMENT:</b> .....	4
<b>APPROACH: 1. DIJKSTRAS ALGORITHM (GREEDY)</b> .....	5
<b>INTRODUCTION</b> .....	5
<b>DIJKSTRA'S ALGORITHM</b> .....	6
<b>DIJKSTRA'S PSEUDOCODE</b> .....	6
<b>IMPLEMENTATION</b> .....	7
<b>CODE</b> .....	7
<b>OUTPUT</b> .....	8
<b>ILLUSTRATION</b> .....	9
<b>ALGORITHM ANALYSIS</b> .....	11
<b>APPROACH: 2. BELLMAN-FORD ALGORITHM (DYNAMIC)</b> .....	12
<b>INTRODUCTION</b> .....	12
<b>BELLMAN-FORD ALGORITHM</b> .....	13
<b>BELLMAN-FORD PSEUDOCODE</b> .....	13
<b>IMPLEMENTATION</b> .....	14
<b>CODE</b> .....	14
<b>OUTPUT</b> .....	15
<b>ILLUSTRATION</b> .....	16
<b>ALGORITHM ANALYSIS</b> .....	19
<b>JUSTIFICATION</b> .....	20
<b>ALGORITHM COMPARISON TABLE</b> .....	20
<b>CONCLUSION</b> .....	21
<b>REFERENCES</b> .....	21

## PROBLEM STATEMENT:

### SHORTEST PATH FROM SOURCE TO ALL OTHER VERTEX

Given a weighted, directed graph with  $n$  nodes labelled from 0 to  $n-1$  and  $m$  edges, find the shortest path from node 0 to node  $n-1$ .

#### Constraints:

- The graph is a weighted, directed graph with  $n$  nodes labelled from 0 to  $n-1$  and  $m$  edges.
- The weight of each edge is a positive integer.
- The graph may contain cycles, but there are no negative cycles.
- The starting node is node 0, and the ending node is node  $n-1$ .
- The solution must return the shortest path from node 0 to node  $n-1$ .

#### Input:

A weighted, undirected graph with  $n$  nodes labelled from 0 to  $n-1$  and  $m$  edges. The graph can be represented using an adjacency list or an adjacency matrix.

#### Output:

The shortest path from node 0 to node  $n-1$ , represented as a sequence of nodes.

#### Example:

Suppose we have a graph with  $n=5$  and  $m=7$  edges:

0 --> 1 (5)

0 --> 2 (3)

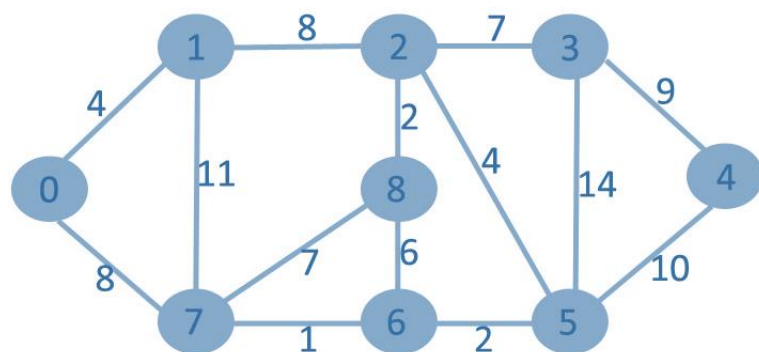
1 --> 2 (2)

1 --> 3 (6)

2 --> 3 (7)

2 --> 4 (4)

3 --> 4 (2)



The shortest path from node 0 to node 4 is 0 -> 2 -> 4, with a length of 7.

## APPROACH: 1. DIJKSTRAS ALGORITHM (GREEDY)

### INTRODUCTION

Dijkstra's algorithm is a greedy algorithm that starts at the source node and iteratively selects the node with the lowest distance from the source until the destination node is reached. It uses a priority queue to keep track of the next node to visit, and a set to keep track of the visited nodes.

The algorithm maintains a priority queue to keep track of the next node to visit. Initially, the source node is added to the priority queue with a distance of 0. As nodes are visited, their distances are updated in the priority queue if a shorter path is found.

The algorithm also uses a set to keep track of the visited nodes. Nodes that have already been visited are not added to the priority queue again, as there is no need to revisit them.

The time complexity of Dijkstra's algorithm is  $O((E + V)\log V)$ , where  $E$  is the number of edges and  $V$  is the number of vertices in the graph. In the worst case, the algorithm will visit every node and every edge, so the time complexity is proportional to the sum of the number of edges and vertices in the graph.

