FACULTY SCIENCE COMPUTER AND INFORMATION TECHNOLOGY

CSC4202

DESIGN AND ANALYSIS OF ALGORITHM

SEMESTER II, 2023/2024

GROUP PROJECT

GROUP MEMBERS :

NAME	MATRIC NUMBER
NORSYAHIRAH BT SAPUAN	211283
NUR A'IN A'LIYA BT AHMAD ALAUDDIN	211278
NURUL NATASHA NABILAH BT MOHD DINO	211909

Initial Project Plan (week 10, submission date: 31 May 2024)

<table>
<tr><td>Group Name</td><td>SafeRangers</td></tr>
<tr><td>Members</td><td>
<table>
<tr><th>Name</th><th>Email</th><th>Phone number</th></tr>
<tr><td>NORSYAHIRAH BT SAPUAN</td><td>211283@student.upm.edu.my</td><td>017-6962402</td></tr>
<tr><td>NUR A’IN A’LIYA BT AHMAD ALAUDDIN</td><td>211278@student.upm.edu.my</td><td>016-3306890</td></tr>
<tr><td>NURUL NATASHA NABILAH BT MOHD DINO</td><td>211909@student.upm.edu.my</td><td>017-6239112</td></tr>
</table>
</td></tr>
<tr><td>Problem scenario description</td><td>Kuala Lumpur is hit by a powerful earthquake, leading to widespread devastation across the city. The immediate priority is to rescue trapped individuals, provide medical assistance to the injured, and restore essential services.</td></tr>
<tr><td>Why it is important</td><td>To save people’s lives by bringing them to a safer place.</td></tr>
<tr><td>Problem specification</td><td>Objective : to rescue trapped individuals, provide medical assistance to the injured, and restore essential services.
Goal : find the shortest and safest paths to reach affected areas</td></tr>
<tr><td>Potential solutions</td><td>Algorithms such as graph and greedy algorithm</td></tr>
<tr><td>Sketch (framework, flow, interface)</td><td>Vertices:
0: Jalan Wani
1: Jalan Melati
2: Jalan Maluri
3: Jalan Serdang
4: Jalan Sri Gombak

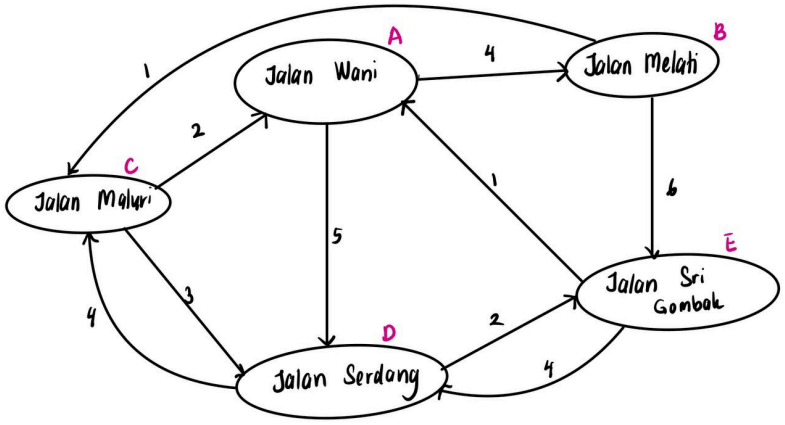
Edges:
- Jalan Serdang (3) -> Jalan Sri Gombak (4)
- Jalan Serdang (3) -> Jalan Maluri (2)
- Jalan Maluri (2) -> Jalan Wani (0)
- Jalan Maluri (2) -> Jalan Serdang (3)
- Jalan Sri Gombak (4) -> Jalan Wani (0)
- Jalan Sri Gombak (4) -> Jalan Serdang (3)
- Jalan Wani (0) -> Jalan Serdang (3)
- Jalan Wani (0) -> Jalan Melati (1)
- Jalan Melati (1) -> Jalan Maluri (2)
- Jalan Melati (1) -> Jalan Sri Gombak (4)

Adjacency Matrix (weights):</td></tr>
</table>

	0 1 2 3 4 0 [0, 4, 0, 5, 0] 1 [0, 0, 1, 0, 6] 2 [2, 0, 0, 3, 0] 3 [0, 0, 4, 0, 2] 4 [1, 0, 0, 4, 0]

Project Proposal Refinement (week 11, submission date: 7 June 2023)

