**UIT 2402 – ADVANCED DATA STRUCTURES AND ALGORITHM ANALYSIS**

**EX 7b: Bellman Ford Algorithm**

**Name:** B.VASUNDHARA

**Roll no:** 3122 21 5002 119

**ALGORITHM:**

1. This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2. This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph. Do following for each edge u-v
   * If dist[v] > dist[u] + weight of edge uv, then update dist[v] to
   * dist[v] = dist[u] + weight of edge uv
3. This step reports if there is a negative weight cycle in the graph. Again traverse every edge and do following for each edge u-v   
   ……If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”

**Program Code:**

class Graph:

    def \_\_init\_\_(self, vertices):

        self.V = vertices # No. of vertices

        self.graph = []

    # function to add an edge to graph

    def addEdge(self, u, v, w):

        self.graph.append([u, v, w])

    # utility function used to print the solution

    def printArr(self, dist):

        print("Vertex Distance from Source")

        for i in range(self.V):

            print("{0}\t\t{1}".format(i, dist[i]))

    # The main function that finds shortest distances from src to

    # all other vertices using Bellman-Ford algorithm. The function

    # also detects negative weight cycle

    def BellmanFord(self, src):

        # Step 1: Initialize distances from src to all other vertices

        # as INFINITE

        dist = [float("Inf")] \* self.V

        dist[src] = 0

        # Step 2: Relax all edges |V| - 1 times. A simple shortest

        # path from src to any other vertex can have at-most |V| - 1

        # edges

        for \_ in range(self.V - 1):

            # Update dist value and parent index of the adjacent vertices of

            # the picked vertex. Consider only those vertices which are still in

            # queue

            for u, v, w in self.graph:

                if dist[u] != float("Inf") and dist[u] + w < dist[v]:

                    dist[v] = dist[u] + w

        # Step 3: check for negative-weight cycles. The above step

        # guarantees shortest distances if graph doesn't contain

        # negative weight cycle. If we get a shorter path, then there

        # is a cycle.

        for u, v, w in self.graph:

            if dist[u] != float("Inf") and dist[u] + w < dist[v]:

                print("Graph contains negative weight cycle")

                return

        # print all distance

        self.printArr(dist)

# Driver's code

if \_\_name\_\_ == '\_\_main\_\_':

    g = Graph(5)

    g.addEdge(0, 1, -1)

    g.addEdge(0, 2, 4)

    g.addEdge(1, 2, 3)

    g.addEdge(1, 3, 2)

    g.addEdge(1, 4, 2)

    g.addEdge(3, 2, 5)

    g.addEdge(3, 1, 1)

    g.addEdge(4, 3, -3)

    # function call

    g.BellmanFord(0)

**Input:**

A screenshot of a computer

Description automatically generated

**OUTPUT:**

A screenshot of a computer

Description automatically generated