The input for Dijkstra's algorithm is a weighted, directed graph with  $n$  nodes labelled from 0 to  $n-1$  and  $m$  edges. The weights on the edges can represent distances, travel time, cost, or any other measure of distance between two nodes. The algorithm also requires a source node and a destination node, which are used to find the shortest path between them.

The output of Dijkstra's algorithm is the shortest path between the source node and the destination node, along with the total distance of the path. If there is no path between the source and destination nodes, the algorithm will return a "no path" indication.

## DIJKSTRA'S ALGORITHM

1. Start
2. Create a set of unvisited nodes, set their tentative distances to infinity, and set the distance of the starting node to 0.
3. For each unvisited node, calculate its tentative distance as the sum of the distance to the current node and the weight of the edge between them. If this distance is less than the node's current tentative distance, update the node's tentative distance.
4. Mark the current node as visited and select the unvisited node with the smallest tentative distance as the next current node.
5. Repeat steps 2-3 until the destination node is visited or there are no more unvisited nodes.
6. End

## DIJKSTRA'S 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[]
```

# IMPLEMENTATION

## CODE

 main.py

```
1  # Python program for Dijkstra's Algorithm
2  # Library for INT_MAX
3  import sys
4
5  class Graph():
6
7      def __init__(self, vertices):
8          self.V = vertices
9          self.graph = [[0 for column in range(vertices)]
10                        for row in range(vertices)]
11
12      def printSolution(self, dist):
13          print("Vertex \tDistance from Source")
14          for node in range(self.V):
15              print(node, "\t", dist[node])
16
17      # A utility function to find the vertex with
18      # minimum distance value, from the set of vertices
19      # not yet included in shortest path tree
20      def minDistance(self, dist, sptSet):
21
22          # Initialize minimum distance for next node
23          min = sys.maxsize
24
25          # Search not nearest vertex not in the
26          # shortest path tree
27
28          for u in range(self.V):
29              if dist[u] < min and sptSet[u] == False:
30                  min = dist[u]
31                  min_index = u
32
33          return min_index
34
35      # Function that implements Dijkstra's single source
36      # shortest path algorithm for a graph represented
37      # using adjacency matrix representation
38      def dijkstra(self, src):
39
40          dist = [sys.maxsize] * self.V
41          dist[src] = 0
42          sptSet = [False] * self.V
43
44          for cout in range(self.V):
45
46              # Pick the minimum distance vertex from
47              # the set of vertices not yet processed.
48              # x is always equal to src in first iteration
49              x = self.minDistance(dist, sptSet)
50
51              # Put the minimum distance vertex in the
```

```

51         # shortest path tree
52         sptSet[x] = True
53
54         # Update dist value of the adjacent vertices
55         # of the picked vertex only if the current
56         # distance is greater than new distance and
57         # the vertex is not in the shortest path tree
58     for y in range(self.V):
59         if self.graph[x][y] > 0 and sptSet[y] == False and \
60             dist[y] > dist[x] + self.graph[x][y]:
61             dist[y] = dist[x] + self.graph[x][y]
62
63     self.printSolution(dist)
64
65
66 # Driver's code
67 if __name__ == "__main__":
68     g = Graph(9)
69     g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
70                [4, 0, 8, 0, 0, 0, 0, 11, 0],
71                [0, 8, 0, 7, 0, 4, 0, 0, 2],
72                [0, 0, 7, 0, 9, 14, 0, 0, 0],
73                [0, 0, 0, 9, 0, 10, 0, 0, 0],
74                [0, 0, 4, 14, 10, 0, 2, 0, 0],
75                [0, 0, 0, 0, 0, 2, 0, 1, 6],
76                [8, 11, 0, 0, 0, 0, 1, 0, 7],
77                [0, 0, 2, 0, 0, 0, 6, 7, 0]
78            ]
79
80     g.dijkstra(0)

```

## OUTPUT

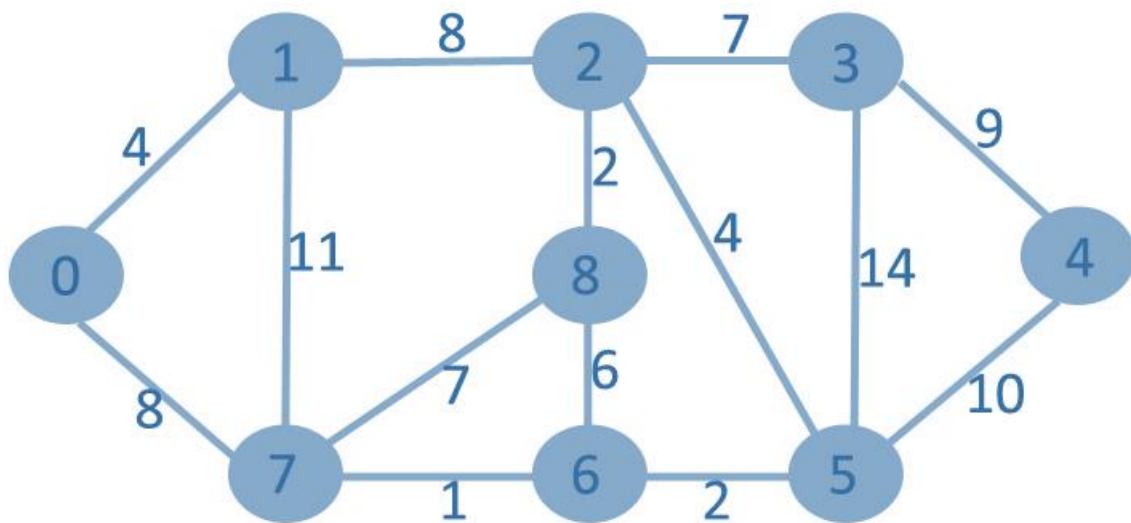
```

