SHORTEST PATH FROM SOURCE TO ALL OTHER VERTEX

Submitted by

PRANAY JHA [RA2111003011092]

Under the Guidance of

MS. REVATHI.M

Assistant Professor, Department of Computing Technologies

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

MAY 2023

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

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 **PRANAY JHA [RA2111003011092]** 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	**HEAD OF THE DEPARTMENT**
Ms. REVATHI.M	**Dr. M Pushpa Latha**
Assistant Professor	Professor and Head,
Department of Computing Technologies	Department of Computing Technologies,
SRM Institute of Science and Technology	SRM Institute of Science and Technology
Kattankulathur Campus, Chennai	Kattankulathur Campus, Chennai

TABLE OF CONTENTS:

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. DIJIKSTRAS 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.

DIJIKSTRA’S ALGORTIHM

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.

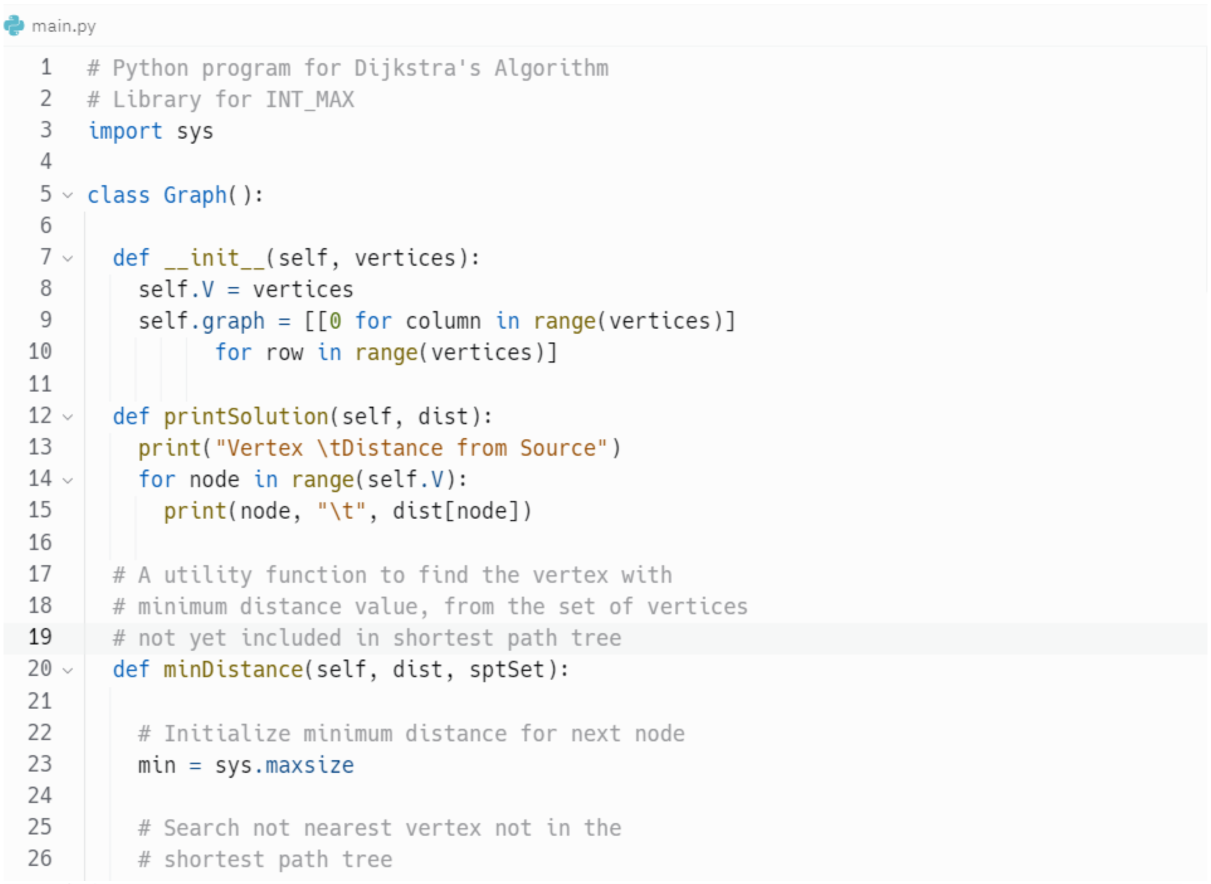
6. End

DIJIKSTRA’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[]
```

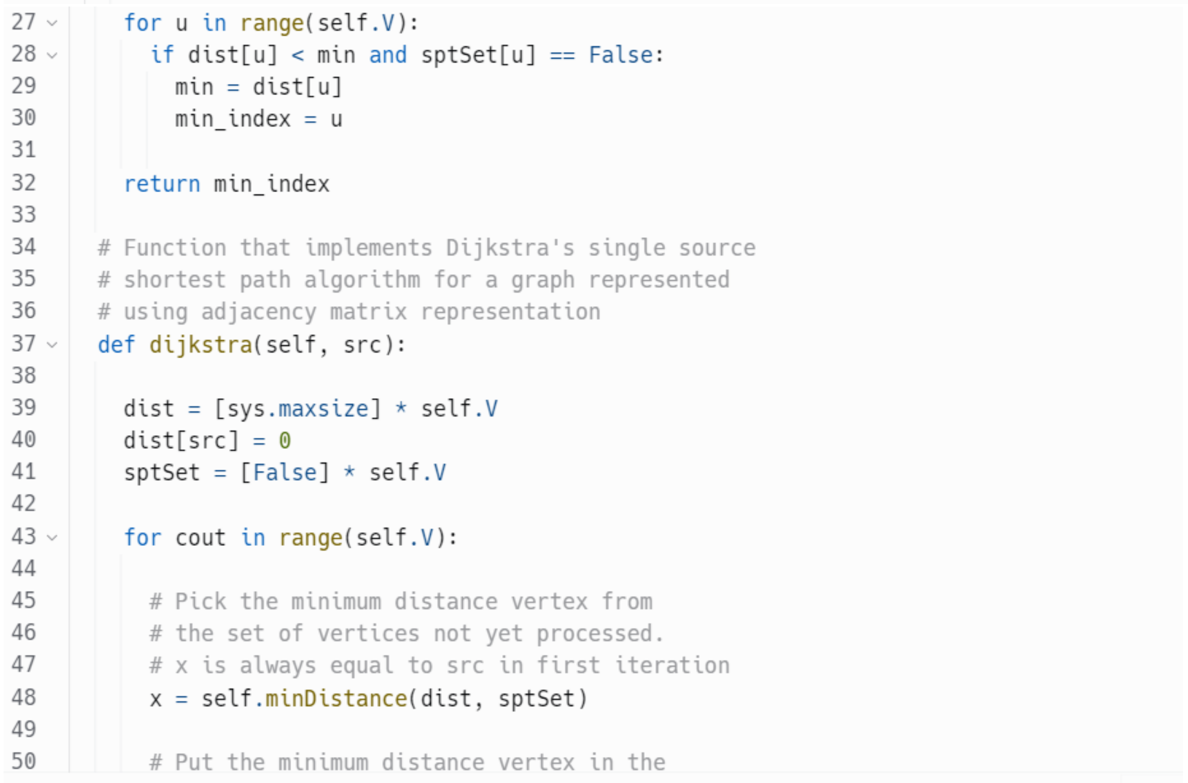
IMPLEMENTATION

CODE

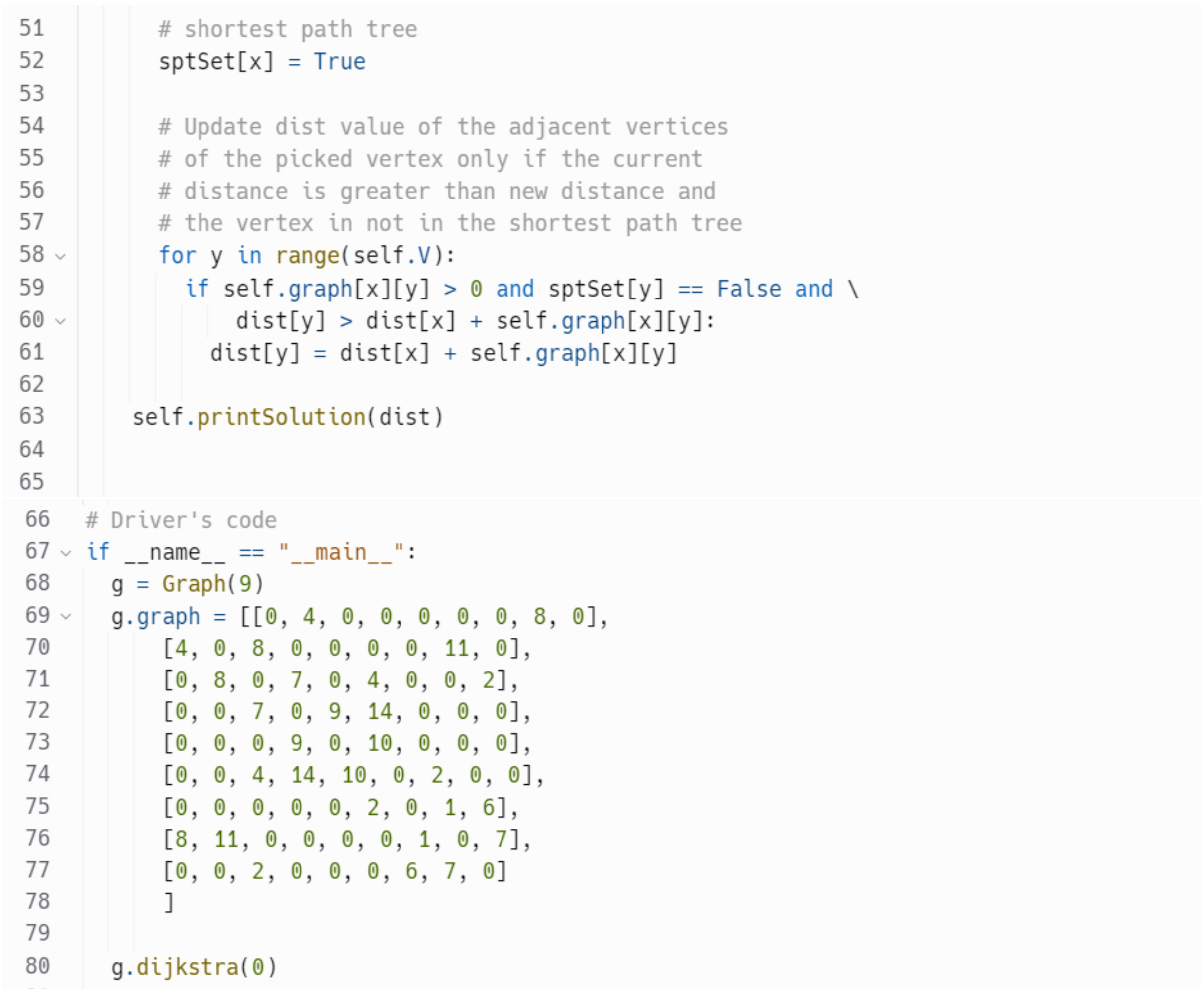


main.py

