Name: Vishal Ramane
Roll.No: 2019430010

Experiment No.5

Aim:
Implementation of Bellman ford , Johnson's algorithm for sparse graphs.

Problem Statement:
Analyzing and comparing the Bellman-Ford (single-source shortest path) algorithm and Johnson's (All-pairs shortest path) algorithm and observing their time-complexity behaviour and noting the key differences between the two algorithms over a sample of sparse graphs.

Bellman-Ford Algorithm

Objective:
- Define and Initialize the sparse graph using Python3 programming language.
- Select a vertex as a source vertex from the graph defined and pass the source vertex to Bellman-Ford algorithm as an argument for processing.
- Define and execute relaxation process to find shortest paths, and report if any negative weight cycle is found during this process.

Methodology:
1. Initialize the distances from source to all vertices as infinite and distance to 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. Calculate the shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.

   a) Do following for each edge (u,v):

      If dist[v] > dist[u] + weight of edge (u,v) , then:

         dist[v] = dist[u] + weight of edge (u,v)

3. Report if there is a negative weight cycle in graph.

   a) Do following for each edge (u,v):

      If dist[v] > dist[u] + weight of edge (u,v) , then:

         "Graph contains negative weight cycle"

4. The idea of step 3 is, step 2 guarantees shortest distances, if the graph doesn't contain negative weight cycle (as there can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times) . If we iterate through all edges one more time and still get a shorter path for any vertex, then there must be a negative weight cycle present.
5. TIme Complexity : O(V.E)

Implementation:
1. To instantiate and initialize a Graph we define a *class Graph* with following member functions.
2. *__init__(self, vertices)* function to instantiate the object of class Graph with number of vertices.
3. *addEdge(self, u, v, w)* function is defined to add edges to the Graph Object, where u,v denote - *edge u to v* and *w* denotes weight of the edge.
4. *printArr(self, dist, src)* function is used to result on the terminal where *dist* is an array of integers representing shortest distance of vertices from source vertex *src.*
5. Finally, *BellmanFord(self, src)* function for calling our algorithm.

Name: Vishal Ramane
Roll.No: 2019430010

Experiment No.5

Johnson's Algorithm

Objective:

- Define and Initialize the sparse graph using Python3 programming language.
- Define a subroutine that implements Bellman-Ford Algorithm using Python3 programming language.
- Define another subroutine that implements Dijkstra's Algorithm using Python3 programming language.
- Define a common subroutine for Johnson's Algorithm that uses above two subroutines, and implements re-weighting process to transform the graph into non-negative weight graph.

Methodology:

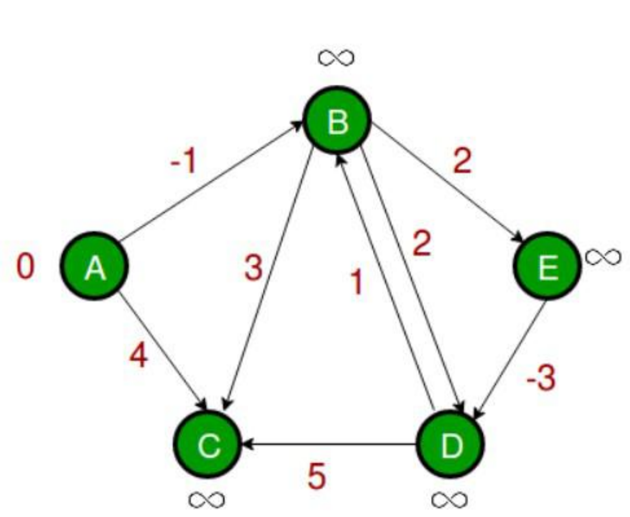
1. Let the given graph be G. Add a new vertex S to the graph, add edges from new vertex to all vertices of G. Let the modified graph be G'.
2. Run Bellman-Ford algorithm on G' with S as source. Let the distances calculated by Bellman-Ford be dist[0], dist[1] , …, dist[V-1] . If we find a negative weight cycle, then return. Note that the negative weight cycle cannot be created by new vertex S as there is no incoming edge to S. All edges are outgoing edges from S.
3. Re-weight the edges of original graph. For each edge (u, v), assign the new weight as :
   " (original weight) + dist[u] – dist[v] ".
   This will convert all negative weights into positive while keeping all shortest paths same.
4. Remove the added vertex S and run Dijkstra's algorithm for every vertex.
5. Time Complexity :
   Bellman Ford is O(V.E) + Dijkstra is O(V.LogV) for every vertex
   so overall, $O(V^2.\log V + V.E)$

Implementation:

1. We define *BellmanFord(edges, graph, num_vertices)* function to execute Bellman-ford algorithm on the given *graph*{*edges,num_vertices*}.
2. Define *Dijkstra(graph, modifiedGraph, src)* and *minDistance(dist, visited)* function to execute Dijkstra's single source shortest path algorithm for every vertex in the *modifiedGraph* for source vertex *src*.
3. Define *JohnsonAlgorithm(graph)* function which will :
   a. call *BellmanFord(edges, graph, num_vertices)* to get distance of all vertices from source *src*.
   b. Define the *modifiedGraph* by using re-weighing process to convert all negative weights into postitive weights.
   c. Call *Dijkstra(graph, modifiedGraph, src)* function on previously defined *modifiedGraph.*

Name: Vishal Ramane
Roll.No: 2019430010

Experiment No.5



Results:

Bellman-Ford Algorithm

```
(base) C:\Users\Vishal Ramane\OneDrive\College\AAC Lab\Code\Expt-5>python -u "c:\Users\Vishal Ramane\OneDrive\College\AAC Lab\Code\Expt-5\BellmanFord.py"

 For source vertex:4

Vertex          Distance from Source
0               inf
1               -2
2               1
3               -3
4               0

(base) C:\Users\Vishal Ramane\OneDrive\College\AAC Lab\Code\Expt-5>
```

