Test Case Results for all the four algorithms: Verified

Input:
There are three lines each of which have 2 space separated integers denoting the source and destination from the **Project2_Input-File4.csv**

```
192 163
138 66
465 22
```

Output:

```
Distance from 192 to 163: 819 feet
Path: [192, 157, 194, 158, 161, 163]
----------------------------------------------------------------------------------------------------------------
Distance from 138 to 66: 2728 feet
Path: [138, 162, 136, 159, 119, 116, 114, 112, 70, 110, 79, 108, 107, 103, 105, 85, 67, 66]
----------------------------------------------------------------------------------------------------------------
Distance from 465 to 22: 6738 feet
Path: [465, 377, 380, 372, 363, 364, 305, 247, 210, 233, 201, 170, 128, 98, 78, 41, 23, 22]
----------------------------------------------------------------------------------------------------------------
```

Time performance:

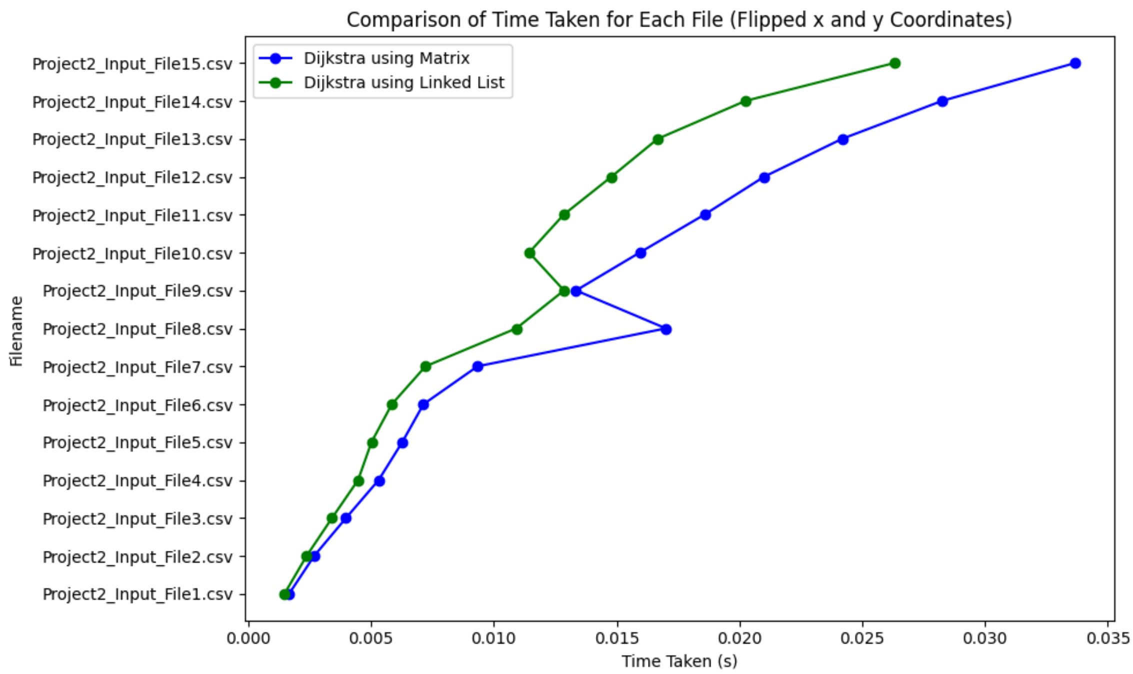
Following are the time taken for Dijkstra algorithm using two-dimensional array:

```
Filename,TimeTaken(s)
Project2_Input_File1.csv,0.001668
Project2_Input_File2.csv,0.00269
Project2_Input_File3.csv,0.003986
Project2_Input_File4.csv,0.005313
Project2_Input_File5.csv,0.006275
Project2_Input_File6.csv,0.007139
Project2_Input_File7.csv,0.009342
Project2_Input_File8.csv,0.017007
Project2_Input_File9.csv,0.013348
Project2_Input_File10.csv,0.015942
Project2_Input_File11.csv,0.018576
Project2_Input_File12.csv,0.021007
Project2_Input_File13.csv,0.024184
Project2_Input_File14.csv,0.028252
Project2_Input_File15.csv,0.033669
```