Group Name	SafeRangers
Members	**Name** / **Role** NORSYAHIRAH BT SAPUAN — Algorithm specialist NUR A'IN A'LIYA BT AHMAD ALAUDDIN — Software Developer NURUL NATASHA NABILAH BT MOHD DINO — Data Analyst
Problem statement	Emergency services unable to rescue many of people because did not know which road to take
Objectives	Develop and implement algorithm to find the shortest path Analyze the time complexity and recurrence of the implemented algorithm
Expected output	The shortest path that can be used by the rescue team to save people faster and more efficiently.
Problem scenario description	Kuala Lumpur is hit by a powerful earthquake, leading to widespread devastation across the city. The immediate priority is to rescue trapped individuals, provide medical assistance to the injured, and restore essential services. However, the damage roads and collapsed buildings, and ongoing aftershocks can affect the rescuing operation because it is time consuming to find a clear path to reach the affected areas.
Why it is important	Identifying the shortest path during an earthquake is crucial because emergency services like ambulances, fire trucks, and rescue teams need to reach affected areas as quickly as possible. The shortest path ensures the fastest response time. In addition, the shortest path can significantly reduce the time it takes for individuals to reach the safety area.
Problem specification	Finding the shortest and clear path that can be used by the rescue team to reach the affected areas quickly and save people's lives.
Potential solutions	Use dijkstra algorithm to find the shortest path
Sketch (framework, flow, interface)	



Methodology

Milestone	Time
Scenario refinement discussion	W10
Find example solutions and suitable algorithms. Discuss why that solution and the example problems relate to the problem in the project	W11
Implement the coding of the chosen problem and complete the coding. Debug any error	W12
Conduct analysis of correctness and time complexity	W13
Prepare online portfolio and presentation	W14

Project Progress (Week 10 - 14)

<table>
<tr><td>Milestone 1</td><td>Scenario refinement discussion</td></tr>
<tr><td>Date (week)</td><td>W10</td></tr>
<tr><td>Description/ sketch</td><td>

Define the primary goal of the scenario
Identify the specific needs of emergency services in Malaysia.
Determine key factors influencing pathfinding such as road conditions, traffic, and hazards.
Identify the data and graph representation.

</td></tr>
<tr><td>Role</td><td>
<table>
<tr><th>Syahirah</th><th>Ain</th><th>Natasha</th></tr>
<tr><td>- Clearly identify the primary objective of developing a pathfinding algorithm for emergency services.
- Align the goals with potential impact on emergency response time and safety.</td><td>- Analyze existing protocols and practices to understand how pathfinding algorithms can enhance efficiency.
- Conduct thorough research to identify and document the specific challenges and requirements of emergency services in Malaysia.</td><td>- Ensure the accuracy and reliability of data used in graph representation.
- Coordinate with Members 1 and 2 to align prototype development with scenario goals and identified needs.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><td>Milestone 2</td><td>Find example solutions and suitable algorithms. Discuss why that solution and the example problems relate to the problem in the project</td></tr>
<tr><td>Date (week)</td><td>W11</td></tr>
<tr><td>Description/ sketch</td><td>

Develop a model for finding the shortest path for emergency services to rescue people during an earthquake.
Compare several algorithms.
Identify suitable algorithms to solve this problem.
Discuss why the chosen algorithm is suitable to find the shortest path during rescue.

</td></tr>
<tr><td>Role</td><td>
<table>
<tr><th>Syahirah</th><th>Ain</th><th>Natasha</th></tr>
<tr><td>- Research and evaluate various algorithms
-Conduct details comparison of the algorithms with different aspects
-Identify the most suitable algorithm for</td><td>- Collect and integrate geographical information
-Prepare the data for use in the model, ensuring it is clean, accurate, and up-to-date.</td><td>- Implement and test different algorithms within the model to compare their performance.
-Work closely with Syahirah and Ain to</td></tr>
</table>
</td></tr>
</table>

	the project and justify its selection.		provide necessary information

Milestone 3	Implement the coding of the chosen problem and complete the coding. Debug any error
Date (week)	12
Description/ sketch	● Implement the chosen algorithm into executable code using the Java programming language. ● Ensure the algorithm accurately reflects the designed framework and logic discussed in previous milestones. ● Implement necessary data structures such as graphs or priority queues to support the algorithm's functionality. ● Collaborate with team members to troubleshoot complex issues and validate the algorithm's correctness. ● Perform thorough testing to verify the algorithm's functionality across different scenarios and inputs.
Role	

Syahirah	Ain	Natasha
- Ensure that the implemented algorithm aligns with the designed framework and meets the specified requirements. - Review the code to verify that all necessary components, such as input handling and output formatting, are correctly implemented.	- Write the chosen algorithm into Java code, ensuring adherence to the established framework and logic. - Write clear and structured code that reflects the algorithm's steps and integrates necessary data structures.	- Implementing necessary data structures such as graphs and priority queues to support the algorithm's operations. - Ensure that data structures are efficient and well-integrated into the overall algorithm implementation.

