FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

CSC 4202

DESIGN & ANALYSIS OF ALGORITHMS

Group Member	Matric no
Kabilan Yogeswaran	209983
Khugan Dass Naidu	211171
Aaren Kanavathy	210816

Github Link:

https://github.com/KabilanYogeswaran/CSC4202-Group-Project

GROUP PROJECT

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

<table>
<tr><td>Group Name</td><td></td></tr>
<tr><td>Members</td><td>
<table>
<tr><th>Name</th><th>Email</th><th>Phone number</th></tr>
<tr><td>Kabilan</td><td>209983@student.upm.edu.my</td><td>017-7469327</td></tr>
<tr><td>Khugan</td><td>211171@student.upm.edu.my</td><td>018-6687407</td></tr>
<tr><td>Aaren</td><td>210816@student.upm.edu.my</td><td>011-36400665</td></tr>
</table>
</td></tr>
<tr><td>Problem scenario description</td><td>A person wants to organize a large treasure hunt competition in a city represented as a graph. Three teams, starting from different landmarks in the city, must collect tokens from all other landmarks before reaching the final treasure location. The objective is to find the most efficient route for each team to ensure a fair and engaging competition.</td></tr>
<tr><td>Why it is important</td><td>Finding an optimal solution for the treasure hunt competition is crucial to ensure fairness by giving each team an equal opportunity, efficiency by minimizing travel times, and resource management by reducing the need for extensive manpower and logistics. It enhances player experience by preventing fatigue and frustration, improves safety in navigating potentially hazardous areas, and aids in preserving the integrity of historical landmarks by minimizing foot traffic. Additionally, streamlined routes provide more accurate data and analytics for future event planning, making the competition more engaging and sustainable while preserving the cultural heritage of the city.</td></tr>
<tr><td>Problem specification</td><td>Objective
Minimize the total travel time for each team while ensuring they visit all other landmarks before reaching the final treasure at landmark H.
Goal:
Find the shortest path for each team from their starting landmark to H, visiting all other landmarks (B,C,D,E,F) along the way.
Constraints
Visit All Landmarks:
Each team must visit every other landmark exactly once before reaching the destination H.
Unique Starting Points:
Each team starts from a different landmark (A, B, C) and aims to reach the common destination (H).</td></tr>
<tr><td>Potential solutions</td><td>Algorithms such as greedy ,dynamic and graph algorithms can be used to solve this issue</td></tr>
<tr><td>Sketch (framework, flow, interface)</td><td></td></tr>
</table>

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

<table>
<tr><td>Group Name</td><td></td></tr>
<tr><td>Members</td><td>
<table>
<tr><th>Name</th><th>Role</th></tr>
<tr><td>Aaren Kanavathy</td><td></td></tr>
<tr><td>Khugan Dass Naidu</td><td></td></tr>
<tr><td>Kabilan Yogeswaran</td><td>Software Developer</td></tr>
</table>
</td></tr>
<tr><td>Problem statement</td><td>To find the optimal routes for each team to minimize travel time while ensuring they visit all required landmarks exactly once before reaching their destination</td></tr>
<tr><td>Objectives</td><td>1. Develop a model for evacuation planning.
2. Design and implement algorithms to solve evacuation problems.
3. Analyze the performance and correctness of the implemented algorithms</td></tr>
<tr><td>Expected output</td><td>1. Efficient routes for teams to reach destination.
2. A functioning Java program that implements the algorithms.
3. Analysis of the algorithm's correctness and time complexity.</td></tr>
<tr><td>Problem scenario description</td><td>Kanavathy is organizing a large treasure hunt competition in a historical city. The city is represented as a graph where landmarks are nodes and pathways between them are edges with non-negative weights representing travel times. Three teams, Team Alpha starting at Landmark A, Team Bravo starting at Landmark B, and Team Charlie starting at Landmark C, must collect tokens from all other landmarks in the city before reaching the final treasure hidden at Landmark H. The goal is to determine the optimal routes for each team to minimize travel time while ensuring they visit all required landmarks exactly once before reaching their destination.</td></tr>
<tr><td>Why it is important</td><td>Finding an optimal solution for the treasure hunt competition is crucial to ensure fairness by giving each team an equal opportunity, efficiency by minimizing travel times, and resource management by reducing the need for extensive manpower and logistics. It enhances player experience by preventing fatigue and frustration, improves safety in navigating potentially hazardous areas, and aids in preserving the integrity of historical landmarks by minimizing foot traffic. Additionally, streamlined routes provide more accurate data and analytics for future event planning, making the competition more engaging and sustainable while preserving the cultural heritage of the city.</td></tr>
<tr><td>Problem specification</td><td>Objective
• Minimize the total travel time for each team while ensuring they visit all other landmarks before reaching the final treasure at landmark H.</td></tr>
</table>

<table>
<tr><td></td><td>Goal:

• Find the shortest path for each team from their starting landmark to H, visiting all other landmarks (B,C,D,E,F) along the way.