```python
# Python program for Dijkstra's Algorithm
# Library for INT_MAX
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])

  # A utility function to find the vertex with
  # minimum distance value, from the set of vertices
  # not yet included in shortest path tree
  def minDistance(self, dist, sptSet):

    # Initialize minimum distance for next node
    min = sys.maxsize

    # Search not nearest vertex not in the
    # shortest path tree
```

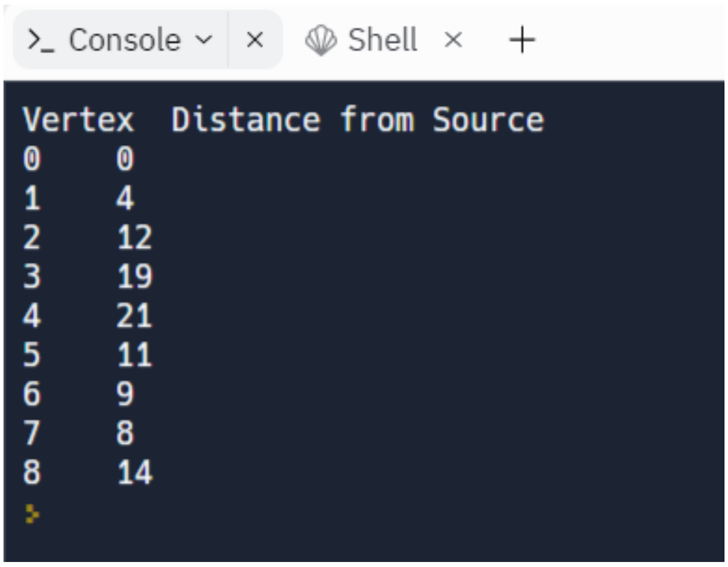


```python
    for u in range(self.V):
      if dist[u] < min and sptSet[u] == False:
        min = dist[u]
        min_index = u

    return min_index

  # Function that implements Dijkstra's single source
  # shortest path algorithm for a graph represented
  # using adjacency matrix representation
  def dijkstra(self, src):

    dist = [sys.maxsize] * self.V
    dist[src] = 0
    sptSet = [False] * self.V

    for cout in range(self.V):

      # Pick the minimum distance vertex from
      # the set of vertices not yet processed.
      # x is always equal to src in first iteration
      x = self.minDistance(dist, sptSet)

      # Put the minimum distance vertex in the
```



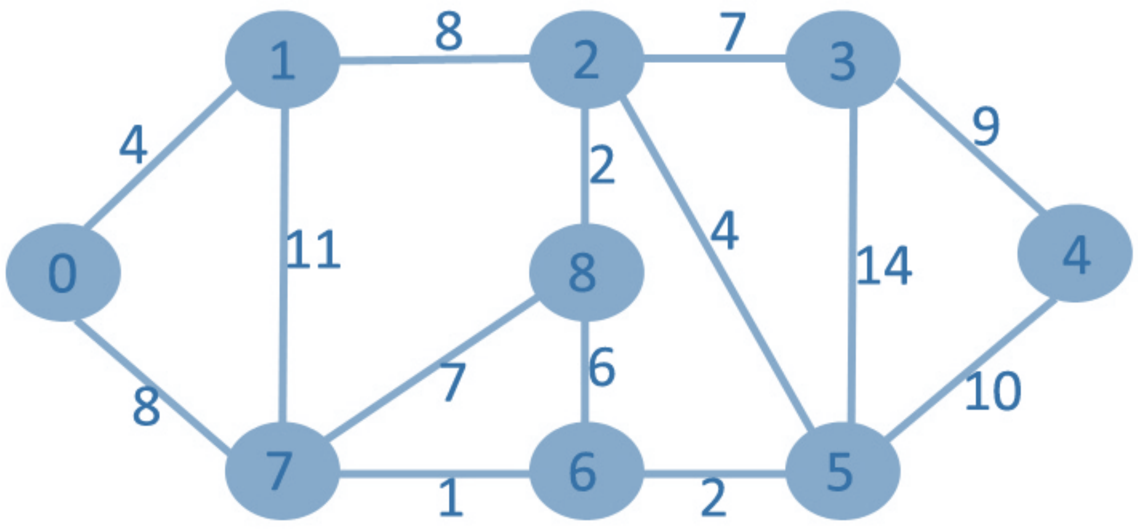
```
      # shortest path tree
      sptSet[x] = True

      # Update dist value of the adjacent vertices
      # of the picked vertex only if the current
      # distance is greater than new distance and
      # the vertex in not in the shortest path tree
      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)