<table>
<tr><td>Milestone 4</td><td>Conduct analysis of correctness and time complexity</td></tr>
<tr><td>Date (week)</td><td>13</td></tr>
<tr><td>Description/ sketch</td><td>Checking the correctness of Dijkstra algorithm using asymptotic notation
Analyze the idea of dijkstra algorithm paradigm by emphasising which part needs recurrence and the function for the optimization.
Analyze time complexity using asymptotic notation</td></tr>
<tr><td>Role</td><td><table>
<tr><th>Syahirah</th><th>Ain</th><th>Natasha</th></tr>
<tr><td>- Ensure a thorough understanding of Dijkstra's algorithm, its steps, and how it functions.
- Describe the correctness steps by using asymptotic notation and ensure each step maintains the desired complexity.</td><td>- Identify the paradigm used by Dijkstra's algorithm and break down the algorithm into its main steps such as initialization and the operation in the loop.
- Analyze which part of the algorithm can be expressed using a recurrence relation.</td><td>- Discuss with Syahirah and Ain about the data structures used in the algorithm.
- Sum up the individual complexities to find the overall time complexity and express the time complexity using asymptotic notation .</td></tr>
</table></td></tr>
</table>

<table>
<tr><td>Milestone 5</td><td>Prepare online portfolio and presentation</td></tr>
<tr><td>Date (week)</td><td>14</td></tr>
<tr><td>Description/ sketch</td><td>Preparing documentation with all information
Preparing slide presentation
develop online portfolio using google site</td></tr>
<tr><td>Role</td><td><table>
<tr><th>Syahirah</th><th>Ain</th><th>Natasha</th></tr>
<tr><td>-Prepare comprehensive documentation with all relevant information about the project.
-Ensure the documentation is well-organized, clear, and covers all aspects of the project.
-Coordinate with team members to gather necessary details and</td><td>-Design the slides to be visually appealing and informative, summarizing key points from the project.
-Practice and ensure smooth delivery of the presentation, potentially coordinating with other team members who will present.</td><td>-Develop the online portfolio using Google Sites.
-Ensure the online portfolio is user-friendly, visually appealing, and contains all necessary information about the project.
-Regularly update the online portfolio with new information and</td></tr>
</table></td></tr>
</table>

	incorporate them into the documentation.		ensure it is accessible to all stakeholders.

Link Github: https://github.com/peanutt02/CSC4202-4.git

1.0 Scenario

Geographical Setting: Area in Malaysia

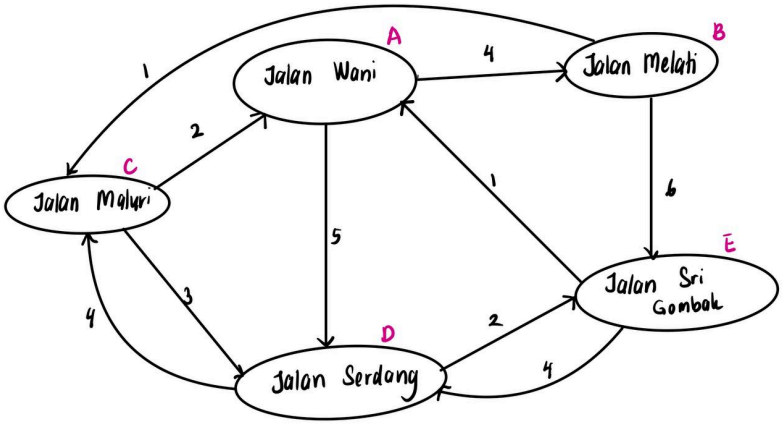
Type of Disaster: Earthquake

Damage Impact:

- Downtown are severely affected with collapsed buildings and blocked main roads
- Outskirts experiencing moderate damage with some roads blocked or damaged
- Emergency routes obstructed by fallen structures
- Communication networks partially down

Importance of Advanced Algorithm Design (AAD): In the aftermath of the earthquake, swift and effective response is crucial to save lives and mitigate further damage. Advanced Algorithm Design (AAD) plays a pivotal role in coordinating search and rescue operations, optimizing resource allocation, and ensuring that aid reaches the most critically affected areas quickly. Algorithms for finding the shortest path and other optimization techniques are essential for navigating through the city's damaged infrastructure and planning efficient rescue and relief operations.

Illustration :



Scenario Overview: Kuala Lumpur is hit by a powerful earthquake, leading to widespread devastation across the city. The immediate priority is to rescue trapped individuals, provide medical assistance to the injured, and restore essential services. Emergency services, including the Fire and Rescue Department, Civil Defence Force, and medical teams, are mobilized. However, the damaged roads, collapsed buildings, and ongoing aftershocks create significant challenges for the responders. Coordinating these efforts requires an

efficient, real-time decision-making system that can handle the complexity and urgency of the situation.

Based on the illustration above there is the road's name (a string), its value represents distance in kilometers (an integer). Then, implement an algorithm to assist emergency services in finding the shortest path from Jalan Serdang as the evacuation site.

Goal: The primary goal is to develop an algorithm that helps emergency services in Malaysia to find the shortest and safest paths to reach affected areas. Compare each route from each place that states in the illustration, to choose which is the shortest path. The starting place for emergency service should be Jalan Serdang.