Constraints

3. Visit All Landmarks:
o Each team must visit every other landmark exactly once before reaching the destination H.
4. Unique Starting Points:

Each team starts from a different landmark (A, B, C) and aims to reach the common destination (H).</td></tr>
<tr><td>Potential solutions</td><td>We use :
Breadth-First Search (BFS)
Purpose: Find all possible paths from a start node to an end node.

Dijkstra's Algorithm
Purpose: Find the shortest path from a start node to an end node.</td></tr>
<tr><td>Sketch (framework, flow, interface)</td><td></td></tr>
<tr><td>Methodology</td><td>
<table>
<tr><td>Milestone</td><td>Time</td></tr>
<tr><td>scenario refinement</td><td>wk10</td></tr>
<tr><td>Find example solutions and suitable algorithm. Discuss in group why that solution and the example problems relate to the problem in the project</td><td>wk11</td></tr>
<tr><td>Edit the coding of the chosen problem and complete the coding. Debug</td><td>wk12</td></tr>
<tr><td>Conduct analysis of correctness and time complexity</td><td>wk13</td></tr>
<tr><td>Prepare online portfolio and presentation</td><td>wk14</td></tr>
</table>
</td></tr>
</table>

Project Progress (Week 10 – Week 14)

<table>
<tr><td>Milestone 1</td><td>Project Initiation and Planning</td></tr>
<tr><td>Date (week)</td><td>Week 10 - 11</td></tr>
<tr><td>Description / sketch</td><td>• Conduct project kickoff meeting to discuss the treasure hunt competition requirements and objectives.
• Define the problem scope: Establish that the city is represented as a graph, with landmarks as nodes and pathways as edges with travel times.
• Assign roles and responsibilities to team members (Aaren, Khugan, and Kabilan).
• Set up initial project schedule and timeline, identifying key milestones such as algorithm design, implementation, and analysis.
• Select communication channels and tools for collaboration throughout the project.</td></tr>
<tr><td>Role</td><td>
<table>
<tr><td>Aaren</td><td>Khugan</td><td>Kabilan</td></tr>
<tr><td>• Facilitate the project kick-off meeting to ensure all stakeholders understand the competition requirements.
• Define the problem scope in collaboration with the team.
• Coordinate the scheduling of tasks and milestones.
• Ensure communication channels are set up for efficient collaboration.</td><td>• Assist in defining the problem scope and constraints of the treasure hunt competition.
• Document the requirements and objectives based on discussions during the kickoff meeting.
• Collaborate with Aaren to establish a detailed project timeline and milestones.</td><td>• Support in outlining the technical requirements based on the defined problem scope.
• Assist in scheduling tasks and milestones according to project requirements.
• Coordinate with Aaren and Khugan to ensure alignment between project objectives and timelines.</td></tr>
</table>
</td></tr>
</table>

Milestone 2	Algorithm Design and Implementation
Date (week)	Week 12 - 13

Description/ sketch	• Design algorithms to solve the evacuation problem:
	  • Develop a model to find optimal routes for each team (Alpha, Bravo, Charlie) from their respective starting landmarks (A, B, C) to the final treasure at landmark H.
	  • Implement Breadth-First Search (BFS) for finding all possible paths from a start node to an end node.
	  • Implement Dijkstra's Algorithm to find the shortest path from a start node to an end node with weighted edges (travel times).
	• Begin coding the algorithms in Java to ensure functionality and correctness.
	• Conduct initial testing and debugging to verify algorithm functionality.

Role			
	Aaren	Khugan	Kabilan
	• Lead the algorithm design phase, focusing on developing efficient routes for each team to minimize travel time. • Ensure that BFS and Dijkstra's Algorithm are implemented correctly and effectively address the problem requirements. • Oversee initial testing to validate algorithm functionality and correctness.	• Collaborate with Aaren in designing algorithms tailored to the evacuation problem. • Assist in coding and implementing BFS and Dijkstra's Algorithm in Java. • Conduct rigorous testing to identify and resolve any implementation issues or bugs.	• Do all the coding and implementing the algorithms using Java programming language. • Verify the integration of BFS and Dijkstra's Algorithm within the project framework. • Assist in debugging and troubleshooting to ensure algorithms meet performance and correctness standards.