# Driver's code
if __name__ == "__main__":
  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
```

Illustration



Starting from node 0, the initial table would be:

Vertex	Distance from 0	Visited
0	0	Yes
1	4	No
2	∞	No
3	∞	No
4	∞	No
5	∞	No
6	∞	No
7	8	No
8	∞	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	∞	No
4	∞	No
5	∞	No
6	∞	No
7	8	Yes
8	∞	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	∞	No
6	∞	No
7	8	Yes
8	∞	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)logV) 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(VlogV)
- While loop iterating through all vertices - O(V)
- Extracting the vertex with the minimum distance from the priority queue - O(logV)
- For loop iterating through all adjacent vertices of the extracted vertex - O(E)
- Updating the distance of adjacent vertices - O(logV)
- Updating the priority queue with the new distances - O(logV)

- Steps 2, 4, 5, and 6 are repeated V times, and step 1 is done once. Therefore, the overall time complexity is O(VlogV + VElogV), which can be simplified to O((V+E)logV).

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 ALGORTIHM

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.

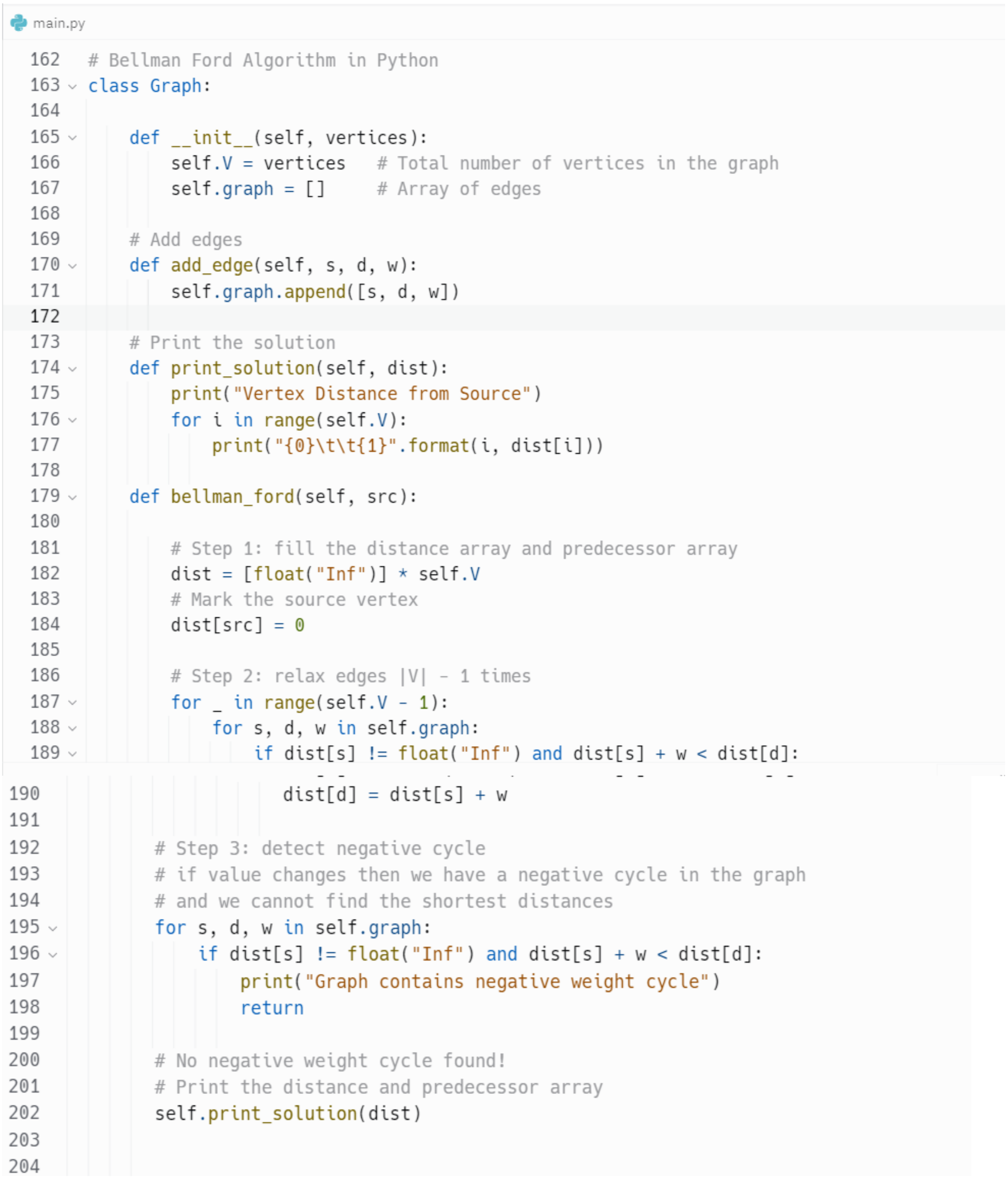
5. If there are no negative cycles, return the distance to each node from the starting node. If there are negative cycles, return "Negative cycle detected”.
6. End

BELLMAN-FORD PSEUDOCODE

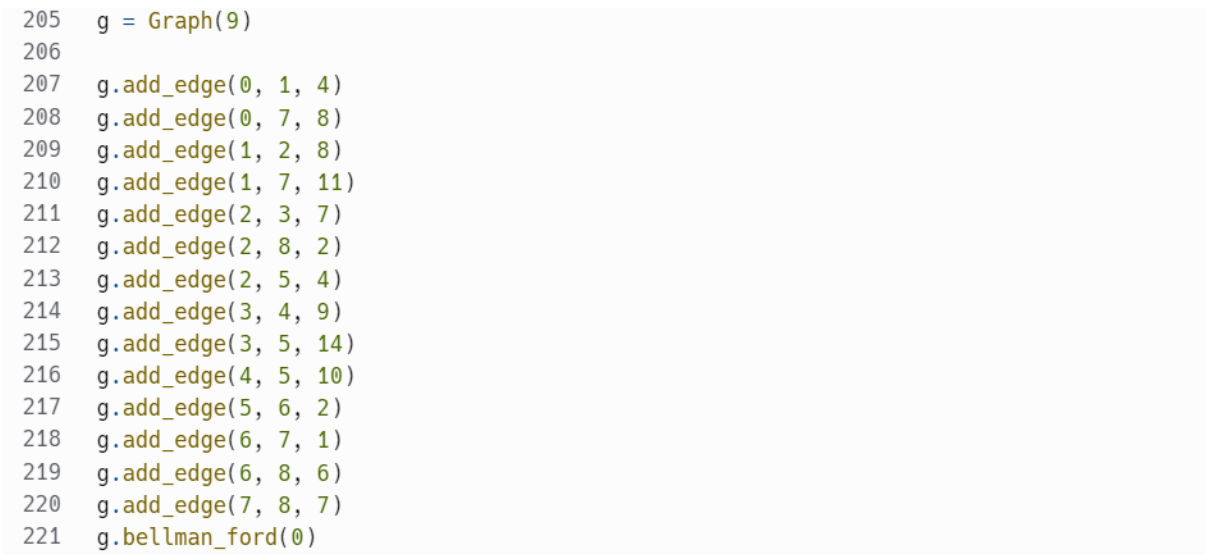
```
function bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite
      previous[V] <- NULL
  distance[S] <- 0