Expected Output to Support Decision Making: By using the suitable algorithm, we can determine the shortest path to rescue the trapped resident timely and efficiently.

- Optimal Route Planning: Algorithms like Dijkstra's are used to calculate the shortest and safest paths for emergency responders to reach affected areas, considering road blockages and structural damages.

- Dynamic Resource Allocation: Real-time data on the extent of damage and availability of resources is used to dynamically allocate rescue teams, medical supplies, and equipment to the most critical areas. This ensures efficient use of limited resources.

- Illustration and Visualization: Real-time maps and dashboards illustrate the current status of the city, highlighting the most affected areas, available routes, and deployment of emergency services. This visual aid supports decision-makers in planning and coordination.

2.0 Model Development

In this model, the connection of the road is represented as a graph where intersections and significant points, such as cities, towns, and key intersections, are depicted as nodes. The roads connecting these nodes are represented as edges in the graph. Each edge has associated attributes that are crucial for determining the optimal path. The primary attribute for each edge is the distance (d), which represents the length of the road. This distance metric is essential for calculating the shortest path from one location to another. By illustrating the road connection like this, we can effectively apply graph-based algorithms to find the shortest and safest routes for emergency services to reach affected areas and evacuation sites in Malaysia.

The model must adhere to several key constraints to ensure it effectively assists emergency services in reaching evacuation sites. Firstly, Each road (edge) connecting any two locations must have a non-negative distance to ensure the validity and realism of distance values used in calculations. Mathematically, for any edge (i,j) in the set of edges E, the distance d_{ij} must satisfy $d_{ij} \geq 0$. Additionally, there must exist a practical path from the starting node where emergency services begin to the evacuation site which is the final destination to ensure that the evacuation site is reachable from the starting location. This requires a sequence of edges forming a continuous path from the start node S to the end node E. During emergencies, roads may have limitations on the number of vehicles they can handle at a given time, known as capacity constraints. These constraints must be considered to ensure that the selected path does not exceed the maximum allowable traffic capacity, which could delay the emergency response efforts. Consequently, the traffic on each edge (i,j) in the path P should not surpass its capacity limit $capacity_{ij}$. Furthermore, the algorithm must compute the shortest path efficiently to be effective for emergency response, meaning it should be quick enough to provide actionable routes for emergency services within the critical time frame required for response. These constraints are essential to ensure the model delivers practical and actionable routes for emergency services to reach evacuation sites effectively and efficiently.

As shown below are the example of the data representation using the two data types which are location and distance(d). Note that Jalan Serdang is the evacuation site (final destination) :

Nodes: Jalan Wani (A), Jalan Melati (B), Jalan Maluri (C), Jalan Serdang (D), Jalan Sri Gombak (E).

Edges:

- $A \rightarrow B : d_{ab} = 4$
- $A \rightarrow D : d_{ad} = 5$
- $B \rightarrow C : d_{bc} = 5$
- $B \rightarrow E : d_{be} = 6$
- $C \rightarrow A : d_{ca} = 2$
- $C \rightarrow D : d_{cd} = 3$
- $D \rightarrow C : d_{dc} = 4$

- $D \rightarrow E : d_{de} = 2$
- $E \rightarrow A : d_{ea} = 1$
- $E \rightarrow D : d_{ed} = 4$

3.0 Specification and Suitability of Algorithm

Choice of Algorithm: Greedy Algorithm (Dijkstra's Algorithm)

Some key characteristics of the scenario need to be focused on to choose the suitable algorithm. Firstly, it includes a graph representing the road connection, where nodes represent the location, and edges represent the roads connecting these nodes. The primary objective is to find the shortest path from the starting point to the evacuation site. Additionally, the distances of the roads are non-negative, ensuring that all calculated paths are realistic and feasible for emergency response purposes. Based on these characteristics, using Dijkstra's algorithm will be the most suitable method for solving the problems in this scenario.

Several reasons for choosing this Dijkstra's algorithm are:

- **Optimal for Non-Negative Weights**: Dijkstra's algorithm is specifically designed to find the shortest path in graphs with non-negative edge weights. Since road distances are naturally non-negative, this makes Dijkstra's algorithm a perfect fit for your scenario.

- **Efficiency**: Dijkstra's algorithm efficiently finds the shortest path from a single source node to all other nodes in the graph. Its time complexity, using a priority queue, is $O((V + E) \log V)$, where V is the number of vertices and E is the number of edges. This is efficient enough for large graphs typical of real-world road networks.

- **Scalability**: Given its polynomial time complexity, Dijkstra's algorithm can handle the scale of real-world road networks, making it suitable for emergency response scenarios where time is critical.

- **Dynamic Changes:** The scenario involves ongoing aftershocks and dynamic changes in the status of the infrastructure. Graph algorithms can be adapted to handle dynamic updates in edge weights and node statuses, making them flexible for real-time decision-making.