<table>
<tr><td>Milestone 3</td><td>Performance Analysis and Refinement</td></tr>
<tr><td>Date (week)</td><td>Week 14</td></tr>
<tr><td>Description/ sketch</td><td>• Analyze the performance of implemented algorithms (BFS and Dijkstra's Algorithm) in terms of time complexity and efficiency.
• Conduct comprehensive testing using various scenarios to evaluate algorithm correctness and effectiveness in finding optimal routes.
• Refine algorithms based on testing results and feedback to improve efficiency and accuracy.
• Prepare documentation detailing the algorithm design, implementation steps, and performance analysis.</td></tr>
<tr><td>Role</td><td>
<table>
<tr><th>Aaren</th><th>Khugan</th><th>Kabilan</th></tr>
<tr><td>• Support in testing and analyzing the performance of BFS and Dijkstra's Algorithm.
• Implement algorithm refinements based on testing feedback and performance metrics.
• Prepare detailed documentation outlining the steps taken for algorithm implementation and refinement.</td><td>• Assist in conducting extensive testing to evaluate the correctness and performance of BFS and Dijkstra's Algorithm.
• Document test cases and results to provide comprehensive feedback for algorithm refinement.
• Collaborate with Aaren and Kabilan to refine algorithms based on performance analysis.</td><td>• Lead the performance analysis of BFS and Dijkstra's Algorithm, focusing on time complexity and efficiency metrics.
• Collaborate with Khugan and Aaren to analyze testing results and identify areas for algorithm refinement.
• Ensure that the documentation accurately reflects the algorithm design and implementation details.</td></tr>
</table>
</td></tr>
</table>

a) Problem Statement

Geographical Settings: Malaysia

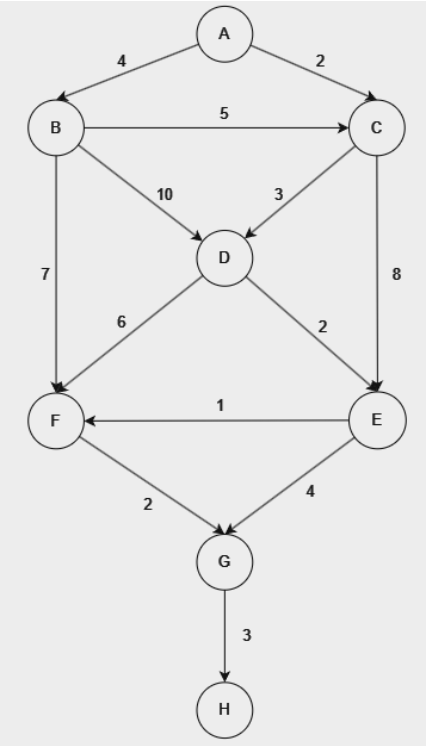
Type of Disaster: Not Applicable

Damage Impact:

- If one team inadvertently has a shorter or less challenging route to the treasure due to suboptimal path planning, it would lead to unfair competition.
- If travel times are not optimized, it could result in delays, causing frustration among participants and spectators.
- In case of an emergency, knowing the shortest paths allows for quicker response times from medical or security teams.
- Longer or more complex routes could increase the risk of participants getting lost or encountering unforeseen hazards.

Importance of Advanced Algorithm Design (AAD): Advanced Algorithm Design (AAD) is crucial for optimizing the treasure hunt competition by ensuring fair and efficient pathfinding for all teams. It handles the complexity of the city map, enabling the use of sophisticated algorithms like Dijkstra's, A*, and Breadth First Search (BFS) to find the shortest, safest routes. AAD optimizes resource allocation, enhances real-time adaptability, ensures fairness, improves safety and emergency response, and boosts participant satisfaction. By incorporating environmental considerations, AAD also minimizes the impact on the city's infrastructure, ensuring a smooth, enjoyable, and sustainable event.

Illustration:



Scenario Overview

Kanavathy is organizing a large treasure hunt competition in a historical city where three teams—Team Alpha, Team Bravo, and Team Charlie—start from different landmarks (A, B, and C respectively) and must collect tokens from all other landmarks before reaching the final treasure at Landmark H. The city's layout is represented as a weighted graph with landmarks as nodes and pathways as edges with travel times. The goal is to ensure a fair, efficient, and enjoyable experience for all participants by optimizing their routes.

Goal

The goal of this project is to develop a robust pathfinding system using Breadth-First Search (BFS) and Dijkstra's algorithm. BFS will be used to identify all possible paths from each starting landmark to the final treasure, ensuring that all nodes (landmarks) are visited. Dijkstra's algorithm will then be employed to determine the shortest possible path for each team, minimizing their travel time and ensuring a balanced competition. This approach aims to optimize resource allocation, enhance participant safety, and ensure a smooth event flow.

Expected Output to support decision making

The expected output includes a comprehensive list of all possible paths from each starting landmark to the treasure at Landmark H, generated by BFS, and the shortest path for each team, calculated using Dijkstra's algorithm. Additionally, the output will include travel times for each path, ensuring organizers can make informed decisions on resource placement and emergency preparedness. This detailed path information will support effective decision-making, ensuring the competition is fair, efficient, and enjoyable for all participants.

b) Model Development

Data Types and Structures: The city's layout will be represented as a weighted graph, where landmarks are nodes and pathways between them are edges with non-negative weights representing travel times. The data will be stored in adjacency lists or adjacency matrices, ensuring efficient traversal and pathfinding operations.

Objective Function: The primary objective is to minimize the total travel time for each team from their respective starting landmarks to the final treasure at Landmark H, while ensuring they visit all other landmarks to collect tokens. The solution must ensure all landmarks are visited.

