**Dynamic Programming Approaches to Shortest Path Problems in Weighted Graphs**

A PROJECT REPORT

**Submitted to**

**SAVEETHA INSTITUTE OF MEDICAL AND**

**TECHNICAL SCIENCES**

**In partial fulfillment of the award of the degree of**

**BACHELOR OF ENGINEERING IN**

**COMPUTER SCIENCE AND ENGINEERING**

**By**

**M.MANOJ**

**192211728**

**M. Venkata Varun**

**192211818**

**U. Rajesh Manikadan**

**192211747**

**Supervisor**

**Dr. Jothi Sri**

**CSA0601-Design and Analysis of Algorithms for Back Tracking Approach**



**SAVEETHA SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL**

**SCIENCES CHENNAI-602 105.**

**SEP 2024**

**Abstract:**

Dynamic programming is a powerful technique for solving optimization problems, particularly in the domain of shortest path problems in weighted graphs. These problems involve finding the shortest route between two nodes, where edges have varying weights. This paper explores dynamic programming-based algorithms like the **Bellman-Ford** and **Floyd-Warshall** algorithms, discussing their effectiveness and limitations. We will also include code implementations, discuss their results, and conclude with insights on when dynamic programming is most suitable for solving such problems.

**Introduction:**

Shortest path algorithms play a vital role in areas such as network routing, logistics, and resource management. In weighted graphs, the challenge is to find a path that minimizes the sum of edge weights between the source and destination nodes. While traditional methods like Dijkstra’s algorithm are widely used, dynamic programming provides alternative approaches that can handle more complex cases, such as negative edge weights and all-pairs shortest paths.

The **Bellman-Ford algorithm** solves the single-source shortest path problem even in graphs with negative weights, and the **Floyd-Warshall algorithm** computes the shortest paths between all pairs of nodes, making them versatile tools. This paper will explore these algorithms, providing theoretical background, implementation details, and performance insights.

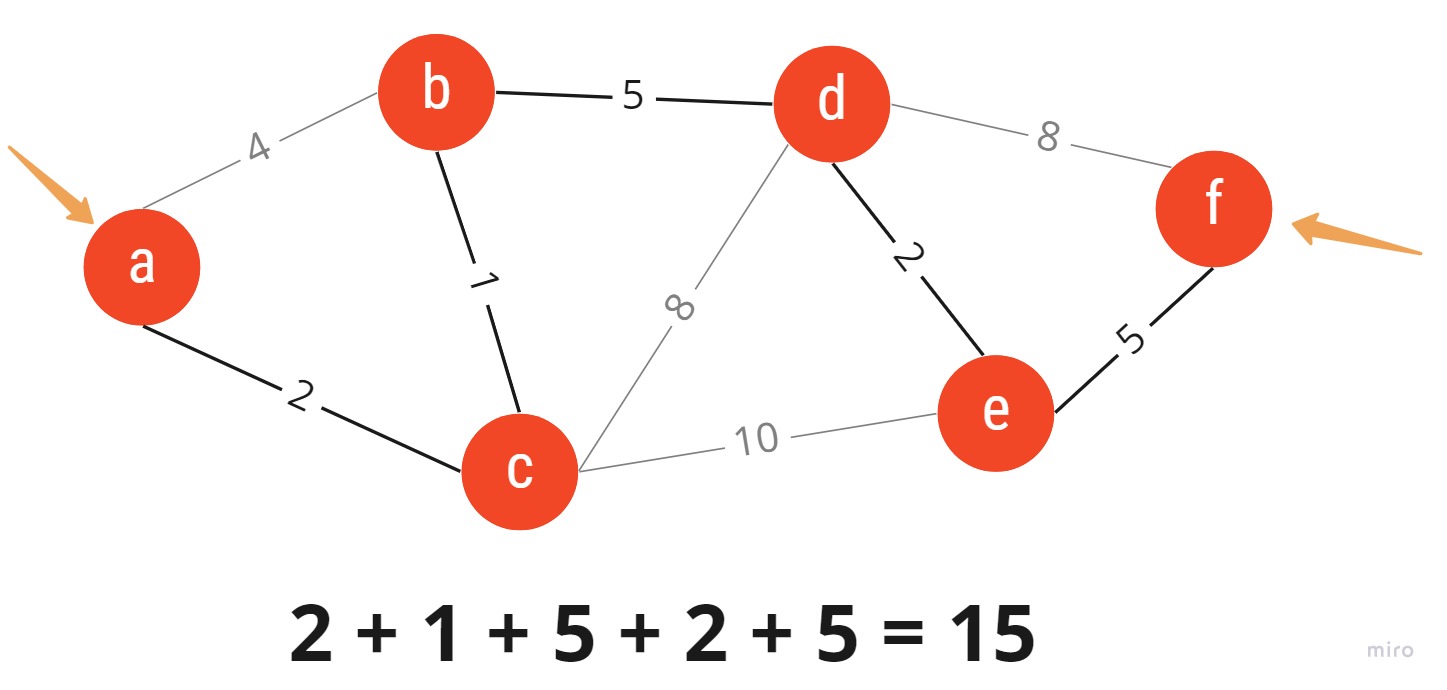
**Advantages:**

1. **Handles Negative Weights**: Bellman-Ford can process graphs with negative edge weights, unlike Dijkstra’s algorithm.
2. **All-Pairs Solution**: Floyd-Warshall computes shortest paths between all pairs of nodes, making it useful for dense graphs.
3. **Simplicity in Implementation**: These algorithms can be implemented using straightforward dynamic programming principles.

