USMAN INSTITUTE OF TECHNOLOGY

Affiliated with NED University of Engineering & Technology, Karachi

Department of Computer Science

B.S. Computer Science

DAA-Design & Analysis of Algorithms

CS-322

Project Title:

Bellman-Ford Algorithm: Single-Source Shortest Path

By

Laiba Khan

21B-077-CS

Batch-2021

Submission Date: 09 June 2025

Instructor

Dr. Lubaid Ahmed

Objective:

The goal of this project is to implement and visualize the Bellman-Ford algorithm in Python to compute the shortest paths from a single source to all other vertices in a weighted directed graph. The project aims to handle both positive and negative edge weights and detect negative weight cycles.

Introduction:

Graph algorithms are fundamental in computer science, particularly in fields like artificial intelligence, networking, and route planning. Among these, shortest path algorithms determine the minimum cost to travel from one vertex to another in a weighted graph. While Dijkstra's algorithm is a well-known solution, it cannot handle negative weights. This limitation makes the Bellman-Ford algorithm more versatile, especially for applications like financial modeling or distributed networks where such weights can occur.

Problem Definition:

In many real-world scenarios, such as network routing and transportation systems, finding the shortest path between locations is crucial. While Dijkstra's algorithm solves this for graphs with non-negative weights, it fails when negative weights are involved. The Bellman-Ford algorithm fills this gap by correctly computing the shortest path even when negative weights are present, and it also checks for negative weight cycles.

Relevant Automata Concepts:

While the Bellman-Ford algorithm is not directly related to formal automata like DFAs or NFAs, it shares foundational principles with computation models:

- **State Transition:** Each vertex can be seen as a state, and edges represent transitions with associated costs.
- **Path Evaluation:** Similar to string processing in automata, Bellman-Ford evaluates multiple paths (sequences of transitions) to find the optimal one.
- **Determinism:** The algorithm deterministically updates distances using relaxation, akin to state transitions in a DFA.
- **Turing Machines** Bellman-Ford solves a well-defined computable problem, representing Turing-decidable logic.

- **Decision Problems** (e.g., "Is there a path of cost ≤ X?")

Methodology:

1. **Input Design:**
   - User specifies the number of vertices, edges, and source vertex.
   - Edge list is provided in the form of (u, v, weight).

2. **Algorithm Design:**
   - Initialize distances: `distance[source] = 0`, all others = ∞.
   - Relax all edges |V|-1 times.
   - Check for negative weight cycles.

3. **Visualization:**
   - A graph is drawn using NetworkX and Matplotlib.
   - Nodes show their labels and distances.
   - Source node is highlighted.

4. **User Interface:**
   - Built using `ipywidgets` for dynamic user input and interaction.

Design & Implementation Approach:

- **Design Phase:**
  - Identified project requirements (negative weights support, user interactivity).
  - Planned use of Python libraries (NetworkX, Matplotlib, ipywidgets).
- **Development Steps:**
  - Developed core Bellman-Ford algorithm function.
  - Built a visualization function to display node distances and graph structure.
  - Created dynamic user input interface using `ipywidgets`.
  - Integrated everything into a single interactive notebook.

- **Testing**:
    - Ran various test cases to validate accuracy.
    - Tested edge cases including invalid inputs and negative weight cycles.
- **Deployment:**
    - Final code is designed for presentation in Jupyter or Google Colab.

Pseudocode:

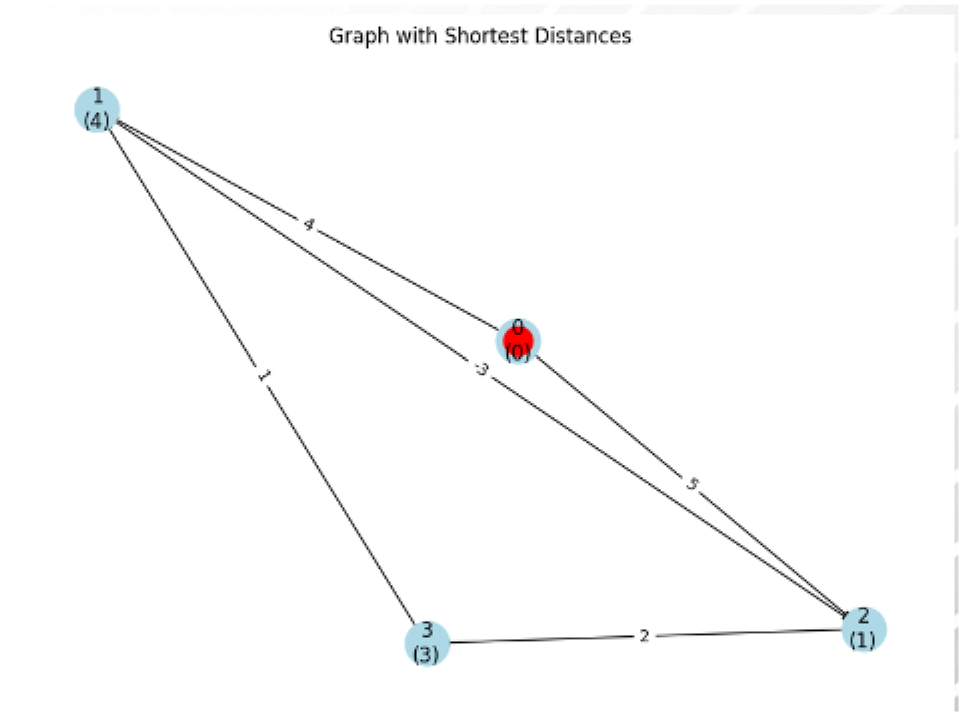
Pseudocode for the algorithm to aid in conceptual understanding:
function BellmanFord(vertices, edges, source):

```
    distance[] ← ∞
    distance[source] ← 0

    for i from 1 to |V|-1:
    for each edge (u, v) with weight w:
    if distance[u] + w < distance[v]:
        distance[v] ← distance[u] + w

    for each edge (u, v) with weight w:
    if distance[u] + w < distance[v]:
    report "Negative cycle detected"
    return None

    return distance
```

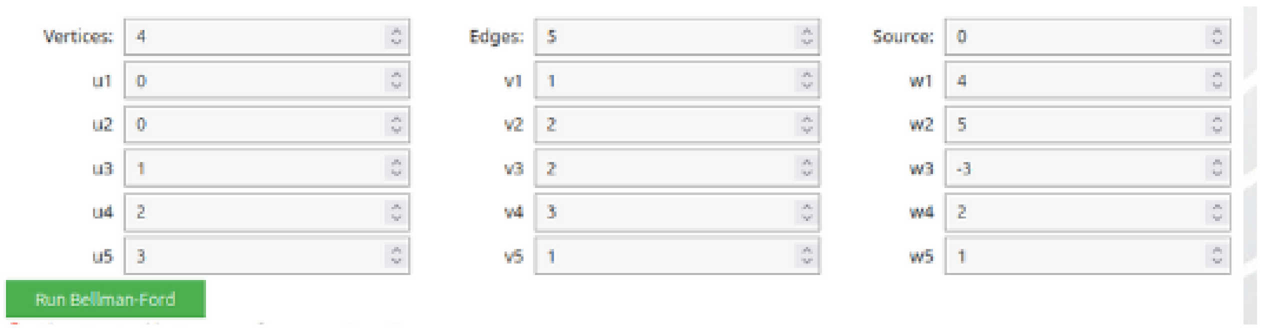
Code Summary:

1. **Main Function:**
   - `bellman_ford(vertices, edges, source)`
   - Calculates shortest distances and checks for negative cycles.

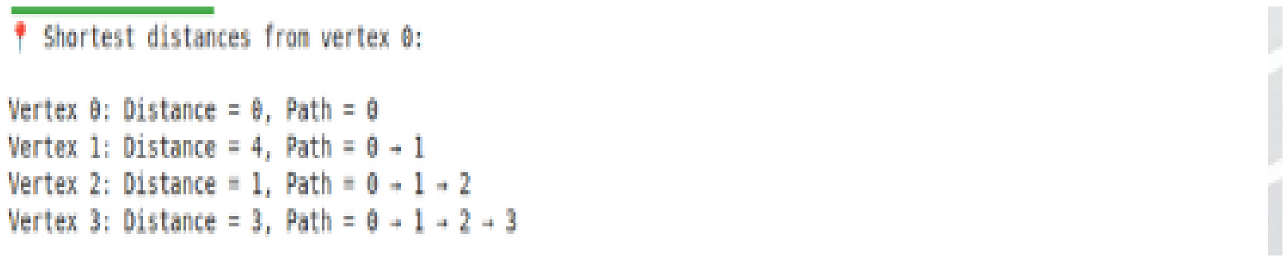
2. **Graph Visualization:**
   - `visualize_graph(...)` draws nodes and edges with weights and distances.