  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

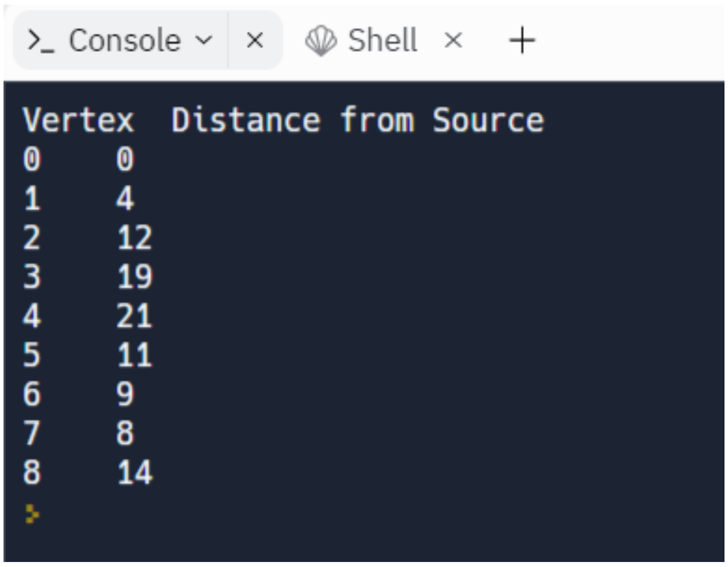

main.py

```python
# Bellman Ford Algorithm in Python
class Graph:

    def __init__(self, vertices):
        self.V = vertices   # Total number of vertices in the graph
        self.graph = []     # Array of edges

    # Add edges
    def add_edge(self, s, d, w):
        self.graph.append([s, d, w])

    # Print the solution
    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):

        # Step 1: fill the distance array and predecessor array
        dist = [float("Inf")] * self.V
        # Mark the source vertex
        dist[src] = 0

        # Step 2: relax edges |V| - 1 times
        for _ in range(self.V - 1):
            for s, d, w in self.graph:
                if dist[s] != float("Inf") and dist[s] + w < dist[d]:
                    dist[d] = dist[s] + w

        # Step 3: detect negative cycle
        # if value changes then we have a negative cycle in the graph
        # and we cannot find the shortest distances
        for s, d, w in self.graph:
            if dist[s] != float("Inf") and dist[s] + w < dist[d]:
                print("Graph contains negative weight cycle")
                return

        # No negative weight cycle found!
        # Print the distance and predecessor array
        self.print_solution(dist)

```



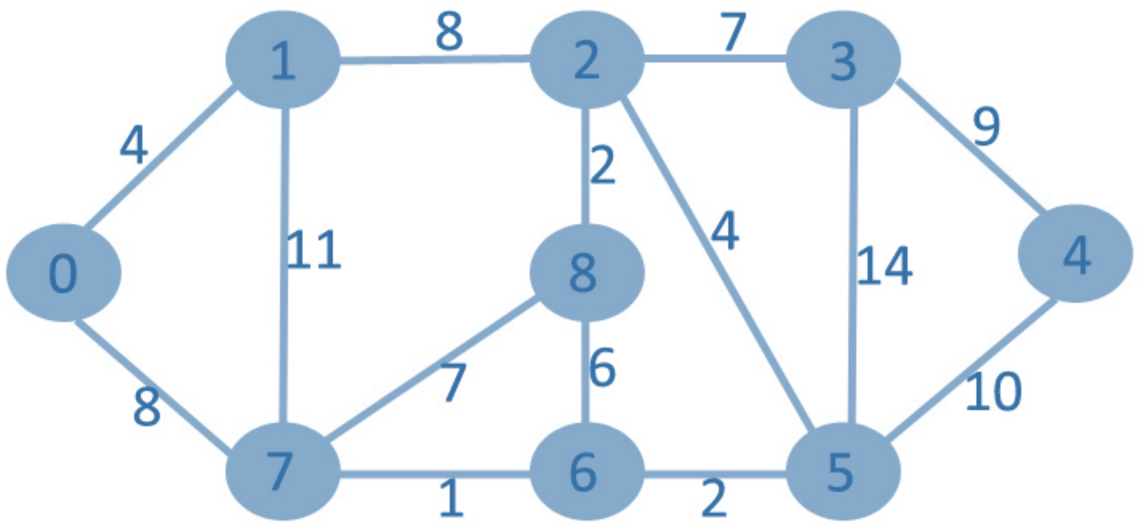
```python
g = Graph(9)

g.add_edge(0, 1, 4)
g.add_edge(0, 7, 8)
g.add_edge(1, 2, 8)
g.add_edge(1, 7, 11)
g.add_edge(2, 3, 7)
g.add_edge(2, 8, 2)
g.add_edge(2, 5, 4)
g.add_edge(3, 4, 9)
g.add_edge(3, 5, 14)
g.add_edge(4, 5, 10)
g.add_edge(5, 6, 2)
g.add_edge(6, 7, 1)
g.add_edge(6, 8, 6)
g.add_edge(7, 8, 7)
g.bellman_ford(0)
```

OUTPUT



```
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	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞

1. **FIRST ITERATION**

Vertex	Distance from 0
0	0
1	4
2	∞
3	∞
4	∞
5	∞
6	∞
7	8
8	∞

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	∞
4	∞
5	∞
6	∞
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	∞
5	∞
6	∞
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*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(ElogV) 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 as 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.

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*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
- Dijkstra's 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