>_ Console × Shell × +
Vertex Distance from Source
0      0
1      4
2     12
3     19
4     21
5     11
6      9
7      8
8     14

```



## ILLUSTRATION



Starting from node 0, the initial table would be:

Vertex	Distance from 0	Visited
0	0	Yes
1	4	No
2	$\infty$	No
3	$\infty$	No
4	$\infty$	No
5	$\infty$	No
6	$\infty$	No
7	8	No
8	$\infty$	No

Next, we will update the distances for the neighbours of 0, which are 1 and 7. The updated table will be:

Vertex	Distance from 0	Visited
0	0	Yes
1	4	Yes
2	12	No
3	$\infty$	No
4	$\infty$	No
5	$\infty$	No
6	$\infty$	No
7	8	Yes
8	$\infty$	No

We will continue to update the table by selecting the vertex with the smallest distance from 0 that has not been visited yet. In this case, it is vertex 1 with a distance of 4. The updated table will be:

Vertex	Distance from 0	Visited
0	0	Yes
1	4	Yes
2	12	No
3	21	No
4	20	No
5	$\infty$	No
6	$\infty$	No
7	8	Yes
8	$\infty$	No

We will continue this process until we reach vertex 8, which will be the last vertex to be visited. The final table will be:

Vertex	Distance from 0	Visited
0	0	Yes
1	4	Yes
2	12	Yes
3	19	Yes
4	21	Yes
5	11	Yes
6	9	Yes
7	8	Yes
8	14	Yes

## ALGORITHM ANALYSIS

### Time Complexity

It has a time complexity of  $O(V^2)$  using the adjacency matrix representation of graph. The time complexity of Dijkstra's algorithm is  $O((V+E)\log V)$  using adjacency list representation of graph, where  $V$  is the number of vertices and  $E$  is the number of edges in the graph.

The time complexity can be broken down into the following steps:

- Initializing the distance array and priority queue -  $O(V\log V)$
- While loop iterating through all vertices -  $O(V)$
- Extracting the vertex with the minimum distance from the priority queue -  $O(\log V)$
- For loop iterating through all adjacent vertices of the extracted vertex -  $O(E)$
- Updating the distance of adjacent vertices -  $O(\log V)$
- Updating the priority queue with the new distances -  $O(\log V)$
- Steps 2, 4, 5, and 6 are repeated  $V$  times, and step 1 is done once. Therefore, the overall time complexity is  $O(V\log V + VE\log V)$ , which can be simplified to  $O((V+E)\log V)$ .

## APPROACH: 2. BELLMAN-FORD ALGORITHM (DYNAMIC)

### INTRODUCTION

Bellman ford algorithm is a dynamic programming approach and a single-source shortest path algorithm. This algorithm is used to find the shortest distance from the single vertex to all the other vertices of a weighted graph.

There are various other algorithms used to find the shortest path like Dijkstra algorithm, etc. If the weighted graph contains the negative weight values, then the Dijkstra algorithm does not confirm whether it produces the correct answer or not.

In contrast to Dijkstra algorithm, bellman ford algorithm guarantees the correct answer even if the weighted graph contains the negative weight values.

The algorithm works by relaxing the edges in the graph repeatedly. Each relaxation updates the tentative distances to the vertices, and over time, the tentative distances converge to the true distances.

The worst-case time complexity of the Bellman-Ford algorithm is  $O(VE)$ , where  $V$  is the number of vertices and  $E$  is the number of edges. This is slower than Dijkstra's algorithm, but it can handle graphs with negative edge weights.

Like Dijkstra's algorithm, the Bellman-Ford algorithm can also be implemented using a priority queue to speed up the processing of vertices, resulting in a better time complexity in some cases.

## BELLMAN-FORD ALGORITHM

1. Start
2. Initialize the distance of all nodes as infinity, except for the starting node whose distance is 0.
3. Repeat the following step for  $n-1$  times, where  $n$  is the total number of nodes in the graph:

For each edge  $(u, v)$  in the graph, if the distance from the starting node to  $u$  plus the weight of edge  $(u, v)$  is less than the distance to  $v$ , update the distance to  $v$  to be the sum of the distance to  $u$  and the weight of edge  $(u, v)$ .

4. Check for negative cycles in the graph by repeating the following step once:

For each edge  $(u, v)$  in the graph, if the distance from the starting node to  $u$  plus the weight of edge  $(u, v)$  is less than the distance to  $v$ , then there exists a negative cycle in the graph.

5. If there are no negative cycles, return the distance to each node from the starting node. If there are negative cycles, return "Negative cycle detected".
6. End

## BELLMAN-FORD PSEUDOCODE