Comparison between other algorithms

Aspect	Greedy Algorithm (Dijkstra's Algorithm)	Sorting Algorithms	Divide and Conquer (DAC)	Dynamic Programming (DP)	Graph Algorithms
Primary Use Case	Shortest path in graphs with non-negative weights	Arranging elements in a specific order	Solving problems by breaking into subproblems	Solving problems with overlapping subproblems and optimal substructure	Various problems on graph structures
Example Algorithms	Dijkstra's Algorithm	Quick Sort, Merge Sort, Bubble Sort	Merge Sort, Quick Sort, Binary Search	Floyd-Warshall, Bellman-Ford, Knapsack	BFS, DFS, Kruskal's MST, Prim's MST
Time Complexity	O((V+E)logV) with a priority queue (binary heap)	Varies: O(n log n) for efficient sorts like Quick Sort and Merge Sort	Varies: O(n log n) for sorting algorithms	Varies: O(V^3) for Floyd-Warshall, O(VE) for Bellman-Ford	Varies: O(V+E) for BFS and DFS, O(E log V) for MST algorithms with heaps
Space Complexity	O(V+E)	O(n) for in-place sorts, otherwise O(n) to O(n log n) for stable sorts	Varies: typically O(log n) to O(n) due to recursion stack	Varies: O(n) to O(n^2) due to memoization tables	Varies: typically O(V+ E)
Optimal for	Problems with a single source shortest path in graphs with non-negative weights	Ordering elements efficiently	Problems that can be divided into independent subproblems	Problems with optimal substructure and overlapping subproblems	Various problems in graph theory like pathfinding, connectivity, and spanning trees
Strengths	- Finds shortest path efficiently with non-negative weights - Simple and intuitive	- Efficient for large data sets - Well-understood and widely used	- Simplifies complex problems - Generally efficient	- Handles complex problems with optimal substructure- Provides exact solutions	- Specifically designed for graph problems - Provides optimal solutions for many problems
Weaknesses	- Not suitable for graphs with negative weights - Can be slower for dense graphs	- Limited to ordering problems - Not directly applicable to other problem types	- May not be suitable for all problem types - Recursive overhead	- Higher space complexity due to memoization - Can be slower for certain problems	- Complexity can vary widely depending on the specific problem and algorithm used
Real-World Applicability	- GPS navigation - Network routing protocols	- Data organization - Search optimization	- Sorting large datasets - Searching large datasets	- Network flow - Shortest path with negative weights - Resource allocation	- Web crawling - Network connectivity - Social network analysis
Suitability for Scenario	Highly suitable: Specifically designed for shortest path problems with non-negative weights,	Not suitable: Sorting does not address the shortest path problem required	Not suitable: While powerful for certain tasks, DAC does not naturally	Moderately suitable: Can solve shortest path problems but is less efficient than	Highly suitable: Various graph algorithms can solve shortest path problems,

	matching the requirements of finding the shortest and safest paths for emergency services	for emergency services	fit the shortest path problem	Dijkstra for non-negative weights	but Dijkstra's is particularly efficient for non-negative weights

Table 3.1 : Comparison Greedy Algorithms with other algorithms

4.0 Pseudocode and Flowchart

Pseudocode

```
function Dijkstra(Graph, startVertex):
    startVertex = findVertexIndex(startVertexData)
    distances = new array(size of Graph)
    visited = new array(size of Graph)

    for i from 0 to size of Graph:
        distances[i] = INFINITY
    distances[startVertex] = 0

    for i from 0 to size of Graph:
        u = findMinDistanceVertex(distances, visited)
        if u == -1:
            break

        visited[u] = true

        for v from 0 to size of Graph:
            if not visited[v] and Graph.adjMatrix[u][v] != 0 and distances[u] != INFINITY:
                alt = distances[u] + Graph.adjMatrix[u][v]
                if alt < distances[v]:
                    distances[v] = alt

    return distances

function findVertexIndex(Graph, data):
    for i from 0 to size of Graph.vertexData:
        if Graph.vertexData[i] equals data:
            return i
    return -1

function findMinDistanceVertex(distances, visited):
    minDistance = INFINITY
    minIndex = -1
    for i from 0 to size of distances:
        if not visited[i] and distances[i] < minDistance:
            minDistance = distances[i]
            minIndex = i
    return minIndex
```

