Programming Assignment 2: Implementing the Bellman-Ford Algorithm in Python

Objective

This assignment is designed to deepen your understanding of the Bellman-Ford algorithm, a fundamental distance-vector routing protocol used to determine the shortest path for routing datagrams in the network layer of computer networks. The Bellman-Ford algorithm stands out for its decentralized nature, where nodes independently calculate the shortest path using an iterative process of computation and information exchange with their immediate neighbors. Upon the completion of this assignment, you should be able to implement and apply the Bellman-Ford algorithm to identify the most efficient paths in a network with dynamic and possibly negative edge weights.

Introduction

The Bellman-Ford algorithm is essential in networks where edge costs can change, such as in fluctuating network traffic conditions, or where there might be negative edge weights. As a distance vector routing protocol, it allows routers to independently calculate the shortest path to all nodes by only knowing the cost to their direct neighbors. This decentralized approach enables each router to construct a distance vector that is iteratively updated and exchanged between neighbors, leading to a consensus of shortest paths throughout the network. Understanding this algorithm is crucial for network engineers and computer scientists who design and manage resilient and efficient communication networks.

Task Overview

In this lab, you will be working with the Bellman-Ford algorithm, a robust method for finding the shortest path in graphs that may contain edges with negative weights. Unlike algorithms such as Dijkstra's, which only work with non-negative edge paths, Bellman-Ford can handle negative edge weights, making it versatile for a wider range of applications. This property is particularly useful in various network scenarios, such as dealing with changing routing costs or potentially malicious routing behaviors.

Network Topology:

Given an undirected (bidirectional) network topology resembling Figure 1.

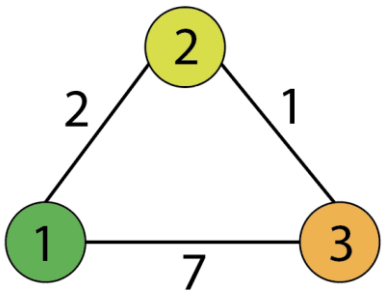

Figure 1: Undirected graph

Nodes: 1, 2, 3
Distance from node 1 to node 2: 2
Distance from node 1 to node 3: 7
Distance from node 2 to node 3: 1

You will undertake the following key tasks:

- **Implement Bellman-Ford Algorithm (50 points):** Write a Python function to implement the Bellman-Ford algorithm. The function should take a graph and a source node as input like a sudo code below:

  ```
  def bellman_ford(graph, source):
  ```

  where the **`graph`** represents the input graph and the **`source`** represents the starting node. The graph can be formatted as a list of edges (source, destination, cost) below:

  ```
  graph = [
      (1, 2, 2),
      (1, 3, 7),
      (2, 3, 1),
  ]
  ```

  The function should return a vector representing the shortest distance from the source to each node in the network. For example, the distance vector obtained from the node 3 should be:
  `[3, 1, 0]`
  Please use the code template provided in Canvas as a starting point. Feel free to modify/change if needed.
- **Test the Algorithm (30 points):** Use the given network description to create a graph. Compute the shortest path from source node 1 to all other nodes. Test your code with different source nodes.

- **Handle Negative Weight Cycles (10 points):** A Negative Weight Cycle in a graph is a cycle (or loop) that has a sum of edge weights that is negative. This means that as you traverse the path of this cycle, the total cost (or distance) decreases with each

complete traversal of the cycle. In a network, negative cycles can cause routing algorithms to continue to decrease the path cost infinitely. This is because, each time the cycle is traversed, the total cost will decrease, and the algorithm may never converge to a stable solution. This can lead to routing tables that constantly change and never settle down, which is known as a "routing loop". The figure below shows an example of a Negative Weight Cycle.

Now, if you start at node 1, go to note 2, then to node 3, and back to node 1, the total cost of the cycle would be:

(1 to 2: -2) + (2 to 3: -2) + (3 to 1: 1) = -3

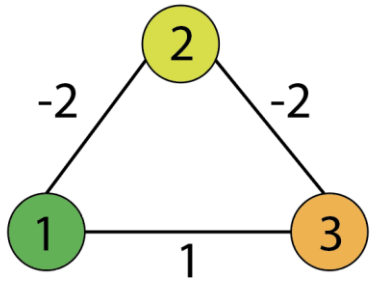

Figure 2: Negative Weight Cycles graph

Using the graph shown in Figure 2, modify your implementation to detect and report any negative weight cycles in the graph. If the graph has a negative weight cycle, the function returns nothing and prints out a short message to the console indicating that a negative weight cycle has been detected.

- **Test your code with a bigger network (10 points):** Test your function using a network provided in Figure 3. Provide the shortest path distance vector for all nodes.

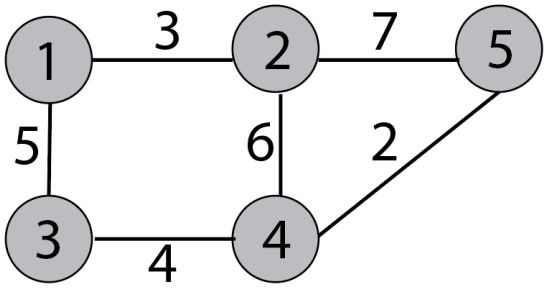

Figure 3: 5 nodes graph

Deliverables:
A Python source code for the Bellman-Ford algorithm implementation.

A report documenting:

- The approach taken to implement the algorithm.
- Observations and analysis of the output on the given network (Figure 1).
- The approach taken to detect a negative weight cycle
- Observations and analysis of the output on the negative cycle network (Figure 2).
- Observations and analysis of the output on the bigger network (Figure 3).

Assessment Criteria:

Correctness and efficiency of the implemented algorithm.
Ability to detect and handle negative weight cycles.
Clarity and thoroughness of the analysis in the report.
This assignment provides a comprehensive exercise on the Bellman-Ford algorithm, combining coding, analysis, and application in network theory. It's suitable for an intermediate-level computer network or algorithms class.