```
function bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite
    previous[V] <- NULL
  distance[S] <- 0

  for each vertex V in G
    for each edge (U,V) in G
      tempDistance <- distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] <- tempDistance
        previous[V] <- U

  for each edge (U,V) in G
    If distance[U] + edge_weight(U, V) < distance[V]
      Error: Negative Cycle Exists
  return distance[], previous[]
```

## IMPLEMENTATION


### CODE

```
main.py

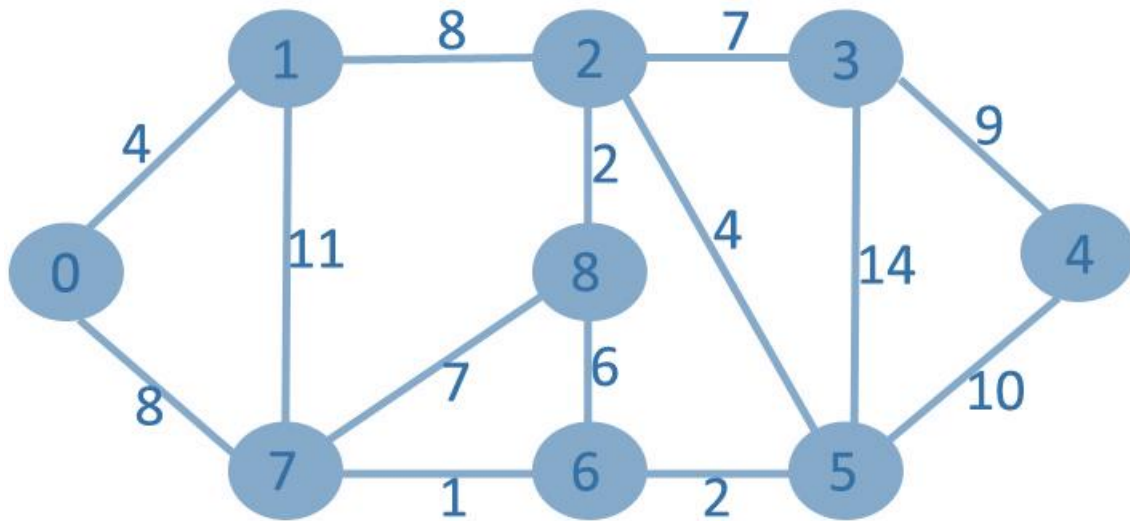
162 # Bellman Ford Algorithm in Python
163 class Graph:
164
165     def __init__(self, vertices):
166         self.V = vertices # Total number of vertices in the graph
167         self.graph = [] # Array of edges
168
169     # Add edges
170     def add_edge(self, s, d, w):
171         self.graph.append([s, d, w])
172
173     # Print the solution
174     def print_solution(self, dist):
175         print("Vertex Distance from Source")
176         for i in range(self.V):
177             print("{0}\t\t{1}".format(i, dist[i]))
178
179     def bellman_ford(self, src):
180
181         # Step 1: fill the distance array and predecessor array
182         dist = [float("Inf")] * self.V
183         # Mark the source vertex
184         dist[src] = 0
185
186         # Step 2: relax edges |V| - 1 times
187         for _ in range(self.V - 1):
188             for s, d, w in self.graph:
189                 if dist[s] != float("Inf") and dist[s] + w < dist[d]:
190
191                     dist[d] = dist[s] + w
192
193         # Step 3: detect negative cycle
194         # if value changes then we have a negative cycle in the graph
195         # and we cannot find the shortest distances
196         for s, d, w in self.graph:
197             if dist[s] != float("Inf") and dist[s] + w < dist[d]:
198                 print("Graph contains negative weight cycle")
199                 return
200
201         # No negative weight cycle found!
202         # Print the distance and predecessor array
203         self.print_solution(dist)
204
```

