**MODULE 4 (B17)**

**IMPLEMENTATION OF BELLMAN FORD ALGORITHM**

**IN ROUTE MAP NAVIGATION**

**[SNEGA SRI A(CB.EN.U4AIE22163)]**

**Description:**

This module focuses on **“Implementation of Bellman Ford Algorithm in Route Map Navigation”**.It mainly focuses on the second part of the program. The algorithm works by iteratively relaxing the edges of the graph. In each iteration, it checks if there is a shorter path to a node by considering all the edges leading to that node. It updates the distance to each node and keeps track of the previous node on the shortest path.

To retrieve the shortest path between two locations, the algorithm follows the chain of previous nodes from the destination node back to the source node.By implementing the Bellman-Ford algorithm in route map navigation, users can benefit from efficient and reliable solutions to find the shortest path between two locations, enabling them to navigate through complex networks effectively and optimize their travel time.

The functions used in this code:

1. **main()** method: This is the entry point of the program. It executes the steps to find the shortest path between two locations using the Bellman-Ford algorithm.
2. **GetLandmarkName** class: This class provides the functionality to retrieve the name of a landmark based on its corresponding number. It contains a method **getLandmarkName()** that takes a number as input and returns the corresponding landmark name.
3. **InstructionsForUser** class: This class displays instructions for the user and prints a table of landmarks with their corresponding numbers. It has a method **LocationTable()** that generates and prints the table.
4. **BellmanFordAlgorithm** class: This class represents the implementation of the Bellman-Ford algorithm. It contains the following methods and attributes:
   * **numNodes**: An integer variable representing the total number of nodes in the graph.
   * **edges**: A list of **Edge** objects representing the edges in the graph.
   * **addEdge()**: A method to add an edge to the graph. It takes the source node, destination node, and weight as input parameters.
   * **findShortestPath()**: A method that calculates and returns the shortest path between two nodes using the Bellman-Ford algorithm. It takes the start and end nodes as input parameters and returns a list of integers representing the nodes in the shortest path.
   * **Edge** class: A nested class representing an edge in the graph. It contains attributes for the source node, destination node, and weight of the edge.

**Motivation:**

The motivation behind implementing “Route Map Navigation using the Bellman-Ford Algorithm” is to provide an efficient and reliable solution for finding the shortest path between two locations in a given network. While there are other well-known algorithms for finding the shortest path, such as Dijkstra's algorithm, they have limitations when negative-weight edges are present.The key motivation for using the Bellman-Ford algorithm to find the shortest path between two locations is its ability to handle negative-weight edges and detect negative-weight cycles.

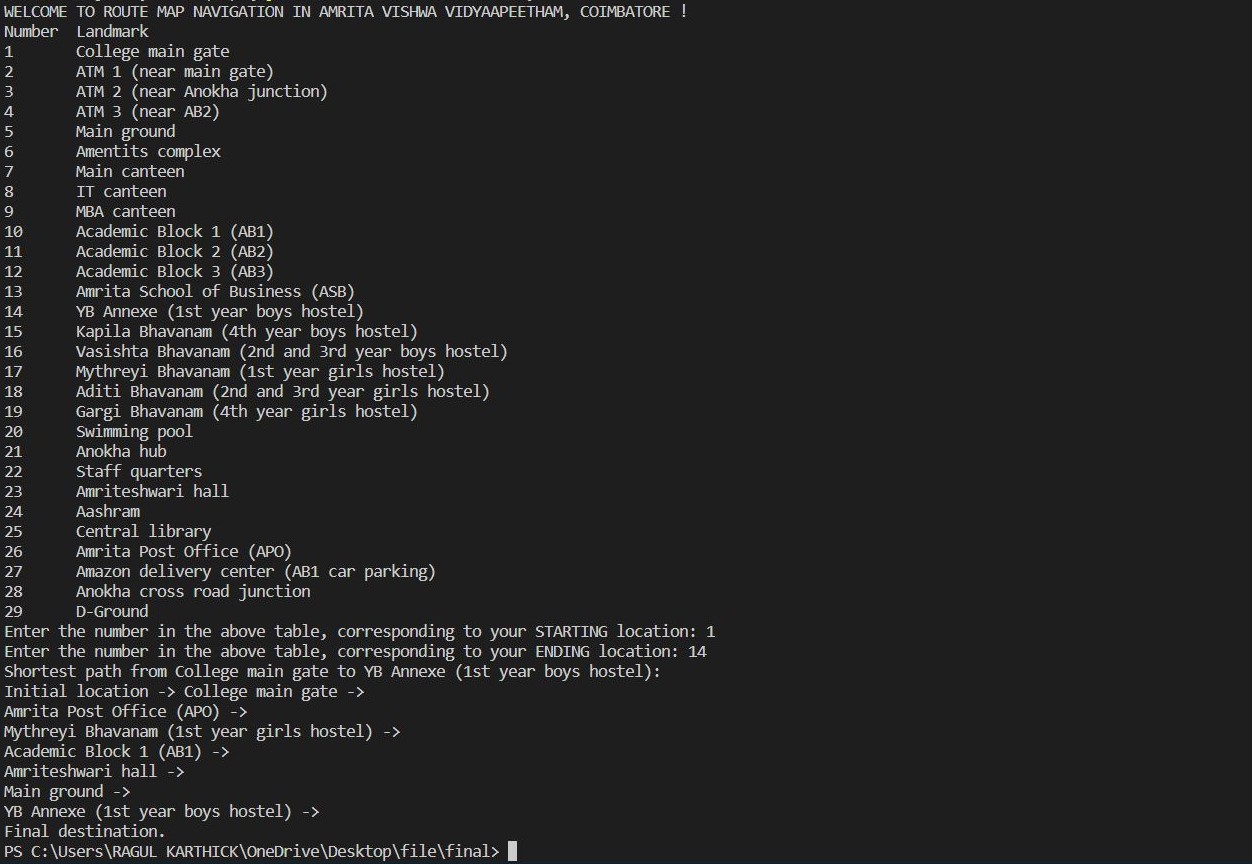
This makes it suitable for a wider range of applications where negative weights may be present, such as in financial modeling, network routing, or transportation systems..By considering the possibility of negative weights, the Bellman-Ford algorithm ensures that it explores all possible paths and iteratively updates the shortest distance estimates until they converge. It also detects negative-weight cycles, which can help identify situations where finding a valid shortest path is not possible.