Flowchart

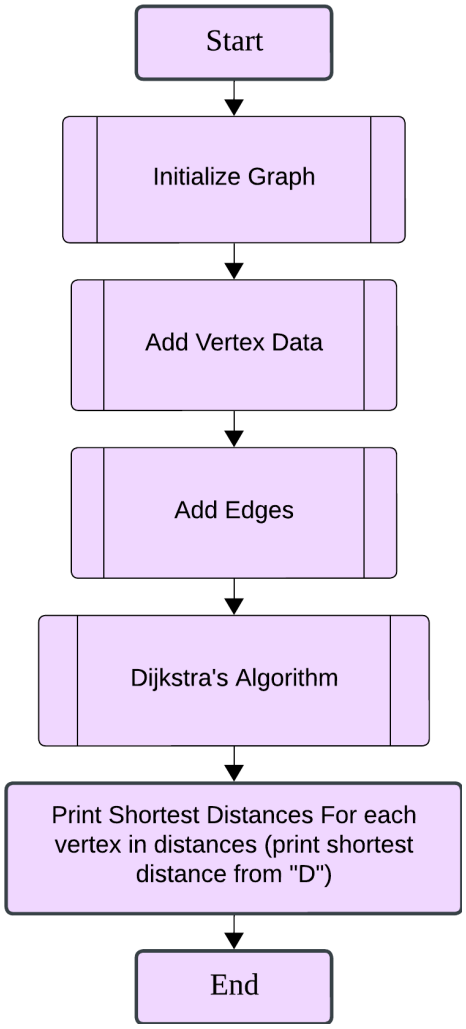


Figure 4.1 Flowchart

5.0 Correctness of an Algorithm

The paradigm used for Dijkstra's algorithm is greedy algorithm. The greedy approach builds up a solution one by one, always choosing the next one that offers the most immediate benefit. In the context of Dijkstra's algorithm, the immediate benefit is the shortest path.

The parts needing recurrence, and the optimization function :

- Initialize : Create arrays to track distances from the start vertex to all other vertices (distances) and to track whether a vertex has been visited (visited). And also set the distance to the start vertex to 0 and all other distances to infinity.
- Relaxation : For the selected vertex u, the algorithm examines each of its unvisited neighbors v. If a shorter path to v through u is found, the distance to v is updated. This is called "relaxation" and ensures that the shortest known distance to each vertex is always accurate.

We can think of a few significant enhancements to maximize the Dijkstra's algorithm implementation that is currently available. Enhancing the effectiveness of determining the minimal distance vertex and managing the graph representation more skillfully are the main areas that require optimization. The key optimization in Dijkstra's algorithm is the use of a priority queue (or min-heap) to efficiently select the unvisited vertex with the smallest known distance. In the algorithm, findMinDistanceVertex performs this task linearly, but using a priority queue reduces the time complexity significantly.

This is analysis of the algorithm’s correctness by using asymptotic notation :

Initialization:

- distances array is initialized with infinity for all vertices except the starting vertex, which is initialized to 0.
- visited array is initialized to false for all vertices.

Main Loop:

1. **Finding Minimum Distance Vertex (findMinDistanceVertex)**:
    - Within each iteration, the algorithm uses findMinDistanceVertex to find the vertex with the smallest known distance that hasn't been visited.

2. **Updating Distances**:

    - For each vertex u, the algorithm examines all vertices v to which u is connected. If v is unvisited and there is a shorter path through u, it updates distances[v].
    - This update operation potentially happens EEE times (where EEE is the number of edges), because each edge is considered at most once when updating distance.

Correctness: Dijkstra's algorithm is correct because it ensures that once a vertex's shortest path is finalized, its shortest path cannot be altered by later updates from vertices with longer paths. This is guaranteed because the algorithm always selects the shortest available path from the set of vertices that haven't been visited yet.

Dijkstra's algorithm, as presented, achieves correctness by ensuring that at each step, it updates the shortest known distances to each vertex until all vertices have been visited.

6.0 Analysis of Algorithms

Based on the code implementation, we use an adjacency matrix and basic array to find the minimum distance which will impact the time complexity. To analyze the best, average, and worst-case time complexity for the usage of Dijkstra algorithm implementation in this scenario, there is a must to analyze the algorithm steps and operation based on graph density and number of vertices and edges. Generally, for the initialization part, need to set the 'distances' and 'visited' arrays as (O(V)). Then, in the main loop, the findMinDistanceVertex function will iterate over all vertices to find the one with the smallest distance. Each iteration takes O(V). Moreover, the edge relaxation will iterate over all vertices and their edges to update the shortest path estimates. The case analysis for each case is explained below:

Best-Case Analysis:

The best case happens when the graph is sparse, meaning that the graph has very few edges than the number of vertex, and all edges lead directly to the destination without any extra computation. However, the algorithm still needs to examine each vertex and edge to confirm their minimal state.

1. Initialization: O(V)

2. Main loop:

- Finding the minimum distance vertex: $O(V) * V = O(V^2)$
- Updating distances: $O(V) * V = O(V^2)$
- Relaxing edges: $O(V) * O(V) = O(V^2)$ (since adjacency matrix is used)

Even in the best case, since Dijkstra's algorithm checks all vertices and edges to ensure the shortest path, the time complexity remains $O(V^2)$.

