Shortest Path Problem

Team Members:

Name:	ID:
Mohamed Bekheet	202201714
Omar Toulba	202202155
Abdel Rahman Omer	202202254

Introduction:

The main concept in this project is the shortest path problem. It finds the shortest path or route from point x to point y where point x is the starting point and y is the destiny or the end point. The graph is the best way to represent this problem and how it works. The graph is a type of non-linear non-primitive data structure that contains a set of vertices and edges. The idea of the shortest path is calculating the minimum weight of the edges from point x to point y in the graph to reach destiny with the minimum amount of cost. The shortest path has a lot of applications in the real world.

For instance, representing a map we need to use a graph to find the shortest path, transportation, network design, telecommunication, and a lot more. There are many ways to implement the shortest path. For example, Dijkstra's Algorithm, Floyd-Warshall Algorithm and Bellman-Ford Algorithm.

Main Body:

First, Dijkstra's algorithm is a method used to find the shortest path from a starting vertex to all other vertices in a graph. It assigns labels to each vertex, representing the minimal length from the starting point to that vertex. The algorithm progresses sequentially, attempting to decrease the label values for each vertex. It terminates when all vertices have been visited.
To implement Dijkstra's algorithm, an array (d[]) is used to store the labels. The label at the starting vertex is set to zero (d[s] = 0), while labels for other vertices are initially set to infinity (d[v] = ∞) to indicate that their distances from the starting vertex are unknown. In computer programming, a large number is often used to represent infinity. A Boolean array (u[v]) is also used to track whether a vertex has been visited. Initially, all vertices are marked as unvisited (u[v] = false).
The algorithm proceeds in n iterations. If all vertices have been visited, the algorithm terminates. Otherwise, from the list of unvisited vertices, the vertex with the minimum label value is chosen (starting with the starting vertex, s). Then, all neighbors of this vertex (vertices connected by edges)

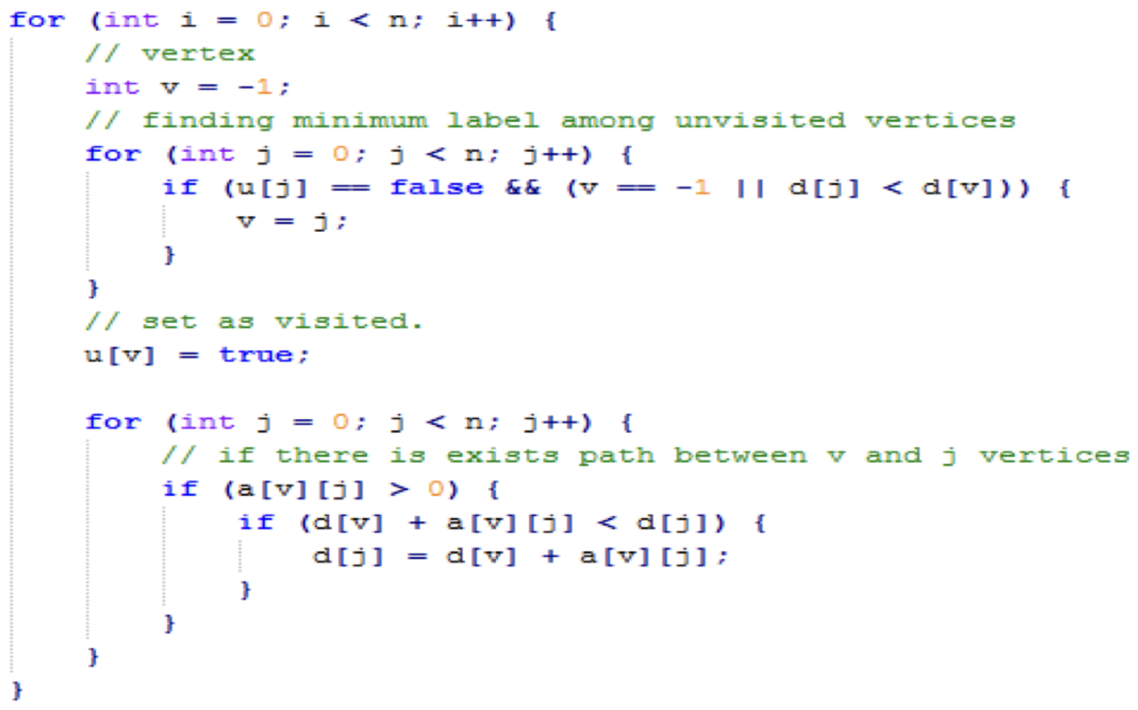
are considered. For each unvisited neighbor, a new length is calculated by adding the label value of the initial vertex (d[v]) and the length of the edge (l) that connects them. Suppose this resulting value is smaller than the current label value. In that case, it is updated accordingly:

d[neighbors] = min(d[neighbors], d[v] + l) (1)

After considering all neighbors, the initial vertex is marked as visited (u[v] = true). This process is repeated n times until all vertices have been visited, and the algorithm terminates. Vertices that are not connected to the starting vertex will remain assigned as infinity.

To reconstruct the shortest path from the starting vertex to other vertices, an array p[] is used. For each vertex v (excluding the starting vertex, $v \neq s$), the number of the penultimate vertex in the shortest path is stored in p[v]. In other words, the complete path from s to v can be obtained by following the sequence:

P = (s, ..., p[p[p[v]]], p[p[v]], p[v], v) (2)



```
for (int i = 0; i < n; i++) {
    // vertex
    int v = -1;
    // finding minimum label among unvisited vertices
    for (int j = 0; j < n; j++) {
        if (u[j] == false && (v == -1 || d[j] < d[v])) {
            v = j;
        }
    }
    // set as visited.
    u[v] = true;

    for (int j = 0; j < n; j++) {
        // if there is exists path between v and j vertices
        if (a[v][j] > 0) {
            if (d[v] + a[v][j] < d[j]) {
                d[j] = d[v] + a[v][j];
            }
        }
    }
}
```

Pseudo code:

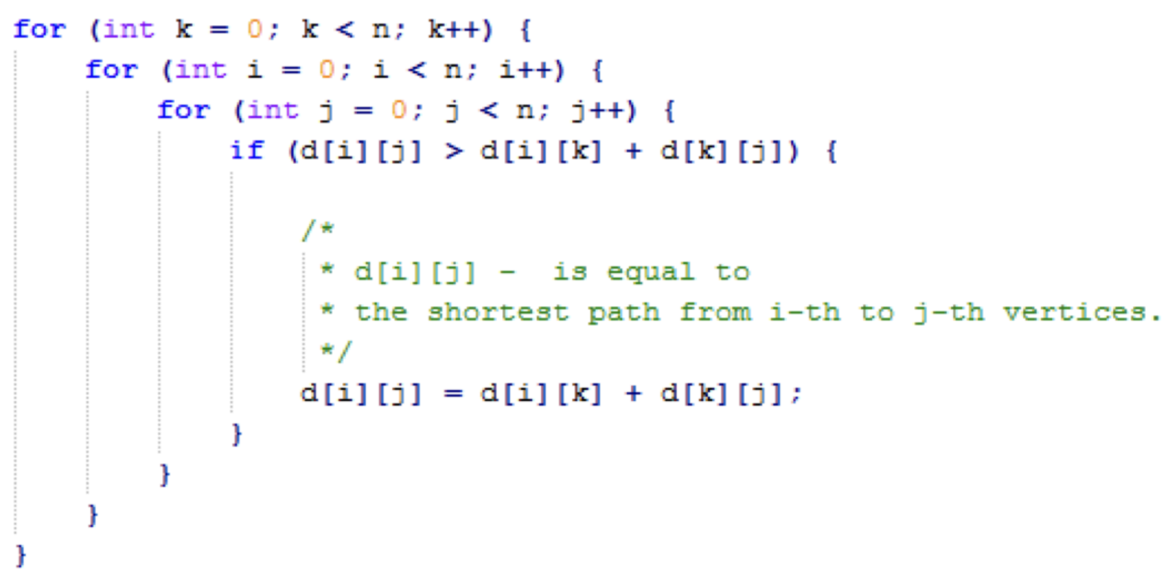
```
function Dijkstra (graph, start):
    // Initialize distances and heap
    distances = {node: infinity for node in graph}
    distances[start] = 0
    heap = [(0, start)]

    while heap is not empty:
        // Extract node with the smallest distance from the heap
        current_distance, current_node = heapq.heappop(heap)

        // Skip if already processed
        if current_distance > distances[current_node]:
            continue
```

```
// Relax neighbors
for neighbor, weight in graph[current_node].items():
    distance = current_distance + weight

    // Update distance if a shorter path is found
    if distance < distances[neighbor]:
        distances[neighbor] = distance
        // Add t
```