Following are the time taken for Dijkstra algorithm using linked list:

```
Filename,TimeTaken(s)
Project2_Input_File1.csv,0.001479
Project2_Input_File2.csv,0.002381
Project2_Input_File3.csv,0.003417
Project2_Input_File4.csv,0.004476
Project2_Input_File5.csv,0.005022
Project2_Input_File6.csv,0.005851
Project2_Input_File7.csv,0.007222
Project2_Input_File8.csv,0.010918
Project2_Input_File9.csv,0.012859
Project2_Input_File10.csv,0.011439
Project2_Input_File11.csv,0.012848
Project2_Input_File12.csv,0.01479
Project2_Input_File13.csv,0.016674
Project2_Input_File14.csv,0.020248
Project2_Input_File15.csv,0.026313
```

Following is the graph for the above data for both approaches of dijkstra's algorithm:



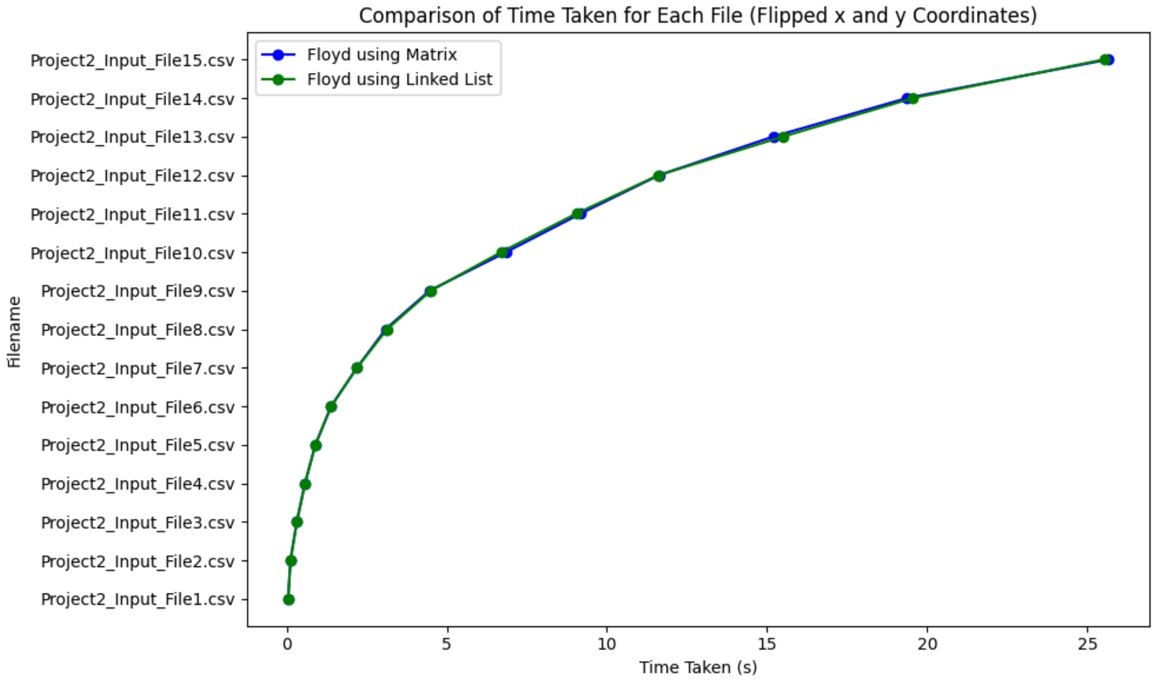
Following are the time taken for Floyd algorithm using two-dimensional array:

```
Filename,TimeTaken(s)
Project2_Input_File1.csv,0.044162
Project2_Input_File2.csv,0.118364
Project2_Input_File3.csv,0.316299
Project2_Input_File4.csv,0.570785
Project2_Input_File5.csv,0.895883
Project2_Input_File6.csv,1.38994
Project2_Input_File7.csv,2.19505
Project2_Input_File8.csv,3.08808
Project2_Input_File9.csv,4.46069
Project2_Input_File10.csv,6.85061
Project2_Input_File11.csv,9.1815
Project2_Input_File12.csv,11.6325
Project2_Input_File13.csv,15.1967
Project2_Input_File14.csv,19.3726
Project2_Input_File15.csv,25.6591
```