Average-Case Analysis:

In the average case, the graph has a random distribution of edges and weights. It can be the number of edges is the same as the number of vertex. The algorithm will process each edge for each vertex on average. Therefore, the average time complexity is the same as the worst-case time complexity due to the structure of the algorithm.

1. Initialization: O(V)

2. Main loop:

- Finding the minimum distance vertex: $O(V) * V = O(V^2)$
- Updating distances: $O(V) * V = O(V^2)$
- Relaxing edges: $O(V) * O(V) = O(V^2)$

The adjacency matrix size and the operations to find the minimum distance vertex and update distances do not change with the number of edges. Thus, the average case time complexity remains as $O(V^2)$.

Worst-Case Analysis:

The worst case for Dijkstra's algorithm occurs in a dense graph, where every vertex is connected to every other vertex (a complete graph) or when the number of edges is close to the maximum number of edges possible. In this scenario, the algorithm will perform the maximum number of operations to find and update the shortest path for each vertex.

1. Initialization: $O(V)$

2. Main loop:

- Finding the minimum distance vertex: $O(V) * V = O(V^2)$
- Updating distances: $O(V) * V = O(V^2)$
- Relaxing edges: $O(V) * O(V) = O(V^2)$ (since adjacency matrix is used)

Therefore, the worst-case time complexity also takes the complexity of $O(V^2)$.

Summary

Best Case Time Complexity : $O(V^2)$

Average Case Time Complexity : $O(V^2)$

Worst Case Time Complexity : $O(V^2)$

The primary reason the naive Dijkstra's algorithm using an adjacency matrix has $O(V^2)$ complexity in all cases is due to the necessity to:

1. Scan through all vertices to find the minimum distance vertex.

2. Update distances for all vertices, necessitating to scan of each row in the adjacency matrix.

Since these operations depend completely on the number of vertices V and not on the number of edges E, the time complexity remains $O(V^2)$ across worst, average, and best cases. This is accurate regardless of the graph's density because the adjacency matrix representation inherently requires $O(V^2)$ operations to process all vertices and linear search for the minimum distance vertex. This quadratic complexity is mainly due to the nested loops for finding the minimum distance vertex and updating distances for all vertices. For more efficient

implementations, priority queues such as min-heaps can be used to bring the complexity down to O(V +E) log V).

7.0 Implementation of Java Coding

```
public class GroupProjectAlgo {
    static class Graph {
        private int[][] adjMatrix;
        private String[] vertexData;
        private int size;

        public Graph(int size) {
            this.size = size;
            this.adjMatrix = new int[size][size];
            this.vertexData = new String[size];
        }

        public void addEdge(int u, int v, int weight) {
            if (0 <= u && u < size && 0 <= v && v < size) {
                adjMatrix[u][v] = weight;
                // Uncomment the line below for an undirected graph
                // adjMatrix[v][u] = weight;
            }
        }

        public void addVertexData(int vertex, String data) {
            if (0 <= vertex && vertex < size) {
                vertexData[vertex] = data;
            }
        }

        public int[] dijkstra(String startVertexData) {
            int startVertex = findVertexIndex(startVertexData);
            int[] distances = new int[size];
            boolean[] visited = new boolean[size];

            for (int i = 0; i < size; i++) {
                distances[i] = Integer.MAX_VALUE;
            }
            distances[startVertex] = 0;

            for (int i = 0; i < size; i++) {
                int u = findMinDistanceVertex(distances, visited);
                if (u == -1) break;

                visited[u] = true;
```

```
            for (int v = 0; v < size; v++) {
                if (!visited[v] && adjMatrix[u][v] != 0 && distances[u] != Integer.MAX_VALUE)
{
                    int alt = distances[u] + adjMatrix[u][v];
                    if (alt < distances[v]) {
                        distances[v] = alt;
                    }
                }
            }
        }
        return distances;
    }

    private int findVertexIndex(String data) {
        for (int i = 0; i < vertexData.length; i++) {
            if (vertexData[i].equals(data)) {
                return i;
            }
        }
        return -1;
    }

    private int findMinDistanceVertex(int[] distances, boolean[] visited) {
        int minDistance = Integer.MAX_VALUE, minIndex = -1;
        for (int i = 0; i < size; i++) {
            if (!visited[i] && distances[i] < minDistance) {
                minDistance = distances[i];
                minIndex = i;
            }
        }
        return minIndex;
    }
}

public static void main(String[] args) {
    Graph g = new Graph(5);

    g.addVertexData(0, "Jalan Wani");
    g.addVertexData(1, "Jalan Melati");
    g.addVertexData(2, "Jalan Maluri");
    g.addVertexData(3, "Jalan Serdang");
    g.addVertexData(4, "Jalan Sri Gombak");

    g.addEdge(3, 4, 2); // D -> E, weight 2
```

```
        g.addEdge(3, 2, 4); // D -> C, weight 4
        g.addEdge(2, 0, 2); // C -> A, weight 2
        g.addEdge(2, 3, 3); // C -> D, weight 3
        g.addEdge(4, 0, 1); // E -> A, weight 1
        g.addEdge(4, 3, 4); // E -> D, weight 4
        g.addEdge(0, 3, 5); // A -> D, weight 5
        g.addEdge(0, 1, 4); // A -> B, weight 4
        g.addEdge(1, 2, 1); // B -> C, weight 1
        g.addEdge(1, 4, 6); // B -> E, weight 6

        // Dijkstra's algorithm from D to all vertices
        System.out.println("Dijkstra's Algorithm starting from vertex Jalan Serdang:\n");
        int[] distances = g.dijkstra("Jalan Serdang");
        for (int i = 0; i < distances.length; i++) {
            System.out.println("Shortest distance from Jalan Serdang (Evacuation Site) to " +
g.vertexData[i] + ": " + distances[i]);
        }
    }
}
```