Constraints: Several constraints must be considered:

1. **Mandatory Visit**: Each team must visit all landmarks at least once before reaching the treasure.
2. **Travel Time**: Paths should minimize the sum of travel times between landmarks.
3. **Fairness**: The shortest path calculated should be the same for all teams to ensure a fair competition.
4. **Safety**: Paths must avoid unsafe areas or pathways marked as hazardous.
5. **Environmental Impact**: Paths should minimize repeated use of specific pathways to prevent environmental degradation.

Examples and Requirements:

- **Example Path**: For Team Alpha starting at Landmark A, a possible path might be A→B→D→F→E→G→HA \to B \to D \to F \to E \to G \to HA→B→D→F→E→G→H, with each segment's travel time summed to provide the total travel time.
- **Space Constraints**: Efficient use of memory to store the graph structure, ensuring it can handle large datasets representing the city map.
- **Time Constraints**: Algorithms must operate within reasonable time limits to provide real-time updates and path recalculations during the event.
- **Value Constraints**: Weights (travel times) must be non-negative and reflect realistic travel durations between landmarks.

Deployment and Real-Time Adjustments:

- Deploy the system on a robust platform capable of handling real-time data processing and updates.
- Implement real-time monitoring to adapt to changes such as pathway closures or hazards, ensuring quick recalculations and updates.

Safety and Emergency Preparedness:

- Prioritize safe routes and develop protocols for rapid emergency response.
- Ensure shortest paths facilitate quick access for emergency teams, enhancing participant safety.

Environmental Considerations:

- Design routes to evenly distribute participant movement, minimizing environmental impact.
- Avoid overuse of specific pathways to protect the city's infrastructure.

c) It's important to figure out the best solution for this treasure hunt challenge for several reasons:

- **Fair Competition:** It's critical to guarantee that every team has an equal opportunity to gather every token and get to the ultimate treasure without facing any unfair advantages. The fairness of the competition would be jeopardised if one team found a more efficient way to finish the work much faster than the others.

- **Time Efficiency:** Identifying the shortest path for every team reduces the overall amount of time needed, as travel times between landmarks differ. This keeps competitors interested and cuts down on downtime by ensuring the competition goes smoothly and ends on time.

- **Resource Management:** Effective resource management is made possible by an ideal solution. For instance, it lessens the requirement for an excessive number of workers to oversee and coordinate the competition throughout time across several landmarks. Less demand on resources and logistics results from shorter routes.

- **Player Experience:** If the treasure hunt is effectively planned and carried out, participants are more likely to have fun. Long and too complex itineraries can cause exhaustion and frustration, which takes away from the whole experience.

- **Safety:** Reducing the amount of time people spend travelling also improves safety, particularly in ancient cities where pathways may be congested or challenging to negotiate. It guarantees that players can finish the search both swiftly and securely.

- **Historical City Preservation:** Historic places can be preserved by cutting down on travel time and, consequently, foot traffic at different locations. This is particularly crucial in ancient cities because more foot traffic may cause the sites to deteriorate.

- **Data and analytics:** By streamlining routes, event planners may collect more accurate information on journey times and trends, which will help them plan more events and comprehend how visitors engage with the city's attractions.

d) Specification and Suitability of Algorithms

Suitability of sorting, DAC, DP, greedy and graph algorithms as a solution paradigm for the chosen problem:

Solution Paradigm	**Suitability**	**Strengths**	**Weaknesses**
Sorting	Low	Simple and well-understood	Does not address the pathfinding problem
		Useful as a subroutine	Not applicable for dynamic or complex graphs
Divide and Conquer (DAC)	Low to Moderate	Useful for breaking down complex problems	Combining solutions of subproblems can be tricky
		Parallelizable	May not handle interconnected paths efficiently
Dynamic Programming (DP)	High	Can handle overlapping subproblems effectively	Can be memory-intensive
		Guarantees finding the optimal solution	Implementation can be complex
		Suitable for problems with multiple optimal sub paths	Requires understanding of the problem's structure
Greedy Algorithms	Moderate to High	Simple and easy to implement	Does not guarantee global optimality for all problems
		Efficient in terms of time and space	Might miss the optimal solution
		Good for problems where local optimality leads to global optimality	Can fail for complex pathfinding tasks with dependencies
Graph Algorithms	Very High	Directly applicable to pathfinding and shortest path problems	Can be computationally intensive for very large graphs
		Can handle weighted graphs and various constraints	Requires a good understanding of graph theory and algorithms

Choice of Algorithm: Greedy Algorithm (Dijkstra's Algorithm & Breadth First Search)

In Kanavathy's treasure hunt competition, both Breadth-First Search (BFS) and Dijkstra's algorithm play crucial roles in ensuring an optimal and fair experience for all participants. BFS is utilized to explore and identify all possible paths from each team's starting landmark to the final treasure at Landmark H. This comprehensive exploration is vital for mapping out the entire city's layout, ensuring that every possible route is considered, and that all landmarks are visited. On the other hand, Dijkstra's algorithm is employed to determine the shortest path for each team, taking into account the varying travel times between landmarks. By focusing on minimizing the total travel time, Dijkstra's algorithm ensures that each team takes the most efficient route, thereby balancing the competition and reducing overall travel times. Together, these algorithms ensure that the treasure hunt is fair, efficient, and enjoyable, with optimal resource allocation, enhanced safety, and real-time adaptability to any changes in the environment.