Following are the time taken for Floyd algorithm using linked list:

```
Filename,TimeTaken(s)
Project2_Input_File1.csv,0.044042
Project2_Input_File2.csv,0.118226
Project2_Input_File3.csv,0.314827
Project2_Input_File4.csv,0.571439
Project2_Input_File5.csv,0.89544
Project2_Input_File6.csv,1.38312
Project2_Input_File7.csv,2.1957
Project2_Input_File8.csv,3.13641
Project2_Input_File9.csv,4.50226
Project2_Input_File10.csv,6.69718
Project2_Input_File11.csv,9.04437
Project2_Input_File12.csv,11.616
Project2_Input_File13.csv,15.5061
Project2_Input_File14.csv,19.5627
Project2_Input_File15.csv,25.5237
```

Following is the graph for the above data for both approaches of floyd's algorithm:



Memory Performance:

1. Two-Dimensional Array Memory Usage in Floyd's Algorithm:

Memory Components:

- **Graph**: A two-dimensional array (graph) of size MAX_SIZE x MAX_SIZE to store the distance between each pair of nodes.
- **Shortest Path Matrix**: Another two-dimensional array (shortestPath) of the same size to store the shortest distances between nodes.
- **Paths Matrix**: A two-dimensional array (paths) to store the next node in the shortest path between two nodes.
- **Edges List**: A list of Edge structs that stores the edges of the graph.

Key Variables:

- Let n be the number of nodes (i.e., MAX_SIZE).
- Let m be the number of edges.

Memory Usage Breakdown:

1. **Graph**:
   - Each entry in the graph is a double, which takes 8 bytes.
   - The size of the graph array is n x n.
   - Memory for graph = 8 * n * n bytes.
2. **Shortest Path Matrix**:
   - Similar to the graph, each entry is a double.
   - Memory for shortestPath = 8 * n * n bytes.
3. **Paths Matrix**:
   - Each entry in paths is an int, which takes 4 bytes.
   - Memory for paths = 4 * n * n bytes.
4. **Edges List**:
   - Each Edge struct consists of two int variables (8 bytes total) and one double variable (8 bytes), so each Edge takes 16 bytes.
   - Memory for edges = 16 * m bytes.

The total memory usage is the sum of all the above components:

- Total memory usage ≈ 8 * n * n (graph) + 8 * n * n (shortestPath) + 4 * n * n (paths) + 16 * m (edges) bytes.

Above calculation results in the following formula for memory usage:

Total Memory ≈ 20n^2+16m bytes
Big-O ≈ O(n^2)

2. Linked List Memory Usage in Floyd's Algorithm:

Memory Components:

- **Graph:** Represented as an adjacency list where each node points to a linked list of adjacent nodes and distances.
- **Shortest Path Matrix:** A 2D array to store the shortest path distances between each pair of nodes.
- **Paths Matrix:** A 2D array to store the next node in the path between source and destination nodes.
- **Edges List:** A list of Edge structs to store the graph's edges.

Key Variables:

- Let n be the number of nodes (i.e., MAX_SIZE).
- Let m be the number of edges.

Memory Usage Breakdown:

1. **Graph (Adjacency List):**
   - Each node in the adjacency list points to a LinkedListNode, which contains an int (4 bytes for data), a double (8 bytes for distance), and a pointer (8 bytes on 64-bit systems - Machine dependent) to the next node.
   - For each edge, there is a LinkedListNode storing this information, so each edge uses 4 (data) + 8 (distance) + 8 (pointer) = 20 bytes.
   - Memory for graph = 20 * m bytes (since each edge will have a corresponding LinkedListNode).
2. **Shortest Path Matrix:**
   - Similar to the matrix-based version, this is a 2D array where each entry is a double (8 bytes).
   - Memory for shortestPath = 8 * n * n bytes.
3. **Paths Matrix:**
   - Each entry in the paths matrix is an int, taking 4 bytes.
   - Memory for paths = 4 * n * n bytes.
4. **Edges List:**
   - Each Edge struct contains two int variables (4 bytes each) and one double variable (8 bytes), for a total of 16 bytes per edge.
   - Memory for edges = 16 * m bytes.

The total memory usage can be calculated as:

- Memory ≈ 20 * m (graph) + 8 * n * n (shortestPath) + 4 * n * n (paths) + 16 * m (edges) bytes.