Second, Floyd-Warshall Algorithm: Explanation and Implementation

Consider the graph G, where vertices were numbered from 1 to n. The notation dijk means the shortest path from i to j, which also passes through vertex k. If there is exists edge between vertices i and j it will be equal to dij0, otherwise it can assigned as infinity. However, for other values of dijk there can be two choices: (1) If the shortest path from i to j does not pass through the vertex k then the value of dijk will be equal to dijk-1. (2) If the shortest path from i to j passes through the vertex k then first it goes from i to k, after that goes from k to j. In this case, the value of dijk will be equal to dikk-1 + dkjk-1. And to determine the shortest path we just need to find the minimum of these two statements [6]:

dij0 = the length of edge between vertices i and j (3) dijk = min (dijk-1, dikk-1 + dkjk-1) (4)



```
for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (d[i][j] > d[i][k] + d[k][j]) {

                /*
                 * d[i][j] -  is equal to
                 * the shortest path from i-th to j-th vertices.
                 */
                d[i][j] = d[i][k] + d[k][j];
            }
        }
    }
}
```

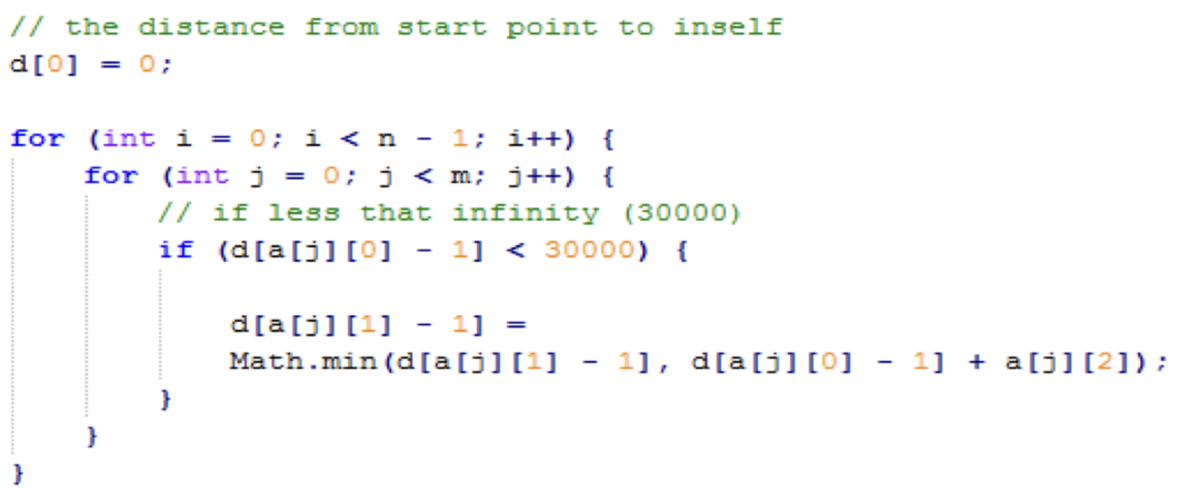
Pseudo code:

```
Folyd -->   function floyd_warshall(graph):
    num_nodes = length(graph)
    distances = create_matrix(num_nodes, inf)

    for node in graph:
        for neighbor, weight in graph[node].items():
            distances[node][neighbor] = weight

    for k in range(num_nodes):
        for i in range(num_nodes):
            for j in range(num_nodes):
                distances[i][j] = min(distances[i][j], distances[i][k] + distances[k][j])
    return distances

// Example usage:
graph = {
    0: {1: 3, 2: 7},
    1: {0: 2, 2: 4},
    2: {0: 1, 1: 5}
}

shortest_paths = floyd_warshall(graph)

print("Shortest paths between all pairs of nodes:")
for row in shortest_paths:
    print(row)
```