- Breadth-First Search (BFS) is ideal for exploring all possible routes in an unweighted graph, ensuring that every landmark and potential pathway is considered. This characteristic is crucial for mapping out the entire city and identifying all feasible routes that the teams can take, providing a complete overview of the travel options available.

- Dijkstra's Algorithm is designed to find the shortest path in weighted graphs, making it perfect for scenarios where travel times between landmarks vary. By focusing on minimizing the total travel time from the starting point to the destination, Dijkstra's algorithm ensures that each team takes the most efficient route, balancing the competition and reducing overall travel times.

- Both BFS and Dijkstra's algorithms ensure that routes are determined based on systematic exploration and optimization principles. BFS provides a level playing field by exploring all possible paths, while Dijkstra's algorithm ensures that the shortest, most efficient path is selected for each team. This characteristic helps maintain fairness and balance in the competition.

- By ensuring that the shortest and safest paths are identified, Dijkstra's algorithm enhances participant safety. It also allows for optimal resource allocation, ensuring that staff, medical teams, and other resources are placed strategically along the most traveled paths, enhancing the overall event management.

e) Design the algorithm to solve the problem and explain the idea of your algorithm paradigm by emphasising which part needs recurrence and the function for the optimization.

Algorithm:

```
function getAllPathsBFS(graph, start, end)
  paths = emptyList()
  queue = newQueue()
  enqueue(queue, [start])
  while queue is not empty do
    path = dequeue(queue)
    lastNode = getLastNode(path)
    if lastNode == end then
      add paths, path
    else
      for each neighbor in getNeighbors(graph, lastNode) do
        if neighbor not in path then
          newPath = copy(path)
          add newPath, neighbor
          enqueue(queue, newPath)
        end if
      end for
    end if
  end while
  return paths
end function

function getShortestPathDijkstra(graph, start, end)
  distances = initializeDistances(graph, start)
  previousNodes = emptyMap()
  priorityQueue = newPriorityQueue()
  enqueue(priorityQueue, newEdge(start, 0))
  while priorityQueue is not empty do
    edge = dequeue(priorityQueue)
    currentNode = edge.destination
    for each neighbor in getNeighbors(graph, currentNode) do
      newDist = distances[currentNode] + neighbor.weight
      if newDist < distances[neighbor.destination] then
        distances[neighbor.destination] = newDist
        enqueue(priorityQueue, newEdge(neighbor.destination, newDist))
        previousNodes[neighbor.destination] = currentNode
      end if
    end for
  end while

  path = reconstructPath(previousNodes, end)
  return path
end function
```

```
function main()
  graph = createGraph()
  addEdgesToGraph(graph)
  startNodes = ["A", "B", "C"]


  for each start in startNodes do
    startTimeBFS = getCurrentTime()
    paths = getAllPathsBFS(graph, start, "H")
    endTimeBFS = getCurrentTime()

    print "All paths from " + start + " to H:"
    for each path in paths do
      print path + " with distance " + getPathDistance(graph, path)
    end for
    print "BFS Execution time: " + (endTimeBFS - startTimeBFS) + " ms"

    startTimeDijkstra = getCurrentTime()
    shortestPath = getShortestPathDijkstra(graph, start, "H")
    endTimeDijkstra = getCurrentTime()

    print "Shortest path from " + start + " to H: " + shortestPath + " with distance " +
getPathDistance(graph, shortestPath)
    print "Dijkstra Execution time: " + (endTimeDijkstra - startTimeDijkstra) + " ms"
  end for
end function
```

The algorithm paradigm used in the provided code is based on graph algorithms, specifically Breadth-First Search (BFS) and Dijkstra's algorithm. These algorithms are designed to solve pathfinding problems in graphs, leveraging iterative techniques to ensure optimal performance.

1. Breadth-First Search (BFS)
Purpose: Find all possible paths from a start node to an end node.
Recurrence and Optimization:

- **Iterative Approach:** BFS uses an iterative approach with a queue to explore all possible paths from the start node to the end node level by level. This avoids the need for recursion and keeps the algorithm manageable even for large graphs.
- **Queue for Path Expansion:** The queue is used to manage the paths being explored. Paths are enqueued and dequeued as they are expanded to explore new nodes. This ensures that all paths are explored systematically without redundancy.
- **Avoiding Cycles:** The algorithm checks if a node is already in the current path to avoid cycles and redundant paths.
- **Path Collection:** Once a path reaches the end node, it is added to the list of valid paths.

2. Dijkstra's Algorithm

Purpose: Find the shortest path from a start node to an end node.