Output:

```
Shortest distance from Jalan Serdang (Evacuation Site) to Jalan Wani: 3
Shortest distance from Jalan Serdang (Evacuation Site) to Jalan Melati: 7
Shortest distance from Jalan Serdang (Evacuation Site) to Jalan Maluri: 4
Shortest distance from Jalan Serdang (Evacuation Site) to Jalan Serdang: 0
Shortest distance from Jalan Serdang (Evacuation Site) to Jalan Sri Gombak: 2
```

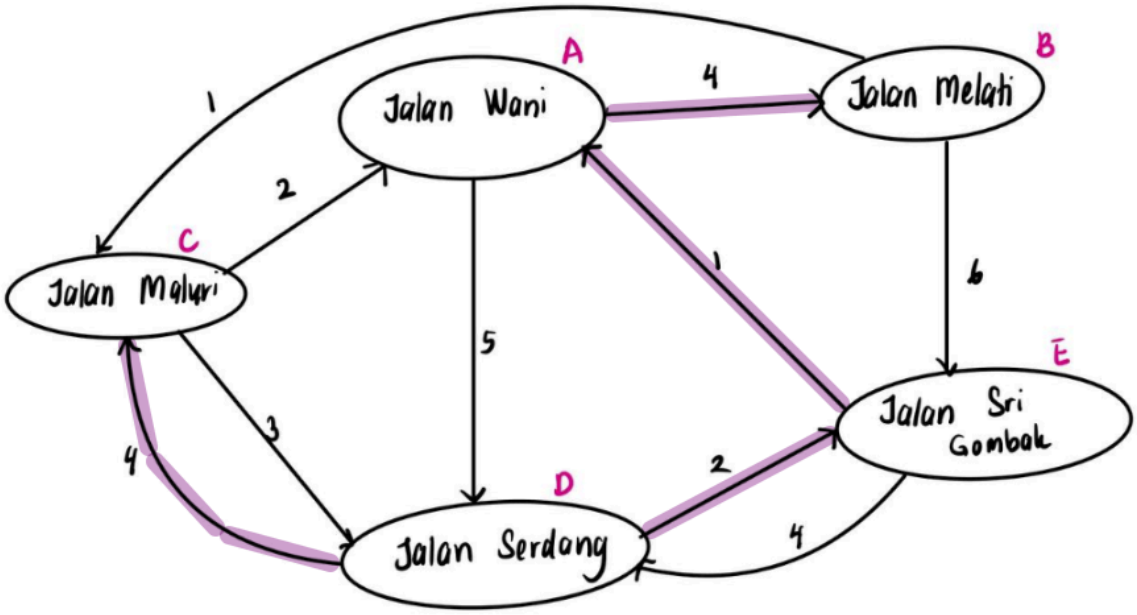


Figure 7.1 Shortest path on graph

The output of the algorithm shows the shortest path between Jalan serdang, which is the evacuation site, to all other places (vertices). For example, the shortest path from Jalan Serdang to Jalan Wani is 5 and from Jalan Serdang to Jalan Melati is 7. Therefore, to find the total of the shortest path for this scenario is by highlighting the path based on the coding output and summing all the values of the paths which is also known as weighted edges. The highlighted path, as determined by the algorithm, includes the following routes: Jalan Serdang to Jalan Sri Gombak, Jalan Sri Gombak to Jalan Wani, Jalan Wani to Jalan Melati, and Jalan Serdang to Jalan Maluri. Therefore, the total of these paths is 2 + 1 + 4 + 4 = 11 and this is also the value of the shortest path by using Dijkstra Algorithm.

References

DSA Dijkstra's Algorithm. (n.d.). Www.w3schools.com. https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php

GeeksforGeeks (2024, Feb 9). Time and Space Complexity of Dijkstra's Algorithm. *GeeksforGeeks*. www.geeksforgeeks.org/time-and-space-complexity-of-dijkstras-algorithm/.