Total Memory ≈ 12n^2+36m bytes
Big-O ≈ O(n^2)

3. Two-Dimensional Array Memory Usage in Dijkstra's Algorithm:

Memory Components:

- **Graph (Adjacency Matrix):** A two-dimensional vector graph of size MAX_SIZE x MAX_SIZE is used to store the distances between nodes in the graph.
- **Shortest Path Array:** A one-dimensional vector shortestPath stores the shortest distance from the source node to every other node.
- **Paths Array:** A one-dimensional vector paths is used to track the next node in the shortest path between the source and destination.
- **Visited Array:** A one-dimensional vector vis is used to track whether a node has been processed in Dijkstra's algorithm.
- **Edges List:** A list of Edge structs to store the edges of the graph. Each edge contains two nodes and the distance between them.

Key Variables:

- Let n be the number of nodes (i.e., MAX_SIZE).
- Let m be the number of edges.

Memory Usage Breakdown:

- **Graph (Adjacency Matrix):**
  - Each entry in the adjacency matrix is a double, which takes 8 bytes.
  - The size of the graph array is n x n.
  - Memory for the graph = 8 * n * n bytes.
- **Shortest Path Array:**
  - Each entry in the shortestPath array is a double, which takes 8 bytes.
  - Memory for the shortestPath array = 8 * n bytes.
- **Paths Array:**
  - Each entry in the paths array is an int, which takes 4 bytes.
  - Memory for the paths array = 4 * n bytes.
- **Visited Array:**
  - Each entry in the vis array is an int, which takes 4 bytes.
  - Memory for the visited array = 4 * n bytes.
- **Edges List:**
  - Each Edge struct consists of two int variables (8 bytes total) and one double variable (8 bytes), so each edge takes 16 bytes.
  - Memory for the edges = 16 * m bytes.

The total memory usage is the sum of all the above components:

- Total memory usage ≈ 8 * n * n (graph) + 8 * n (shortestPath) + 4 * n (paths) + 4 * n (visited) + 16 * m (edges) bytes.

Total Memory ≈ 8n^2+16n+16m bytes
Big-O ≈ O(n^2)

4. Linked List Memory Usage in Dijkstra's Algorithm:

Memory Components:

- **Graph (Adjacency List):** A vector of linked lists (graph) is used to store the adjacency list of the graph. Each node contains its distance and a pointer to the next node.
- **Shortest Path Array:** A one-dimensional vector shortestPath stores the shortest distance from the source node to every other node.
- **Paths Array:** A one-dimensional vector paths is used to track the next node in the shortest path between the source and destination.
- **Visited Array:** A one-dimensional vector vis is used to track whether a node has been processed in Dijkstra's algorithm.

- **Edges List:** A list of Edge structs to store the edges of the graph. Each edge contains two nodes and the distance between them.
- **Linked List Node Structure:** Each LinkedListNode stores an integer data (representing the destination node), a double distance, and a pointer next.

Key Variables:

- Let n be the number of nodes (i.e., MAX_SIZE).
- Let m be the number of edges.

Memory Usage Breakdown:

- **Graph (Adjacency List):**
  - Each linked list node stores an integer (data), a double (distance), and a pointer (next).
  - Each node in the adjacency list takes 4 bytes (int data) + 8 bytes (double distance) + 8 bytes (next pointer) = 20 bytes per node.
  - Memory for all the linked list nodes = 20 * m bytes (since there are m edges in the graph).
- **Shortest Path Array:**
  - Each entry in the shortestPath array is a double, which takes 8 bytes.
  - Memory for the shortestPath array = 8 * n bytes.
- **Paths Array:**
  - Each entry in the paths array is an int, which takes 4 bytes.
  - Memory for the paths array = 4 * n bytes.
- **Visited Array:**
  - Each entry in the vis array is an int, which takes 4 bytes.
  - Memory for the visited array = 4 * n bytes.
- **Edges List:**
  - Each Edge struct consists of two int variables (8 bytes total) and one double variable (8 bytes), so each edge takes 16 bytes.
  - Memory for the edges = 16 * m bytes.

The total memory usage can be calculated as:

- Memory ≈ 20 * m (graph) + 8 * n (shortestPath) + 4 * n (paths) + 4 * n (visited) + 16 * m (edges) bytes.

Total Memory = 20m+8n+4n+4n+16m
Total Memory ≈ 36m+16n bytes
Big-O ≈ O(m + n)