Third, The Bellman-Ford algorithm is an alternative to Dijkstra's algorithm that can handle graphs with negative-weight edges. Unlike Dijkstra's algorithm, Bellman-Ford allows for cycles with negative weights, which can result in multiple paths from the starting point to the destination, with each cycle potentially reducing the length of the shortest path. However, to explain the algorithm, let's assume that our graph does not have cycles with negative weights.

In the Bellman-Ford algorithm, an array called d[] is used to store the minimal length from the starting point (s) to other vertices. The algorithm consists of several phases, and in each phase, it aims to minimize the value of all edges by updating d[b] with the expression d[a] + c, where a and b represent vertices in the graph, and c is the weight of the edge connecting them.

To calculate the lengths of all the shortest paths in the graph, the algorithm requires n - 1 phases, where n is the number of vertices. However, for vertices that are unreachable from the starting point, the corresponding elements in the array will remain assigned as infinity.



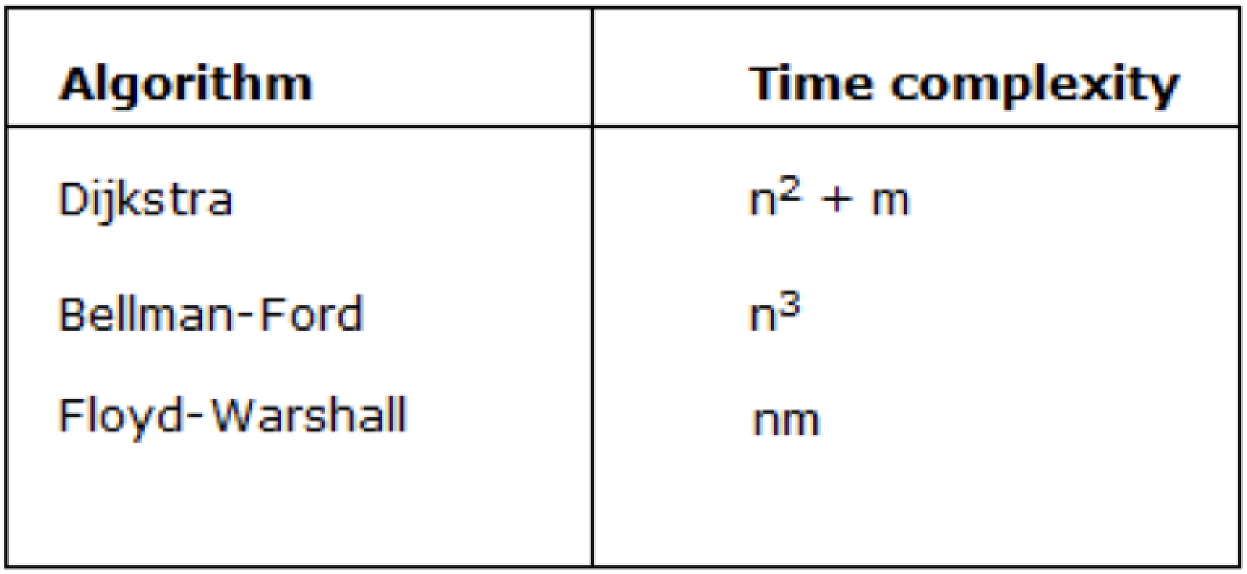
```
// the distance from start point to inself
d[0] = 0;

for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < m; j++) {
        // if less that infinity (30000)
        if (d[a[j][0] - 1] < 30000) {

            d[a[j][1] - 1] =
            Math.min(d[a[j][1] - 1], d[a[j][0] - 1] + a[j][2]);
        }
    }
}
```