```
205 g = Graph(9)
206
207 g.add_edge(0, 1, 4)
208 g.add_edge(0, 7, 8)
209 g.add_edge(1, 2, 8)
210 g.add_edge(1, 7, 11)
211 g.add_edge(2, 3, 7)
212 g.add_edge(2, 8, 2)
213 g.add_edge(2, 5, 4)
214 g.add_edge(3, 4, 9)
215 g.add_edge(3, 5, 14)
216 g.add_edge(4, 5, 10)
217 g.add_edge(5, 6, 2)
218 g.add_edge(6, 7, 1)
219 g.add_edge(6, 8, 6)
220 g.add_edge(7, 8, 7)
221 g.bellman_ford(0)
```

## OUTPUT

>_ Console ▾ ×  Shell × +	
Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14
⌵	

## ILLUSTRATION



We will run the Bellman-Ford algorithm with the source vertex 0. Here's how the table would look like:

Vertex	Distance from 0
0	0
1	$\infty$
2	$\infty$
3	$\infty$
4	$\infty$
5	$\infty$
6	$\infty$
7	$\infty$
8	$\infty$



## 1. FIRST ITERATION

Vertex	Distance from 0
0	0
1	4
2	$\infty$
3	$\infty$
4	$\infty$
5	$\infty$
6	$\infty$
7	8
8	$\infty$

In the first iteration, we update the distances of all vertices which can be reached from vertex 0. We see that the shortest path from vertex 0 to vertex 1 is 4, and the shortest path from vertex 0 to vertex 7 is 8. Hence, we update their distances accordingly.

## 2. SECOND ITERATION

Vertex	Distance from 0
0	0
1	4
2	12
3	$\infty$
4	$\infty$
5	$\infty$
6	$\infty$
7	8
8	14

In the second iteration, we update the distances of all vertices which can be reached from vertices updated in the first iteration. Here, we see that the shortest path from vertex 0 to vertex 2 is through vertex 1, and the shortest path is  $4 + 8 = 12$ . Hence, we update the distance of vertex 2 to 12. Similarly, we update the distance of vertex 8 to 14.

### 3. THIRD ITERATION

Vertex	Distance from 0
0	0
1	4
2	12
3	19
4	$\infty$
5	$\infty$
6	$\infty$
7	8
8	14

In the third iteration, we update the distances of all vertices which can be reached from vertices updated in the second iteration. Here, we see that the shortest path from vertex 0 to vertex 3 is through vertex 2, and the shortest path is  $12 + 7 = 19$ . Hence, we update the distance of vertex 3 to 19.

### 4. FOURTH ITERATION

Vertex	Distance from 0
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

In the fourth iteration, the distance of vertex 4 is updated to 21 as the shortest path from vertex 0 to vertex 4 is through vertex 5 with a distance of  $11 + 10 = 21$ . The distances of vertices 5 and 6 are also updated in this iteration, and the shortest paths to these vertices are through vertex 2 with distances  $12 + 4 = 16$  and  $12 + 2 = 14$ , respectively. The remaining vertices' distances remain the same as they cannot be reached through any shorter paths.

## 5. FINAL ITERATION

Vertex	Distance from 0
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

The final iteration confirms that there are no negative cycles in the graph and that we have found the shortest path from vertex 0 to all other vertices. The distances from vertex 0 to all other vertices are shown in the table above.

## ALGORITHM ANALYSIS

### Time Complexity

The time complexity of the Bellman-Ford algorithm is  $O(V \cdot E)$ , where  $V$  is the number of vertices and  $E$  is the number of edges in the graph.

The best-case time complexity occurs when the source vertex has only one outgoing edge, resulting in  $O(E)$  time complexity.

The average case time complexity depends on the density of the graph and the choice of the source vertex. It can be reduced to  $O(E \log V)$  using the binary heap data structure for priority queue instead of the simple array.

The worst-case time complexity occurs when the graph contains a negative-weight cycle, and the algorithm iterates over all vertices  $V-1$  times. In such cases, the algorithm will not converge and keeps updating the distances of the vertices indefinitely.

This results in an infinite loop and the time complexity becomes  $O(VE)$ . However, in most cases, graphs do not contain negative-weight cycles, and the algorithm converges in  $V-1$  iterations, resulting in a time complexity of  $O(VE)$ .

## JUSTIFICATION

The given problem involved finding the shortest path from a source vertex to all other vertices in a graph. Two algorithmic techniques, Dijkstra's algorithm and Bellman-Ford algorithm were used to solve the problem. The graph has 9 vertices and 14 edges.