Name: Vishal Ramane
Roll.No: 2019430010

Experiment No.5

Johnson's Algorithm

```
(base) C:\Users\Vishal Ramane\OneDrive\College\AAC Lab\Code\Expt-5>python -u "c:\Users\Vishal Ramane\OneDrive\College\AAC Lab\Code\Expt-5\JohnsonAlgo.py"
Modified Graph: [[0, 1, 4, 0, 0], [0, 0, 1, 3, 0], [0, 0, 0, 0, 0], [0, 0, 2, 0, 0], [0, 0, 0, 0, 0]]

Shortest Distance with vertex 0 as the source:

Vertex 0: 0
Vertex 1: 1
Vertex 2: 2
Vertex 3: 1
Vertex 4: 1

Shortest Distance with vertex 1 as the source:

Vertex 0: inf
Vertex 1: 0
Vertex 2: 1
Vertex 3: 0
Vertex 4: 0

Shortest Distance with vertex 2 as the source:

Vertex 0: inf
Vertex 1: inf
Vertex 2: 0
Vertex 3: inf
Vertex 4: inf

Shortest Distance with vertex 3 as the source:

Vertex 0: inf
Vertex 1: 0
Vertex 2: 1
Vertex 3: 0
Vertex 4: 0

Shortest Distance with vertex 4 as the source:

Vertex 0: inf
Vertex 1: 0
Vertex 2: 1
Vertex 3: 0
Vertex 3: 0
Vertex 4: 0
```

n 3.7.3 64-bit ('base': conda) ⊗ 0 ⚠ 1

Name: Vishal Ramane
Roll.No: 2019430010

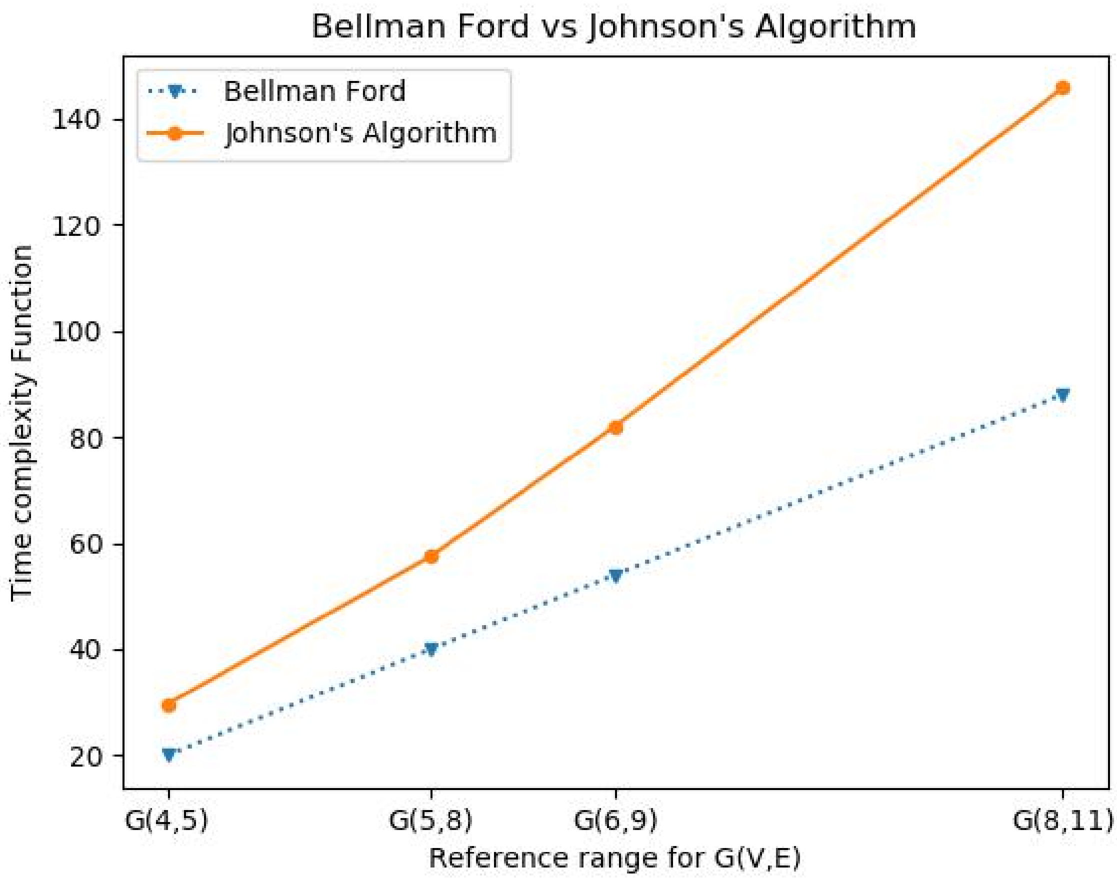
Experiment No.5

Observations:

- Differences:

Bellman-Ford Algorithm	Johnson's Algorithm
Single-source shortest path algorithm.	All -pairs shortest path algorithm.
Based on Dynamic Programming design technique.	Consists of two subroutines Bellman-Ford Algorithm (based on Dy.Programming) and Dijkstra's Algorithm (based on Greedy Algorithm).
Uses Relaxation Process.	Uses Re-weighing process to convert negative weights to positive weights.
Time Complexity : O(V.E)	Time Complexity : $O(V^2.\log V + V.E)$

- Time Complexity :



Conclusion:

Thus Bellman Ford Single source shortest path algorithm and Johnson's All-pairs shortest path algorithm are successfully implemented, studied and analyzed for sparse graphs.