Pseudo code:

```
Bellman-> function bellman_ford(graph, start):
  // Initialize distances
  distances = {node: infinity for node in graph}
  distances[start] = 0


  // Relax edges repeatedly
```

```
    for _ in range(len(graph) - 1):
        for node in graph:
            for neighbor, weight in graph[node].items():
                if distances[node] + weight < distances[neighbor]:
                    distances[neighbor] = distances[node] + weight

    // Check for negative cycles
    for node in graph:
        for neighbor, weight in graph[node].items():
            if distances[node] + weight < distances[neighbor]:
                raise ValueError("Graph contains a negative cycle")

    return distances

// Example usage:
graph = {
    'A': {'B': -1, 'C': 4},
    'B': {'C': 3, 'D': 2, 'E': 2},
    'C': {},
    'D': {'B': 1, 'C': 5},
    'E': {'D': -3}
}

start_node = 'A'
shortest_distances = bellman_ford(graph, start_node)

print("Shortest distances from {}: {}".format(start_node, shortest_distances))
```

Now turning to the most important part which one of these algorithms is the best to answer this question easily we need to know the complexity analysis for which of each.



Algorithm	**Time complexity**
Dijkstra	$n^2 + m$
Bellman-Ford	n^3
Floyd-Warshall	nm

<u>So, the best algorithm is Floyd-Warshall because it has minimum Time complexity</u>

<u>Conclusion:</u>

In conclusion, the time complexity analysis of Dijkstra's, Floyd-Warshall, and Bellman-Ford's algorithms indicates that these algorithms are efficient in solving the shortest path problem. They all provide a single solution. However, the Genetic Algorithm (GA) offers an advantage over these algorithms as it has the potential to generate multiple optimal solutions, as the outcome can vary each time the GA is executed.

In the future, there are plans to enhance and expand the proposed GA framework for finding the shortest path or distance between two locations on a map, representing various types of networks. Additionally, other artificial intelligence techniques like fuzzy logic and neural networks can be incorporated to improve existing shortest-path algorithms, making them more intelligent and efficient.

Reference

Magzhan, K., & Jani, H. M. (2013). A review and evaluations of shortest path algorithms. *Int. J. Sci. Technol. Res*, *2*(6), 99-104. https://www.researchgate.net/profile/Magzhan-Kairanbay/publication/310594546_A_Review_and_Evaluations_of_Shortest_Path_Algorithms/links/5ac49631a6fdcc1a5bd06106/A-Review-and-Evaluations-of-Shortest-Path-Algorithms.pdf

Yu, G., & Yang, J. (1998). On the robust shortest path problem. *Computers & operations research*, *25*(6), 457-468. https://www.sciencedirect.com/science/article/abs/pii/S0305054897000853?casa_token=ssajWlTsUqgAAAAA:7-cJ7PzQBEsr7Rni6T87sTkcEAwePClQeH6pz5o-r7hN1L4VDfOJDoVkhNhW6guJ7qwJyh-R3pA

Fu, L., Sun, D., & Rilett, L. R. (2006). Heuristic shortest path algorithms for transportation applications: State of the art. *Computers & Operations Research*, *33*(11), 3324-3343.https://www.sciencedirect.com/science/article/abs/pii/S030505480500122X