Recurrence and Optimization:

- **Priority Queue:** Dijkstra's algorithm uses a priority queue to always expand the node with the shortest known distance. This is crucial for ensuring that the shortest path is found efficiently.
- **Distance Map:** A map is maintained to store the shortest known distance from the start node to every other node. This is updated as shorter paths are discovered.
- **Previous Node Tracking:** A map of previous nodes is used to reconstruct the shortest path once the end node is reached. This avoids the need for recurrence by backtracking from the end node to the start node.
- **Edge Relaxation:** For each node, all its neighbors are examined. If a shorter path to a neighbor is found, the distance and previous node are updated, and the neighbor is added to the priority queue for further exploration.

Key Points of Recurrence and Optimization

- **BFS (getAllPathsBFS):** This function iteratively explores all possible paths from the start node to the end node using a queue, avoiding the need for recursive function calls. The main optimization is to systematically explore paths level by level, ensuring all paths are found without redundancy.
- **Dijkstra's Algorithm (getShortestPathDijkstra):** This function iteratively updates the shortest known distances using a priority queue and a map of distances. The main optimization is the use of a priority queue to always expand the shortest path first, ensuring the algorithm converges efficiently to the shortest path.

f) Define the algorithm specification.

Input

1. **Graph:** A directed or undirected graph G=(V,E)G = (V, E)G=(V,E) where:
    - VVV is a set of vertices (landmarks).
    - EEE is a set of edges (pathways) with non-negative weights (travel times).
2. **Start Node:** A specific landmark where the team starts.
3. **End Node:** The landmark where the treasure is hidden.

Output

1. **BFS Output:** A list of all possible paths from the start node to the end node.
2. **Dijkstra Output:** The shortest path from the start node to the end node along with the travel time.

Assumptions

1. All edge weights (travel times) are non-negative.
2. The graph is connected, meaning there is at least one path from any landmark to any other landmark.

g) Develop a program Java language

```
import java.util.*;

public class TreasureHunt {

    public static void main(String[] args) {
        Graph graph = new Graph();
        graph.addEdge("A", "B", 4);
        graph.addEdge("A", "C", 2);
        graph.addEdge("B", "C", 5);
        graph.addEdge("B", "D", 10);
        graph.addEdge("B", "F", 7);
        graph.addEdge("C", "D", 3);
        graph.addEdge("C", "E", 8);
        graph.addEdge("D", "E", 2);
        graph.addEdge("D", "F", 6);
        graph.addEdge("E", "F", 1);
        graph.addEdge("E", "G", 4);
        graph.addEdge("F", "G", 2);
        graph.addEdge("G", "H", 3);

        String[] startNodes = {"A", "B", "C"};

        for (String start : startNodes) {
            long startTimeBFS = System.nanoTime();
            List<List<String>> paths = getAllPathsBFS(graph, start, "H");
            long endTimeBFS = System.nanoTime();

            System.out.println("All paths from " + start + " to H:");
            for (List<String> path : paths) {
                System.out.println(path + " with distance " +
getPathDistance(graph, path));
            }

            long startTimeDijkstra = System.nanoTime();
            List<String> shortestPath = getShortestPathDijkstra(graph, start,
"H");
            long endTimeDijkstra = System.nanoTime();

            System.out.println("Shortest path from " + start + " to H: " +
shortestPath + " with distance " + getPathDistance(graph, shortestPath));
            System.out.println("Dijkstra Execution time: " + (endTimeDijkstra -
startTimeDijkstra) / 1e6 + " ms");
            System.out.println();
        }
    }

    private static List<List<String>> getAllPathsBFS(Graph graph, String start,
String end) {
        List<List<String>> paths = new ArrayList<>();
        Queue<List<String>> queue = new LinkedList<>();
        queue.add(Arrays.asList(start));

        while (!queue.isEmpty()) {
            List<String> path = queue.poll();
            String last = path.get(path.size() - 1);

            if (last.equals(end)) {
```

```
                paths.add(new ArrayList<>(path));
            } else {
                for (Edge edge : graph.getNeighbors(last)) {
                    if (!path.contains(edge.destination)) {
                        List<String> newPath = new ArrayList<>(path);
                        newPath.add(edge.destination);
                        queue.add(newPath);
                    }
                }
            }
        }

        return paths;
    }

    private static List<String> getShortestPathDijkstra(Graph graph, String start,
String end) {
        Map<String, Integer> distances = new HashMap<>();
        Map<String, String> previousNodes = new HashMap<>();
        PriorityQueue<Edge> priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(e -> e.weight));

        for (String node : graph.getNodes()) {
            distances.put(node, Integer.MAX_VALUE);
        }
        distances.put(start, 0);
        priorityQueue.add(new Edge(start, 0));

        while (!priorityQueue.isEmpty()) {
            Edge edge = priorityQueue.poll();
            String currentNode = edge.destination;

            for (Edge neighbor : graph.getNeighbors(currentNode)) {
                int newDist = distances.get(currentNode) + neighbor.weight;
                if (newDist < distances.get(neighbor.destination)) {
                    distances.put(neighbor.destination, newDist);
                    priorityQueue.add(new Edge(neighbor.destination, newDist));
                    previousNodes.put(neighbor.destination, currentNode);
                }
            }
        }

        List<String> path = new ArrayList<>();
        for (String at = end; at != null; at = previousNodes.get(at)) {
            path.add(at);
        }
        Collections.reverse(path);
        return path;
    }

    private static int getPathDistance(Graph graph, List<String> path) {
        int distance = 0;
        for (int i = 0; i < path.size() - 1; i++) {
            String from = path.get(i);
            String to = path.get(i + 1);
            for (Edge edge : graph.getNeighbors(from)) {
                if (edge.destination.equals(to)) {
                    distance += edge.weight;
                    break;
```

```
                }
            }
        }
        return distance;
    }
}

class Graph {
    private final Map<String, List<Edge>> adjList = new HashMap<>();
    private final Map<String, Integer> nodeIndexMap = new HashMap<>();
    private int index = 0;

    public void addEdge(String source, String destination, int weight) {
        adjList.putIfAbsent(source, new ArrayList<>());
        adjList.putIfAbsent(destination, new ArrayList<>());
        adjList.get(source).add(new Edge(destination, weight));
        if (!nodeIndexMap.containsKey(source)) {
            nodeIndexMap.put(source, index++);
        }
        if (!nodeIndexMap.containsKey(destination)) {
            nodeIndexMap.put(destination, index++);
        }
    }

    public List<Edge> getNeighbors(String node) {
        return adjList.get(node);
    }

    public Set<String> getNodes() {
        return adjList.keySet();
    }

    public int getNodeIndex(String node) {
        return nodeIndexMap.get(node);
    }
}

class Edge {
    String destination;
    int weight;

    Edge(String destination, int weight) {
        this.destination = destination;
        this.weight = weight;
    }
}
```