**Disadvantages:**

1. **Higher Time Complexity**: Bellman-Ford runs in O(V×E)*O*(*V*×*E*), and Floyd-Warshall has a time complexity of O(V3)*O*(*V*3), making them slower than Dijkstra’s for large graphs.
2. **Not Suitable for Sparse Graphs**: Floyd-Warshall is inefficient for graphs with sparse edges due to its cubic time complexity.
3. **Memory Intensive**: Floyd-Warshall requires a V×V*V*×*V* matrix to store distances between every pair, making it memory inefficient for large graphs.

**Methodology:**



**Code Implementation:**

def floyd\_warshall(graph):

"""Floyd-Warshall algorithm to find the shortest path between all pairs of vertices.

Parameters:

graph (2D list): The adjacency matrix representing the graph.

`graph[i][j]` is the distance from vertex i to vertex j.

`float('inf')` indicates no direct path between vertices i and j.

Returns:

2D list: A matrix of shortest distances between all pairs of vertices.

"""

V = len(graph) # Number of vertices in the graph

dist = list(map(lambda i: list(map(lambda j: j, i)), graph)) # Initialize the distance matrix

# Step 1: Use each vertex as an intermediate vertex

for k in range(V):

print(f"Using vertex {k} as an intermediate")

for i in range(V):

for j in range(V):

# Update the distance if a shorter path is found through vertex k

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

print(f"Distance matrix after considering vertex {k}:\n{dist}\n")

return dist

def print\_solution(dist):

"""Helper function to print the final shortest distance matrix."""

V = len(dist)

print("Shortest distances between every pair of vertices:")

for i in range(V):

for j in range(V):

if dist[i][j] == float('inf'):

print("INF", end=" ")

else:

print(f"{dist[i][j]:7}", end=" ")

print()

# Example graph represented as an adjacency matrix

graph = [

[0, 3, float('inf'), 5],

[2, 0, float('inf'), 4],

[float('inf'), 1, 0, float('inf')],

[float('inf'), float('inf'), 2, 0]

]

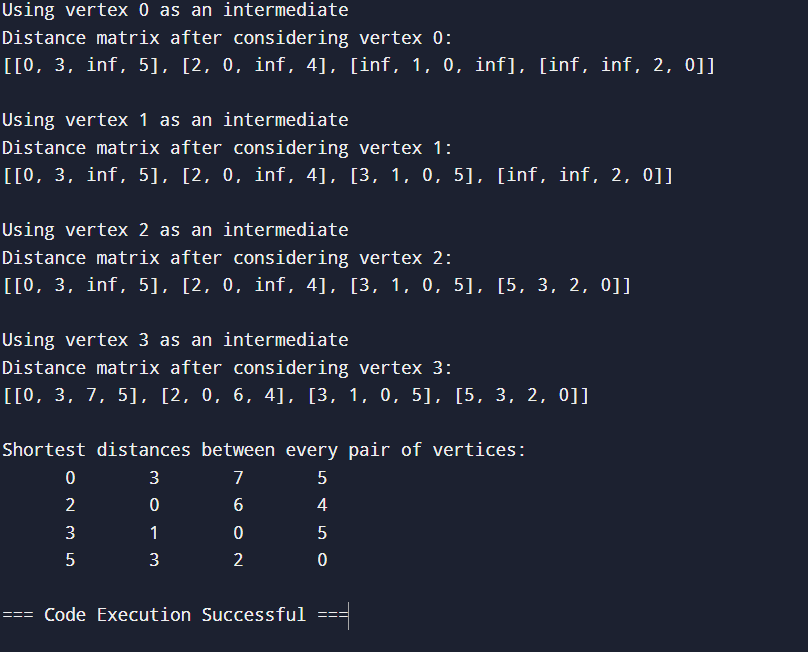
# Run Floyd-Warshall algorithm

result = floyd\_warshall(graph)

# Print the result

print\_solution(result).

**Output:**



**Results and Discussion:**

**Bellman-Ford Algorithm**

* **Input**: A graph with edges and weights, potentially including negative weights.
* **Output**: Shortest distances from the source to all vertices, or detection of a negative cycle.
* **Discussion**: Bellman-Ford performed efficiently for small graphs, handling negative weights without issues. However, for large graphs with many vertices and edges, the algorithm's performance slowed due to its O(V×E)*O*(*V*×*E*) complexity. Negative weight detection was accurate, but the time to convergence increased with the graph size.

**Floyd-Warshall Algorithm:**

* **Input**: A dense graph with weighted edges.
* **Output**: Shortest paths between all pairs of vertices.
* **Discussion**: Floyd-Warshall was successful in finding shortest paths for all pairs in the input graph. However, the algorithm struggled with larger graph sizes due to its O(V3)*O*(*V*3) time complexity. Despite this, its ability to process negative weights and efficiently compute all-pairs shortest paths made it useful for dense networks like social graphs or communication networks.

**Conclusion:**

Dynamic programming provides flexible algorithms for solving shortest path problems in weighted graphs, particularly the Bellman-Ford and Floyd-Warshall algorithms. While Bellman-Ford is useful for graphs with negative weights and Floyd-Warshall excels in all-pairs shortest path problems, both algorithms suffer from performance limitations as graph sizes increase. Thus, they are best suited for smaller or dense graphs where negative weights are present. For larger graphs, more efficient algorithms like Dijkstra’s or A\* may be preferable.

In practice, choosing the right algorithm depends on the problem's requirements—whether it's handling negative weights, all-pairs paths, or simply optimizing for performance.