3. **Interactive UI:**
   - Users input graph details using widgets and get instant output.



Vertices:	4	Edges:	5	Source:	0
u1	0	v1	1	w1	4
u2	0	v2	2	w2	5
u3	1	v3	2	w3	-3
u4	2	v4	3	w4	2
u5	3	v5	1	w5	1



```
📍 Shortest distances from vertex 0:

Vertex 0: Distance = 0, Path = 0
Vertex 1: Distance = 4, Path = 0 → 1
Vertex 2: Distance = 1, Path = 0 → 1 → 2
Vertex 3: Distance = 3, Path = 0 → 1 → 2 → 3
```

Important Functions:

- `bellman_ford()` - Implements the algorithm logic.
- `visualize_graph()` - Shows graph with current distances.
- `on_button_click()` - Processes input and runs the algorithm.

Complexity:

- **Time Complexity:** O(V × E), where V is the number of vertices and E is the number of edges.
- **Space Complexity:** O(V), for storing distance array.

Algorithm Complexity & Comparison:

Bellman-Ford Algorithm:

- **Time Complexity:** O(V × E)
- **Space Complexity:** O(V)
- Suitable for graphs with **negative weights**.
- Detects **negative weight cycles**.

Comparison with Other Algorithms:

- **Dijkstra's Algorithm:**
  - Time Complexity: O((V + E) log V) using priority queue
  - Space Complexity: O(V)
  - **Does not support negative weights**
  - Generally faster on non-negative graphs

- **Floyd-Warshall Algorithm:**
  - Time Complexity: $O(V^3)$
  - Space Complexity: $O(V^2)$
  - Computes **all-pairs shortest paths**
  - Less efficient for single-source problems

Comparison Table:

Algorithm	Time Complexity	Space Complexity	Handles Negative Weights	Best Use Case
Bellman-Ford	O(V × E)	O(V)	Yes	Single-source with negative weights
Dijkstra	O((V + E) log V)	O(V)	No	Single-source with positive weights
Floyd-Warshall	$O(V^3)$	$O(V^2)$	Yes	All-pairs shortest paths

Conclusion:

Bellman-Ford is the best choice when graphs may contain negative weights and only single-source shortest paths are required. For non-negative weights, Dijkstra is more efficient. Floyd-Warshall is preferable for dense graphs needing all-pairs shortest paths.

Challenges & Solutions:

1. **Dynamic Input Handling:**
    - Challenge: Creating dynamic fields for edge input.
    - Solution: Used `ipywidgets` to generate inputs based on user-defined edge count.

2. **Error Checking:**
    - Challenge: User might enter invalid vertices.
    - Solution: Input validation logic was added before processing edges.

3. **Negative Weight Detection:**
    - Challenge: Ensure correctness in presence of negative weights.
    - Solution: Proper implementation of cycle detection post relaxation.

4. **Visualization:**
    - Challenge: Displaying distances with node labels.
    - Solution: Used formatted strings to show distance on each node.

5 . Generalization:

- Challenge: Making code reusable for any graph input.
- Solution: Designed dynamic UI and decoupled input/output.

Conclusion:

The Bellman-Ford algorithm is essential for computing shortest paths in graphs with negative weights. This project demonstrates a complete implementation with user interaction and visualization, suitable for academic evaluation and practical understanding. Through this work, we explored graph theory, algorithm optimization, and user-friendly design. The project also reinforced the significance of computational models and problem-solving in computer science.

Learning Outcomes:

- Gained a deeper understanding of dynamic programming and graph algorithms.
- Developed skills in user interface creation and algorithm visualization.
- Understood the practical application of shortest path algorithms in real-life scenarios.