Overall, the motivation behind using the Bellman-Ford algorithm is to have a reliable and versatile algorithm for finding the shortest path between two locations, even in the presence of negative-weight edges and cycles.

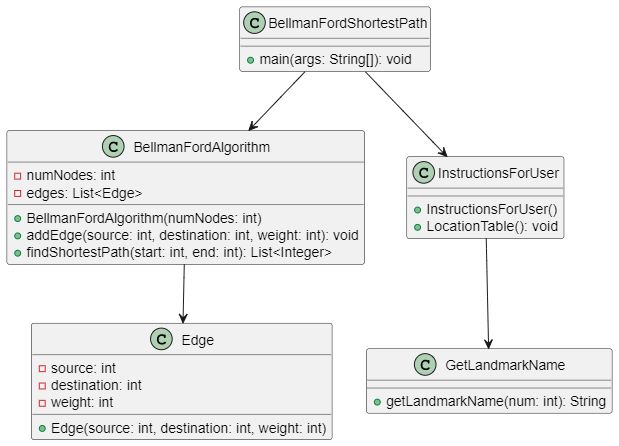
**Relevance of the module in entire system:**

If any person who needs a route map to navigate between any 2 points in the given set of points, This JAVA program would be useful in helping them navigating between their desired locations The importance of finding the shortest path using the Bellman-Ford algorithm lies in its ability to optimize resource allocation, improve efficiency, and enhance decision-making in various domains, including communication networks, transportation systems, logistics, infrastructure design, emergency response, and internet routing. By identifying the most efficient path, it helps save time, reduce costs, and improve overall system performance.

**Input and Output:**



**Appendix:**



1. The **GetLandmarkName** and **InstructionsForUser** classes are instantiated to provide utility functions for getting landmark names and displaying instructions to the user, respectively.
2. An instance of the **BellmanFordAlgorithm** class is created to represent the weighted graph. It is initialized with a total of 36 nodes.
3. Edges are added to the graph using the **addEdge** method of the **BellmanFordAlgorithm** class. Each edge is defined by a source node, destination node, and weight. These edges represent the connections between different locations in the graph, with their corresponding weights representing the distances or costs.
4. The user is prompted to enter the starting and ending locations by providing the corresponding numbers from a table. Input validation is performed to ensure the entered values are within the range of 1 to 29.
5. The **findShortestPath** method of the **BellmanFordAlgorithm** class is invoked to compute the shortest path between the start and end nodes. The method returns a list of nodes representing the path.
6. The numbers of the nodes in the shortest path are stored in an array called **checkPoints** for further processing if needed.
7. The shortest path is printed to the console by iterating over the **shortestPath** list. The landmark names corresponding to each node in the path are retrieved using the **getLandmarkName** method of the **GetLandmarkName** class.
8. The output displays the initial location, followed by the sequence of landmark names representing the shortest path, and finally the final destination.
9. It prints the shortest path from the starting location to the ending location using the landmark names obtained from the **GetLandmarkName** class. The path is printed in the form of "Initial location -> Landmark 1 -> Landmark 2 -> ... -> Final destination."