Output:

```
All paths from A to H:
[A, B, F, G, H] with distance 16
[A, C, E, G, H] with distance 17
[A, B, C, E, G, H] with distance 24
[A, B, D, E, G, H] with distance 23
[A, B, D, F, G, H] with distance 25
[A, C, D, E, G, H] with distance 14
[A, C, D, F, G, H] with distance 16
[A, C, E, F, G, H] with distance 16
[A, B, C, D, E, G, H] with distance 21
[A, B, C, D, F, G, H] with distance 23
[A, B, C, E, F, G, H] with distance 23
[A, B, D, E, F, G, H] with distance 22
[A, C, D, E, F, G, H] with distance 13
[A, B, C, D, E, F, G, H] with distance 20
Shortest path from A to H: [A, C, D, E, F, G, H] with distance 13
Dijkstra Execution time: 3.5004 ms

All paths from B to H:
[B, F, G, H] with distance 12
[B, C, E, G, H] with distance 20
[B, D, E, G, H] with distance 19
[B, D, F, G, H] with distance 21
[B, C, D, E, G, H] with distance 17
[B, C, D, F, G, H] with distance 19
[B, C, E, F, G, H] with distance 19
[B, D, E, F, G, H] with distance 18
[B, C, D, E, F, G, H] with distance 16
Shortest path from B to H: [B, F, G, H] with distance 12
Dijkstra Execution time: 0.0743 ms

All paths from C to H:
[C, E, G, H] with distance 15
[C, D, E, G, H] with distance 12
[C, D, F, G, H] with distance 14
[C, E, F, G, H] with distance 14
[C, D, E, F, G, H] with distance 11
Shortest path from C to H: [C, D, E, F, G, H] with distance 11
Dijkstra Execution time: 0.0301 ms
```

h) Analysis of the algorithm's correctness as well as time complexity (best, average and worst time) by using asymptotic notation.

1. Breadth First Search Algorithm

Algorithm Correctness Analysis

- Initialization: The algorithm starts by initializing an empty list to store paths and a queue with the initial path containing only the start node.
- BFS Loop: The loop processes each path from the queue:
  - If the last node in the path is the end node, the path is added to the result list.
  - Otherwise, all neighbours not already in the path are appended to the current path and enqueued.
- Termination: The loop continues until all possible paths from start to end have been processed.

Correctness: The BFS algorithm correctly explores all possible paths from the start node to the end node by using a queue to maintain the breadth-first traversal, ensuring all paths are found without duplicates.

Time Complexity

- **Best Case: O(v + ϵ)**

  In the best-case scenario for BFS, the goal node (H) is located very close to the starting node. For instance, if Team Alpha starts at landmark A and the treasure at landmark H is directly connected via a single edge, BFS will find the shortest path immediately. Here, the algorithm explores each node and edge only once before finding the goal. The number of vertices (v) and edges (ϵ) contribute directly to the complexity, making it O(v + ϵ).

- **Average Case: O(b^d)**

  The average case for BFS occurs when the goal node is neither too close nor too far from the starting node. For example, Team Bravo starting at B might need to explore multiple paths of varying lengths before reaching H. The branching factor (b) represents the average number of child nodes per node, and the depth (d) represents the levels explored until the goal is found. The complexity O(b^d) reflects the exponential growth of nodes explored, typical in an average-case scenario where BFS systematically expands each level before moving deeper.

- **Worst Case: O(b^d)**

  In the worst-case scenario, the BFS must explore all possible nodes and edges before finding the goal. This can happen if the goal node (H) is at the maximum depth from the starting node, and all other paths need to be considered. For instance, if Team Charlie starts at C and H is the farthest node, BFS will systematically explore each node level-by-level until reaching H. This exhaustive exploration leads to the time complexity being O(b^d), where every possible node at each level is expanded.

2. Dijkstra Algorithm

Algorithm Correctness Analysis

- Initialization: Distances are initialized with infinity, except for the start node which is set to zero. The priority queue is initialized with the start node.
- Dijkstra Loop: The loop processes nodes from the priority queue:
  - For each neighbour, if a shorter path is found, the distance is updated, and the neighbour is enqueued.
- Reconstruction: After processing all nodes, the shortest path is reconstructed from the end node back to the start node using the previous nodes map.

Correctness: Dijkstra's algorithm guarantees the shortest path in a graph with non-negative weights by always expanding the least-cost node first and updating paths accordingly.

Time Complexity

- **Best Case: O(v log_v)**

  The best-case scenario for Dijkstra's algorithm occurs when all edges have equal weight, or the shortest path is found early in the process. For example, if Team Alpha starting at A finds a direct, minimal path to H without needing to update the distances of many other nodes, the algorithm quickly terminates. Here, the priority queue operations dominate, leading to a complexity of O(v log_v) due to efficient extraction and update operations on a relatively small set of nodes.

- **Average Case: O((v + E) log_v)**

  In an average scenario, Dijkstra's algorithm needs to update the shortest paths for a substantial portion of the graph. For example, Team Bravo starting at B might need to explore and update distances through several nodes and edges before determining the shortest path to H. The complexity O((v + E) log_v) reflects the need to process each vertex and edge while maintaining and updating the priority queue, which logarithmically scales with the number of vertices.

- **Worst Case: O((v + E) log_v)**

  The worst-case scenario for Dijkstra's algorithm involves dense graphs where each vertex is connected to many others, and the shortest path requires extensive updates. For instance, if Team Charlie starts at C and needs to explore a network where nearly every node is connected, the algorithm must consider and update distances for all vertices and edges. This results in the complexity O((v + E) log_v), where the combination of vertex and edge processing along with priority queue operations dictates the algorithm's efficiency.

3. Overall Algorithm

Time Complexity

- **Best Case:** $O(V + E + N \,.\, \mathrm{V} \log_{\square} V)$

  In the best-case scenario, the overall algorithm, which combines BFS and Dijkstra's, quickly finds the shortest paths for each team (Alpha, Bravo, and Charlie) starting from their respective landmarks (A, B, and C) to the treasure at landmark H. The BFS component efficiently finds all possible paths without extensive exploration, contributing O(V + E) complexity. The Dijkstra's algorithm, run N times (once for each team), quickly finds the shortest paths due to minimal edge relaxation and updates, adding O(N * V log V) complexity. The combined best-case complexity is O(V + E + N * V log V).

- **Average Case:** $O(V + E + N \,.\, b^d + N \,.\, (\mathrm{V} + \mathrm{E}) \log_{\square} V)$

  In the average case, the algorithm needs to explore a moderate number of paths and updates. BFS finds multiple paths to the goal nodes, which is typical in a scenario where the teams need to navigate through several intermediate landmarks. This adds O(V + E) for the BFS part and O(N * b^d) for exploring all possible paths for N teams. Dijkstra's algorithm, also run N times, involves processing a substantial portion of the graph, leading to O(N * (V + E) log V) due to edge relaxation and priority queue operations. The average-case complexity is thus O(V + E + N * b^d + N * (V + E) log V).

- **Worst Case:** $O(V + E + N \,.\, b^d + N \,.\, (\mathrm{V} + \mathrm{E}) \log_{\square} V)$

  In the worst-case scenario, the algorithm must exhaustively explore all nodes and edges for BFS and Dijkstra's components. For BFS, this means exploring every possible path level-by-level until all are covered, leading to O(V + E + N * b^d). Dijkstra's algorithm, run N times, must consider dense graph connections and multiple updates across vertices and edges, contributing O(N * (V + E) log V) to the complexity. The worst-case scenario thus combines these to yield O(V + E + N * b^d + N * (V + E) log V).

Flowchart:


Create Graph
Add Edges
Initialize Start Nodes [A, B, C]
BFS Pathfinding
Start BFS Timer
Get All Paths using BFS
Stop BFS Timer
Print All Paths and BFS Execution Time
Dijkstra Pathfinding
Start Dijkstra Timer
Get Shortest Path using Dijkstra
Stop Dijkstra Timer
Print Shortest Path and Dijkstra Execution Time
For each Start Node (A, B, C)