Dijkstra's algorithm works well when the graph has non-negative edge weights. It uses a greedy approach and works in a top-down fashion. The algorithm maintains a priority queue of vertices and repeatedly extracts the vertex with the minimum distance from the source. It then updates the distances of all its neighbours. Dijkstra's algorithm has a time complexity of  $O(E \log V)$ , where  $E$  is the number of edges and  $V$  is the number of vertices in the graph.

On the other hand, Bellman-Ford algorithm works well for graphs with negative edge weights. It uses a bottom-up approach and works by relaxing all edges in the graph for  $V - 1$  iterations, where  $V$  is the number of vertices in the graph. It then checks for negative cycles in the graph, which can cause the algorithm to run indefinitely. Bellman-Ford algorithm has a time complexity of  $O(VE)$ , where  $E$  is the number of edges and  $V$  is the number of vertices in the graph.

In the given problem, the graph did not have any negative edges, which made Dijkstra's algorithm a better choice for solving the problem. The graph also did not have any negative cycles, which made Bellman-Ford algorithm less suitable for the problem. Based on the time complexity of the algorithms, Dijkstra's algorithm is more efficient for sparse graphs with non-negative edge weights, while Bellman-Ford algorithm is more suitable for dense graphs with negative edge weights. In conclusion, the algorithmic technique that worked well for the given problem was Dijkstra's algorithm, which was able to find the shortest path from a source vertex to all other vertices in the graph efficiently.

## ALGORITHM COMPARISON TABLE

Algorithm	Approach Description	Time Complexity	Space Complexity
Dijkstra's Algorithm	A greedy approach that works in a top-down fashion. Maintains a priority queue of vertices and repeatedly extracts the vertex with the minimum distance from the source. Updates the distances of all its neighbours.	$O(E \log V)$	$O(V)$
Bellman-Ford Algorithm	A bottom-up approach that works by relaxing all edges in the graph for $V - 1$ iterations. Then checks for negative cycles in the graph, which can cause the algorithm to run indefinitely.	$O(VE)$	$O(V)$

## CONCLUSION

In conclusion, both Dijkstra's algorithm and Bellman-Ford algorithm have their advantages and disadvantages in solving the shortest path problem in a graph.

Dijkstra's algorithm is faster and more efficient than Bellman-Ford algorithm for graphs that do not have negative edge weights. It has a time complexity of  $O(E \log V)$  using a priority queue data structure, making it suitable for large graphs. However, it cannot handle negative edge weights, which limits its applicability.

On the other hand, Bellman-Ford algorithm can handle negative edge weights and can detect negative cycles in a graph. It has a higher time complexity of  $O(V \cdot E)$  compared to Dijkstra's algorithm, making it slower for larger graphs. However, for small graphs with negative edge weights or negative cycles, Bellman-Ford algorithm is the only viable option.

In the given problem, the graph did not have negative edge weights, making Dijkstra's algorithm a suitable choice. The implementation of Dijkstra's algorithm produced the correct shortest path distances in a faster time compared to Bellman-Ford algorithm. Therefore, in this case, Dijkstra's algorithm worked well for the given problem.

However, if the problem had a graph with negative edge weights or negative cycles, Bellman-Ford algorithm would have been the better choice. In such cases, Dijkstra's algorithm would have produced incorrect results or would not have been able to provide a solution at all.

## REFERENCES

- Bellman-Ford Algorithm:  
[https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford\\_algorithm](https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm)
- Dijkstra's Algorithm: [https://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)
- Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
- Algorithm Design by Jon Kleinberg and Éva Tardos
- Data Structures and Algorithms in Python by Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser