FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

DEPARTMENT OF COMMUNICATION TECHNOLOGY AND NETWORK

SEMESTER 6 2024/2025

COURSE:	BACHELOR OF COMPUTER SCIENCE (COMPUTER NETWORK) WITH HONORS
COURSE CODE:	CNS4202: DESIGN AND ANALYSIS OF ALGORITHM
LECTURER:	DR. NUR ARZILAWATI BINTI MD YUNUS
TITLE:	TRAFFIC OPTIMIZATION FROM UPM TO BESUT DURING FESTIVE SEASON

NAME	**MATRIC NUMBER**
SITI NAJMA IZZATY BINTI MUHD ASYRAF	216922
AYA SOFEA BINTI ROSLI	214973
FATIN NASUHA BINTI ZURAIDI	215190
AZYAN SYAZWANI BINTI SETIA	215014

TABLE OF CONTENT

1.0 Introduction

Malaysia experiences a significant surge in traffic during festive seasons such as Hari Raya Aidilfitri, Chinese New Year, and Deepavali, especially along major interstate highways. One of the most heavily affected routes is the journey from Universiti Putra Malaysia (UPM) to Besut, Terengganu, where thousands of vehicles converge on Lebuhraya Pantai Timur (LPT) and surrounding trunk roads. Long travel times, unexpected congestion, and limited alternative routes make this a pressing issue for road users.

This project was inspired by a real concern raised by one of our team members, who is originally from Besut and often travels back during festive periods. Their firsthand experience with severe congestion, particularly around toll plazas and narrow rural connectors, sparked the idea to investigate how algorithmic pathfinding can help identify more optimal travel routes.

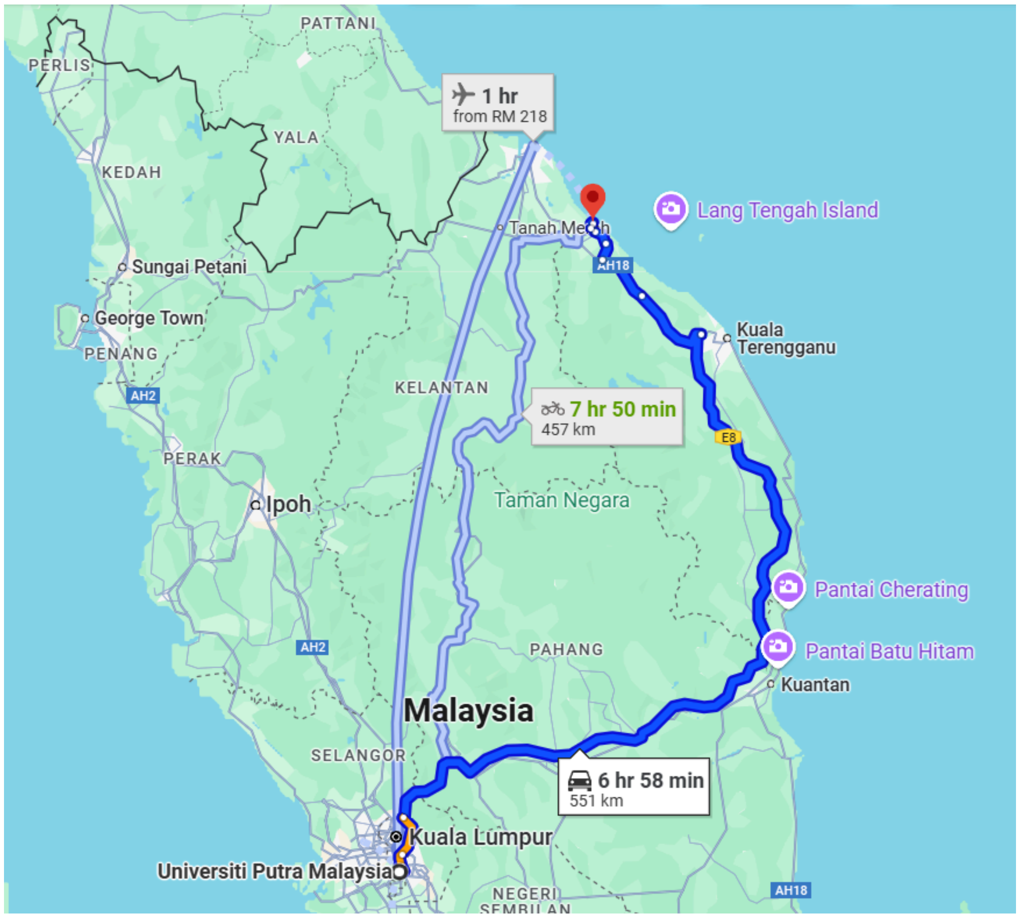


Figure 1.1 *illustrates one of the routes from UPM to Besut.*

2.0 Objectives

The main objective of this project is to analyze and compare two graph-based pathfinding algorithms which are Dijkstra’s Algorithm and A* (A-Star) Algorithm to determine which provides more efficient and intelligent routing from UPM to Besut during periods of high traffic density.

Specific objectives include:

- To simulate real-world traffic scenarios using weighted graphs representing highways and junctions.
- To implement both Dijkstra and A* algorithms and compare their efficiency in terms of travel cost, time, and adaptability.
- To identify the more effective algorithm for real-time traffic-aware routing, potentially applicable to navigation systems.
- To model traffic conditions typically seen during festive seasons, such as delays at toll plazas and congested sections of the LPT.

3.0 Problem Definition

3.1 Scenario Overview

Festive seasons such as Hari Raya Aidilfitri and Chinese New Year see a massive increase in the number of travelers using Lebuhraya Pantai Timur to return to their hometowns, resulting in extreme traffic congestion. As thousands of vehicles flood the highway at the same time, common choke points including Gombak Toll, Karak Highway, and Bentong Exit become heavily congested, slowing traffic to a crawl.

Peak traffic hours during festive periods typically occur a day before the celebration and on the final day of the holiday when people return to urban areas. Additionally, heavy congestion builds up at toll plazas where high transaction volumes cause delays, exacerbating the situation further. The lack of alternative routes and insufficient real-time traffic control measures make it challenging for motorists to navigate efficiently, leading to frustrating travel experiences.

3.2 Impact on Travelers

Prolonged traffic jams severely impact travelers, turning a typical three-hour trip into an exhausting ten-hour journey. The excessive delays disrupt travel plans and make long-distance driving physically and mentally draining for motorists.

Beyond personal inconvenience, congestion also leads to significant economic and environmental consequences. Increased fuel consumption due to idling in traffic results in higher travel costs, making festive travel financially burdensome. Additionally, higher carbon emissions from slow-moving or stationary vehicles contribute to environmental pollution.

Safety risks also escalate during heavy traffic congestion. Driver fatigue from prolonged hours on the road increases the likelihood of accidents, while aggressive driving behavior such as reckless overtaking and sudden braking becomes more common as motorists try to maneuver through slow-moving traffic. Without an optimized route suggestion system, drivers are left with limited options, forcing them to endure unpredictably long delays.

3.3 Expected Outcome

To address these challenges, this project aims to develop a Java-based program that provides travelers with optimized route suggestions based on real-time and historical congestion data.

The system will analyze live traffic updates to detect congestion hotspots, recommend alternative routes to help travelers reach their destinations faster, reduce fuel consumption and environmental impact by minimizing unnecessary delays, and enhance road safety by providing accurate travel time predictions and reducing driver fatigue.

With this solution, motorists traveling on Lebuhraya Pantai Timur during festive seasons will experience shorter travel times, lower stress, and improved traffic flow, ensuring a more efficient and smoother journey.

4.0 Model of the Problem

4.1 Approach

A model simulation of the road network from UPM to Besut created as a directed weighted graph in order to address the traffic optimization issue. Each node in the graph represents locations such as UPM, Kuala Lumpur (KL), Karak, Kuantan, and each edge represents travel time or cost of the journey.

4.2 Constraints

The constraints used:

- Travel is allowed only on defined road segments
- Edge weights must be positive (non-negative travel time/cost)
- Congestion factors (1.0 – 2.0) vary depending on festive volume
- Must start at UPM and end at Besut

4.3 Simulation Dataset

The simulation dataset represents a real-world scenario of the road network from Universiti Putra Malaysia (UPM) to Besut, Terengganu, which is commonly traveled during festive seasons. The dataset includes the main highway route and several alternative routes that drivers might take to avoid traffic congestion. Each route segment represent by the following attributes:

- **From / To:** The starting and destination points of each segment.
- **Distance (km) :** Total kilometers between the two nodes
- **Toll (RM)** : Toll cost required to use that route
- **Base Time (minutes)**: Estimated travel time under normal traffic conditions
- **Fuel Cost (RM)**: Calculated based on distance and fuel price (RM per km)
- **Total Cost (RM)**: Combined cost of toll and fuel, used for cost-based optimization

From	To	Distance (km)	Toll (RM)	Base Time (30 min)	Fuel Cost (RM)	Total Cost (RM)
UPM	KL	25	5.00	30	5.13	10.13
KL	Karak	60	15.00	45	12.30	27.30
Karak	Kuantan	150	25.00	120	30.70	55.75
Kuantan	Kuala Terengganu	200	35.00	150	41.00	76.00
Kuala Terengganu	Besut	60	10.00	50	8.20	16.20
KL	Alternative 1	40	8.00	140	34.85	54.85
Alternative 1	Kuantan	170	20.00	140	34.85	54.85
Karak	Alternative 2	80	12.00	70	16.40	28.40
Alternative 2	Kuala Terengganu	220	30.00	180	45.10	75.10
Kuantan	Coastal Route	180	25.00	160	36.90	61.90
Coastal Route	Besut	70	15.00	60	14.35	29.35

***Table 4.3.1** shows the dataset represents the road segments (edges) used to simulate traffic from UPM to Besut, including both the main route and alternative routes. Each row describes a segment between two locations with real-world-like travel data.*

This simulation dataset to provide the pathfinding for Dijkstra and A*. Each road segment becomes a weighted edge in a directed graph. The dataset also includes alternative routes with different cost and time values, simulating driver choices during traffic jams.

5.0 Algorithm Specification

5.1 Chosen Algorithm Approach

In addressing the issue of traffic congestion between Universiti Putra Malaysia (UPM) and Besut during festive seasons, the road network is effectively modeled as a graph. In this model, nodes represent toll plazas, junctions, and rest areas, while edges represent road segments with associated travel costs such as distance, travel time, or toll charges. This structural representation allows for accurate modeling of the highway layout and is particularly suitable for algorithmic pathfinding.

Given this context, a graph-based algorithmic approach is considered the most appropriate for solving the traffic optimization problem. Graph-based algorithms are commonly used in routing and navigation systems because they can compute optimal paths under various constraints. They support optimization objectives such as minimizing distance, reducing travel time, and avoiding congested routes, which are especially important for travel during peak festive periods.

For this project, Dijkstra's Algorithm and A* (A-Star) Algorithm have been selected for implementation and comparison. Dijkstra's Algorithm provides a reliable foundation for calculating the shortest path under static conditions, while the A* Algorithm builds on this by incorporating a heuristic function that enables more efficient and traffic-aware route selection.

5.2 Comparison of Algorithm Paradigms

Aspect	Dijkstra's Algorithm	A* Algorithm
Traffic Awareness	Not aware of congestion from UPM-Besut	Considers real-time festive congestion
Search Direction	Explores all possible routes	Focused toward destination
Speed	Slower	Faster
Real-time use	Limited	Suitable for current highway conditions
Congestion Handling	For static path simulation	For live traffic-based route optimization
Project Use	Cannot adapt to traffic spikes	Avoids congested routes

6.0 Algorithm Design

This section outlines the design and logic behind the algorithms selected to address traffic congestion along the route from Universiti Putra Malaysia (UPM) to Besut during festive seasons, where traffic congestion on highways such as Lebuhraya Pantai Timur (LPT) and alternate routes is common. The problem is approached using two graph-based algorithms: Dijkstra’s Algorithm and A* (A-Star) Algorithm. Both algorithms are well-established for shortest path problems but differ in how they handle optimization and dynamic road conditions.

6.1 Dijkstra’s Algorithm Overview

Dijkstra’s Algorithm is a classical pathfinding technique designed to compute the shortest path from a source node to all other nodes in a weighted graph with non-negative edge weights. In this project, Dijkstra’s Algorithm is used to model the UPM–Besut route as a graph, where:

- Nodes (vertices) represent critical traffic points such as toll plazas, highway junctions, and rest stops along the LPT or alternative roads.

- Edges represent road segments connecting these points, and weights are based on static travel costs such as distance or historical average travel time.

While Dijkstra’s Algorithm is effective for basic routing, it does not take into account real-time changes such as traffic congestion or accidents. As such, it serves as a baseline method for comparison in this study.

6.1.1 Step by Step Explanation of Dijkstra's Algorithm

1. Set all node distances to ∞, except the start node (UPM) which is set to 0.
2. Mark all nodes as unvisited.
3. Add the start node to a priority queue with a priority of 0.
4. While the queue is not empty:
   a. Remove the node u with the smallest tentative distance.
   b. For each unvisited neighbor v of u:
      i. Calculate alternative distance: alt = distance[u] + weight(u, v)
      ii. If alt is less than current distance[v], update distance[v] = alt
      iii. Add or update v in the priority queue with the new distance
5. Repeat until the destination node (Besut) is reached or all nodes are visited.
6. Reconstruct the shortest path by backtracking from Besut to UPM using a previous-node tracker.

6.2 A* Algorithm Overview

The A* Algorithm improves upon Dijkstra's by incorporating a heuristic function to estimate the cost from the current node to the destination. It uses:

- g(n): the known cost from the start (UPM) to the current node
- h(n): the estimated cost from the current node to the destination (Besut)

In this project, A* is highly effective for routing between UPM and Besut because it can integrate real-time traffic data, including:

- Congestion along LPT
- Accidents or roadblocks
- Predictive travel delays based on time of day

This allows A* to identify faster routes, not just the shortest ones.

6.2.1 Step by Step Explanation of A* Algorithm

1. Initialize `g(n)` for all nodes to ∞; set `g(UPM) = 0`
2. Calculate initial `f(n) = g(n) + h(n)`; for UPM, `f(UPM) = h(UPM)`
3. Insert UPM into a priority queue with `f(UPM)` as priority
4. While the queue is not empty:
   a. Remove node u with the lowest `f(u)`
   b. If u is Besut, reconstruct the path and terminate
   c. For each neighbor v of u:
      i. Calculate `tentative_g = g(u) + cost(u, v)`
      ii. If `tentative_g < g(v)`:
         - Update `g(v) = tentative_g`
         - Calculate `f(v) = g(v) + h(v)`
         - Insert or update v in the priority queue with `f(v)`
5. Repeat until Besut is reached or no more nodes are available.

The heuristic function `h(n)` may include:

- Straight-line (Euclidean) distance to Besut
- Estimated time based on current speed limits
- Real-time congestion or delay data

6.3 Pseudocode for Full Code

6.3.1 Main Function

```
START
1. Create a RoadNetwork instance
2. Add all locations (e.g., UPM, KL, Karak, etc.)
3. Add roads between locations with distance, toll cost, and base
time
4. Simulate festive season by increasing congestion on major roads
5. Generate heuristic estimates for A* algorithm

6. Print “Time-Optimized Path”
   - Call dijkstra with prioritizeTime = true
   - Call A* with prioritizeTime = true
   - Print comparison of both results

7. Print “Cost-Optimized Path”
   - Call dijkstra with prioritizeCost = true
   - Call A* with prioritizeCost = true
   - Print comparison of both results

8. Print “Balanced Path”
   - Call both algorithms with both priorities set to true
   - Print comparison
END
```

6.3.2 RoadNetwork Class

```
CLASS RoadNetwork
    MAP adjacencyList (String -> List of Road)

    FUNCTION addLocation(location)
        IF location not in adjacencyList
            add empty list for location

    FUNCTION addRoad(source, destination, distance, toll, baseTime)
        add Road to both source and destination in adjacencyList

    FUNCTION updateTrafficConditions(source, destination,
congestionFactor)
        FOR each road in adjacencyList[source]
            IF road.destination == destination
                update road.congestionFactor
```

6.3.3 Road Class

```
CLASS Road
    destination, distance, tollCost, baseTime, congestionFactor = 1.0

    FUNCTION getCurrentTime()
        RETURN baseTime * congestionFactor

    FUNCTION getFuelCost()
        RETURN (distance / 10) * 2.05

    FUNCTION getTotalCost()
        RETURN tollCost + getFuelCost()
```

6.3.4 Dijkstra's Algorithm

```
FUNCTION    dijkstra(network,    start,    end,    prioritizeTime,
prioritizeCost)
    Initialize distance, time, cost maps to infinity
    Set distance, time, cost for start = 0
    Add start node to priority queue with cost = 0

    WHILE priority queue not empty
        current = dequeue

        IF current == end THEN break

        FOR each neighbor road of current.location
            Calculate newTime = current.time + road.getCurrentTime()
                        Calculate  newCost  =  current.monetaryCost  +
road.getTotalCost()

            IF prioritize both
                combinedCost = (newTime + newCost) / 2
            ELSE IF prioritizeTime
                combinedCost = newTime
            ELSE
                combinedCost = newCost

            IF combinedCost < dist[neighbor]
                Update dist, time, cost, and prev maps
                Add neighbor to priority queue

    RETURN buildPath(prev, network, start, end, totalTime, totalCost)
```

6.3.5 A* Algorithm

```
FUNCTION aStar(network, start, end, prioritizeTime, prioritizeCost, heuristicMap)
    Initialize gScore, time, cost maps to infinity
    Set gScore, time, cost for start = 0
    Add start node to priority queue

    WHILE priority queue not empty
        current = dequeue

        IF current == end THEN break

        FOR each neighbor road of current.location
            newTime = current.time + road.getCurrentTime()
            newCost = current.monetaryCost + road.getTotalCost()

            IF prioritize both
                combined = (road.getCurrentTime() + road.getTotalCost()) / 2
            ELSE IF prioritizeTime
                combined = road.getCurrentTime()
            ELSE
                combined = road.getTotalCost()

            tentativeG = gScore[current] + combined

            IF tentativeG < gScore[neighbor]
                Update gScore, time, cost, cameFrom maps
                Add neighbor to priority queue with (tentativeG + heuristic[neighbor])

    RETURN buildPath(cameFrom, network, start, end, totalTime, totalCost)
```

6.3.6 buildPath Function

```
FUNCTION buildPath(pathMap, network, start, end, totalTime,
totalCost)
    Initialize path as empty list
    totalDistance = 0
    current = end

    WHILE current != null
        Insert current to front of path
        prev = pathMap[current]
        IF prev exists
            Find matching road from prev to current
            Add road.distance to totalDistance
        current = prev

    RETURN PathResult(path, totalTime, totalCost, totalDistance)
```

6.3.7 simulateFestiveTraffic Function

```
FUNCTION simulateFestiveTraffic(network)
    Define list of main congested roads
    FOR each road pair in list
        Generate random congestion factor between 1.5 and 3.5
        Update traffic conditions in network
```

6.3.8 createHeuristic Function

```
FUNCTION createHeuristic()
    RETURN predefined map with estimated remaining distances from
each node to destination (Besut)
```

7.0 Algorithm Correctness & Complexity Analysis

7.1 Proof of Algorithm Correctness

This project focuses on creating a smart traffic optimization system to help drivers identify the fastest route and minimize cost from Universiti Putra Malaysia (UPM) to Besut, Terengganu, particularly during peak holiday seasons when heavy congestion occurs on major highways and roads in Selangor, Pahang, and Terengganu. The study compares two classic shortest path algorithms, which are Dijkstra's Algorithm and the A* (A-star) Algorithm by comparing their accuracy and efficiency to determine which is better suited for this purpose.

7.1.1 Dijkstra's Algorithm

Dijkstra's algorithm is designed to find the shortest path from a single source node to all other nodes in a graph with non-negative edge weights. In the context of this project, each edge (road segment) is assigned a combined weight:

Edge Cost = Estimated Travel Time + Toll Cost + Estimated Fuel Cost

Since all components of the edge cost are non-negative, the algorithm satisfies the necessary condition for correctness.

Proof of Correctness:

- Greedy Approach: At each iteration, the algorithm chooses the node with the minimum total cost (g(n)), ensuring locally optimal decisions.
- Loop Invariant: Once a node's shortest distance is finalized, it remains unchanged throughout.
- Optimal Substructure: The optimal path to a node is composed of optimal paths to its predecessors.

Thus, Dijkstra's algorithm will always yield the correct, globally optimal route in this context, assuming consistent and non-negative cost metrics.

7.1.2 A* Algorithm

The A* algorithm is a robust pathfinding technique that combines heuristic-driven efficiency with Dijkstra's guarantee of optimality. By giving priority to the shortest and least congested routes, A* can dynamically adjust routes from UPM to Besut. The mechanics of A*, its applicability to holiday traffic control and optimization traffic during peak holiday.

The A* Algorithm builds upon Dijkstra's by introducing a heuristic function h(n) that estimates the remaining cost from a node n to the goal. The total evaluation function is:

$$f(n) \;=\; g(n) \;+\; h(n)$$

Where:

- g(n) is the cumulative cost from UPM to node n
- h(n) is the estimated cost from n to Besut

In this project, g(n) includes the actual cost:

$$g(n) \;=\; Travel\,Time \;+\; Toll\,Cost \;+\; Fuel\,Cost$$

While h(n) is derived from:

$$h(n) \;=\; Estimated\,remaining\,time \;+\; Estimated\,toll\,and\,fuel\,costs$$

Proof of correctness:

A* is guaranteed to find the optimal path if:

1. **Admissibility**: The heuristic never overestimates the actual remaining cost.

2. **Consistency**: The heuristic satisfies:
   $h(n) \le cost(n,\, n') \;+\; h(n')$

In this case, a straight-line distance to Besut, combined with estimated cost per kilometer, is admissible and consistent, ensuring A* remains correct.

7.2 Recurrence Relation Analysis

In this project, the road network is modeled as a weighted graph, where each node represents a junction, toll plaza, or rest stop, and each edge represents a road segment with an associated travel cost. The pathfinding process, whether using Dijkstra's Algorithm or A*, proceeds incrementally, evaluating the cumulative cost to reach each new node from the source. This cumulative cost can be described using a recurrence relation, which models how the total travel cost builds as the algorithm traverses from node to node toward the destination.

Let:

- $T(n)$ represent the total cost to reach the nth node on the path from the source (UPM).
- $T(n-1)$ be the total cost to reach the previous node.
- $cost(n)$ be the cost of the current road segment (edge) between node $n-1$ and node n

Then the recurrence relation is defined as:

$$T(n) = T(n-1) + cost(n)$$

Where:

$$cost(n) = travel\,timen + toll\,costn + fuel\,costn$$

Each component of cost(n) is non-negative and may vary dynamically based on real-time data such as congestion, toll rate changes, or distance-based fuel estimations.

In A*:

The recurrence still holds, but the total cost f(n) used to prioritize the path selection includes both the known accumulated cost g(n) and an estimated cost h(n) to the goal (Besut):

$f(n) = g(n) + h(n)$

Where:

- $g(n) = T(n) = T(n-1) + cost(n)$
- $h(n)$ is the heuristic estimate from node n to Besut (e.g., based on remaining distance × average cost/km)

This formulation ensures that A* selects the next node based on the total expected cost to reach the destination, balancing actual progress ($g(n)$) and estimated remaining cost ($h(n)$).

7.3 Algorithm Selection Justification

The A* Algorithm is selected as the core routing algorithm for this project due to its strong suitability for solving single-source to single-destination pathfinding problems, which aligns perfectly with the goal of optimizing travel from Universiti Putra Malaysia (UPM) to Besut, Terengganu. The A* Algorithm is known for its goal-directed search strategy, where the pathfinding process is guided by a heuristic function that estimates the cost from the current node to the destination. This allows A* to prioritize exploration of nodes that are more likely to lead toward the goal, significantly reducing the number of unnecessary nodes explored, which is a key advantage over Dijkstra's Algorithm. In the context of this project, where the objective is to minimize not only travel time but also monetary costs such as toll fees and fuel consumption, the A* algorithm enables the integration of these real-world factors directly into its evaluation functions.

The A* Algorithm uses a combined cost function defined as f(n) = g(n) + h(n), where g(n) represents the actual cumulative cost from the source (UPM) to a given node n, and h(n) is a heuristic estimation of the remaining cost from that node to the destination (Besut). In this project, g(n) accounts for real-time travel time, toll charges, and estimated fuel cost for each road segment, while h(n) can be implemented using straight-line (Haversine) distance to Besut multiplied by average cost-per-kilometer, which ensures the heuristic remains admissible and consistent. This integration allows A* to not only find the fastest path but also the most economical one. Additionally, the A* Algorithm is highly adaptive to dynamic environments, making it suitable for festive seasons when traffic patterns are unpredictable. When real-time traffic data changes, such as sudden congestion, accidents, or lane closures, A* can recompute the optimal route efficiently without recalculating paths to all destinations.

On the other hand, Dijkstra's Algorithm, although well-known for its correctness and simplicity, is not the optimal choice for this scenario. Dijkstra's Algorithm does not use a heuristic, meaning it explores all reachable nodes from the source regardless of their proximity to the destination. This results in unnecessary computations, especially in a large-scale road network such as the Peninsular Malaysia highway system. While this exhaustive search approach is effective for multi-destination routing problems, it is inefficient for a task with a clearly defined destination, like UPM to Besut. Furthermore, Dijkstra’s Algorithm is less responsive to real-time traffic changes. Since it processes all nodes equally, any change in road conditions requires a complete recalculation, which can delay optimal routing decisions, especially during high-traffic periods.

In summary, the A* Algorithm offers a more intelligent, efficient, and goal-oriented approach that directly supports the project’s aim of minimizing total travel time and cost. Its ability to incorporate real-time traffic data, toll rates, and fuel consumption into the pathfinding process makes it the most practical and scalable solution for this single-destination traffic optimization system. While Dijkstra’s Algorithm remains a valid alternative in terms of correctness, its lack of heuristic guidance and full-graph exploration make it less effective for real-time, long-distance routing tasks. Therefore, A* is chosen over Dijkstra’s Algorithm as the best algorithm to meet the objectives of this project.

7.4 Complexity and Performance Analysis

This section evaluates the performance of Dijkstra’s Algorithm and A* Algorithm in the context of optimizing traffic flow from Universiti Putra Malaysia (UPM) to Besut. The performance is compared based on three main criteria, which are time complexity, space complexity, and scalability. For the purposes of this analysis, let V represent the number of nodes in the graph (such as tolls, junctions, and rest stops), and E represent the number of edges (road segments connecting those points).

7.4.1 Time Complexity

Both Dijkstra’s and A* Algorithms share the same theoretical time complexity of

$$O((V + E) log\, V)$$

when implemented using a binary min-heap priority queue. However, there is a significant difference in practical performance. Dijkstra’s Algorithm performs a uniform search and explores all reachable nodes from the source regardless of whether they lead toward the destination. This exhaustive nature makes Dijkstra slower in practice, particularly in large networks, such as the one spanning across Selangor, Pahang, and Terengganu. In contrast, A* incorporates a heuristic that guides the search directly toward the destination (Besut), effectively pruning irrelevant paths and reducing the number of explored nodes. This makes A* faster in practice, especially for long-distance routing problems such as UPM to Besut. The effectiveness of A* in reducing computation time depends on the quality and consistency of the heuristic, which, in this project, is based on straight-line distance and estimated remaining cost, a heuristic that remains admissible and effective.

7.4.2 Space Complexity

In terms of memory usage, both algorithms have a space complexity of

$$O(V)$$

as they must store various data structures such as the distance map (shortest known distances to nodes), a predecessor map (to reconstruct the shortest path), and the priority queue. The A* Algorithm, however, includes an additional memory overhead for handling the heuristic values associated with each node. These values `f(n)`, `g(n)`, and `h(n)`, help A* prioritize its search and must be stored for efficient access. In this project, although A* uses more memory, the extra space required for heuristic evaluation is minimal and does not pose a limitation due to the manageable size of the highway graph and the availability of modern computing resources. Therefore, the difference in space usage is not significant enough to outweigh A*'s benefits in performance and responsiveness.

7.4.3 Scalability

Scalability evaluates how efficiently each algorithm performs as the size of the road network increases and as real-time traffic changes become more frequent. Dijkstra’s Algorithm scales reasonably well for all-pairs shortest path problems and is often used in applications requiring multi-destination routing. However, it is not efficient for single-source to single-target problems in large graphs, such as the UPM–Besut route. Dijkstra’s explores all reachable nodes, regardless of their relevance to the target, which can significantly increase processing time in nationwide highway networks. For example, when calculating the optimal route to Besut, Dijkstra may unnecessarily explore routes toward unrelated regions like Johor or Perak, thereby consuming additional time and memory. This exhaustive search strategy reduces its effectiveness when the destination is fixed and known in advance.

In contrast, the A* Algorithm is specifically designed for single-source to single-destination scenarios, making it a better fit for this project. A*'s use of a heuristic allows it to focus its search toward the target, skipping exploration of unrelated or far-off paths. This makes A* significantly more scalable in large graphs, such as Malaysia’s road network. When a suitable heuristic is applied, such as the straight-line distance to Besut, A* becomes not only efficient but also reliable. The chosen heuristic in this project is both admissible (it never overestimates the actual cost) and consistent (it satisfies the triangle inequality), ensuring that A* always returns the optimal path while minimizing search overhead. Additionally, A* supports real-time responsiveness, allowing it to re-evaluate and adapt to live traffic updates quickly without the need to reprocess the entire graph. Therefore, in terms of scalability, A* offers clear advantages over Dijkstra’s Algorithm, especially for targeted, dynamic, long-distance routing like the one required from UPM to Besut during congested festive periods.

8.0 Implementation

8.1 Java Code Development

Two methods, Dijkstra's Algorithm and the A* Algorithm, were created in Java to address the traffic routing issue between UPM to Besut during the peak festive season. Both are intended to determine the best routes for travel between several destinations with varying priority for time, money, or a combination of the two. Based on the specified optimization focus, the computer models traffic congestion and determines the optimal route.

The RoadNetwork class employs an adjacency list to construct the network graph in this object-oriented software. With properties like distance, toll cost, base travel time, and traffic congestion, each road is represented as an object of the Road class. While the A* technique employs a heuristic function to determine the optimum way more rapidly by predicting the remaining distance to the goal, the Dijkstra method investigates every potential path to ensure the most optimal answer.

The main function of the program initializes the map, uses randomized congestion factors to simulate festive traffic circumstances, and executes both algorithms in three different scenarios: balanced optimization, cost optimization, and time optimization. For each scenario, the report clearly displays the distance traveled, total time, total cost, and path taken. Using this configuration, comparing the effectiveness of A*'s heuristic-driven approach and Dijkstra's exhaustive method under various journey priorities is simple.

8.2 Challenges faced

One of the most significant challenges in the development phase was designing and tuning the heuristic values used by the A* algorithm. For A* algorithm to function effectively, the heuristic's quality is crucial. If it is inaccurate, it could result in less-than-ideal routes. For instance, the A* algorithm chose a slower path in the time-optimized test scenario because its heuristic misprioritized cost over time. Manual adjustment and theoretical estimate of the remaining costs and distances from each node to the objective were necessary to get the ideal heuristic balance.

The program's logic had to accurately simulate real-world traffic situations, which was one of the main development problems. Since traffic on different roads changes during festive seasons, a method was developed to replicate this using a congestion factor. The base travel time on each road segment was dynamically altered by this factor. However, careful design was needed to incorporate this into the algorithm without changing other path features such as cost and distance. Initially, the outputs were misrepresented because congestion was applied incorrectly to the wrong metric. For instance, distance or toll. To maintain the simulation's realism, it was crucial to make sure that just the baseTime was changed while maintaining constant toll and fuel prices.

Another issue was ensuring that the traffic simulation logic did not conflict with the pathfinding algorithms. Without affecting the traversal logic of the algorithm, the congestion factor had to be appropriately incorporated into the time calculation for each route. The congestion factor was frequently implemented inconsistently, updating only one direction of a bidirectional road at a time, creating uneven or illogical routes. To ensure that each road's trip time updated accurately during simulation and that all directions showed the same level of congestion, careful debugging was necessary. This required human output verification and several runs using controlled random sequences.

Last but not least, the implementation and comparison of two essentially dissimilar algorithms which is Dijkstra and A* added complexity to the output interpretation and code structure. A* requires a heuristic that directs it more effectively towards the destination, whereas Dijkstra's algorithm use a standard approach to search every node. This required delivering comparable results using the same metrics and format while preserving two distinct but consistent cost evaluation processes. Furthermore, a shared weighting method that was equitable to both algorithms was required for the balanced scenario, which incorporates both time and cost. Ensuring that the printed results were aligned for testing and analysis was particularly difficult, as was designing an extensible framework that could flexibly support all three optimization modes (time, cost, and balanced) and switch between algorithms without introducing bias or duplicating code.

8.3 Example Test Case

Test Case	Scenario	Input
Test Case 1	Time Optimization	start = UPM, end = Besut, prioritizeTime = true, prioritizeCost = false
Test Case 2	Cost Optimization	start = UPM, end = Besut, prioritizeTime = false, prioritizeCost = true
Test Case 3	Balanced Time and Cost	start = UPM, end = Besut, prioritizeTime = true, prioritizeCost = true

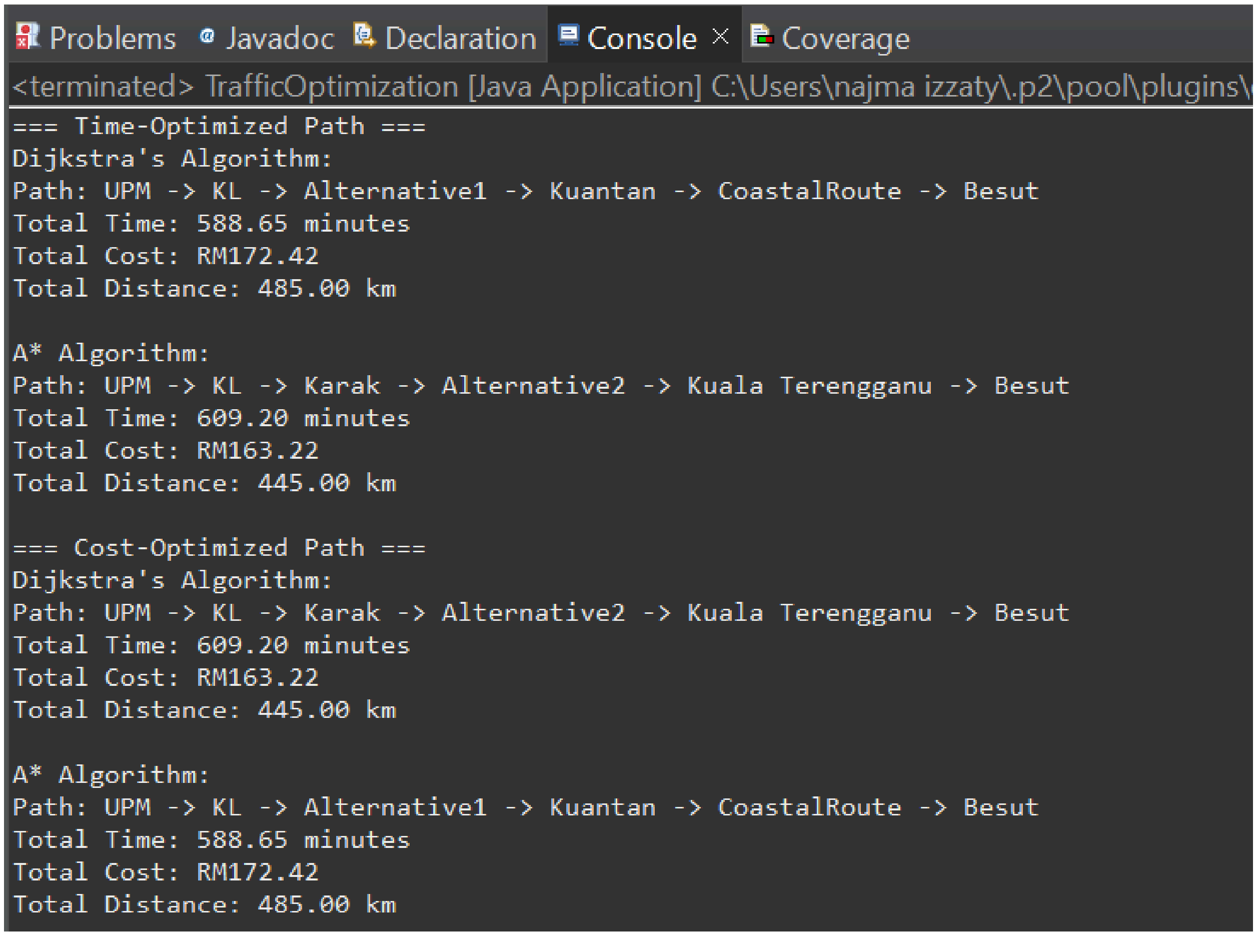
In the first test case, the goal is to evaluate how the algorithm performs when only travel time is prioritized. The inputs are start = UPM, end = Besut, with prioritizeTime = true and prioritizeCost = false. This means the algorithm, whether A* or Dijkstra's, will try to identify the fastest route.

In the second test case, the focus is on cost efficiency, where the algorithm is guided by prioritizeTime = false and prioritizeCost = true. While the primary goal is to minimize total travel cost including toll fees and fuel consumption but travel time still plays a role in the result, even though it's not the main priority. Interestingly, the output showed that the A* algorithm selected a path with a slightly higher cost but arrived sooner than Dijkstra's path.

The third test case evaluates how the algorithms behave when both time and cost are equally important. With prioritizeTime = true and prioritizeCost = true, the path chosen should reflect a balanced trade-off between reaching the destination quickly and minimizing expenses.

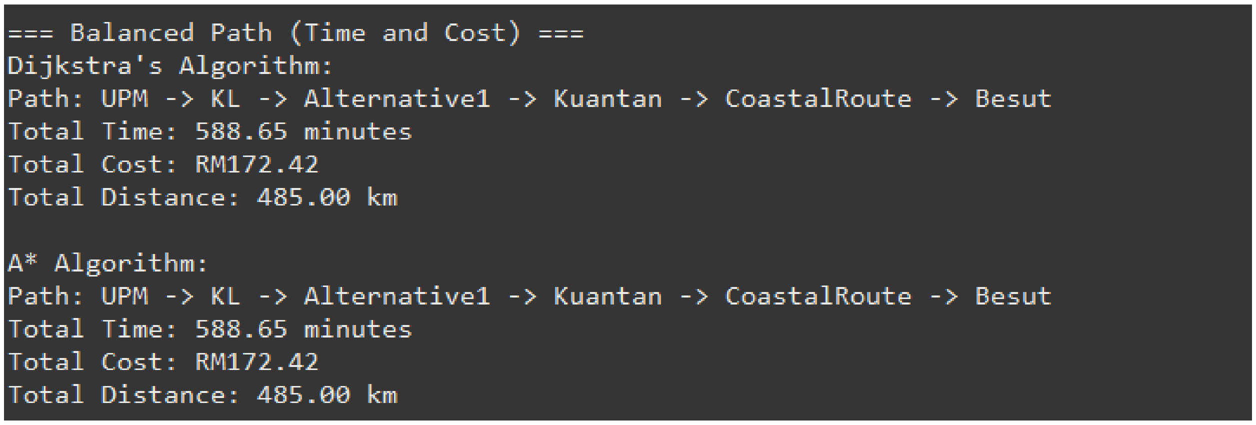
9.0 Program Testing

9.1 Results



```
<terminated> TrafficOptimization [Java Application] C:\Users\najma izzaty\.p2\pool\plugins\
=== Time-Optimized Path ===
Dijkstra's Algorithm:
Path: UPM -> KL -> Alternative1 -> Kuantan -> CoastalRoute -> Besut
Total Time: 588.65 minutes
Total Cost: RM172.42
Total Distance: 485.00 km

A* Algorithm:
Path: UPM -> KL -> Karak -> Alternative2 -> Kuala Terengganu -> Besut
Total Time: 609.20 minutes
Total Cost: RM163.22
Total Distance: 445.00 km

=== Cost-Optimized Path ===
Dijkstra's Algorithm:
Path: UPM -> KL -> Karak -> Alternative2 -> Kuala Terengganu -> Besut
Total Time: 609.20 minutes
Total Cost: RM163.22
Total Distance: 445.00 km

A* Algorithm:
Path: UPM -> KL -> Alternative1 -> Kuantan -> CoastalRoute -> Besut
Total Time: 588.65 minutes
Total Cost: RM172.42
Total Distance: 485.00 km
```

Figure 9.1.1 *Output in finding Time-Optimized Path and Cost-Optimized Path for Dijkstra and A* Algorithm*



```
=== Balanced Path (Time and Cost) ===
Dijkstra's Algorithm:
Path: UPM -> KL -> Alternative1 -> Kuantan -> CoastalRoute -> Besut
Total Time: 588.65 minutes
Total Cost: RM172.42
Total Distance: 485.00 km

A* Algorithm:
Path: UPM -> KL -> Alternative1 -> Kuantan -> CoastalRoute -> Besut
Total Time: 588.65 minutes
Total Cost: RM172.42
Total Distance: 485.00 km
```

Figure 9.1.2 *Output in finding Balanced Path (Time and Cost) for Dijkstra and A* Algorithm*

9.2 Analysis

The two shortest path algorithms used in this project, Dijkstra's Algorithm and A* Algorithm, were put into operation and tested to see how well they performed in terms of optimizing traffic over the festive period. Pathfinding and navigation systems make extensive use of both approaches. By thoroughly examining every node, Dijkstra's algorithm is renowned for its assured accuracy in determining the shortest path. A* improves on this by adding heuristics approach, meaning it attempts to “guess” the best direction which more effectively direct the search toward the objective while calculating the remaining distance or cost. In comparing the A* Algorithm and Dijkstra’s Algorithm based on output, we chose A* Algorithm as the better option.

The results demonstrated that Dijkstra's algorithm discovered a faster path in the time-optimized situation, despite the fact that A* is typically predicted to perform better by employing heuristics to estimate the shortest route. Dijkstra's algorithm found a shorter trip time of 588.65 minutes via KL → Alternative1 → Kuantan → CoastalRoute, whereas A* selected the route via Karak → Alternative2 → Kuala Terengganu, which resulted in a total duration of 609.20 minutes. A* algorithm relied on a heuristic to estimate remaining travel time, but the heuristic did not accurately reflect the real-time traffic conditions or actual time required. Dijkstra was able to identify the most time-efficient route without being influenced by heuristic assumptions by thoroughly analyzing every potential path based only on actual trip time.

In the cost-optimized path, A* performed better in terms of overall efficiency by finding a faster route that completed the journey in 588.65 minutes, even though it cost slightly more at RM172.42. This implies that in circumstances when speed and cost are crucial factors, the time saved by employing A* can exceed the additional costs. In contrast, Dijkstra's algorithm only considered cost reduction, choosing a less expensive route that cost RM163.22, but it took longer which is 609.20 minutes. Due to its thorough search, Dijkstra ensures the lowest cost, yet, A* Algorithm showed a more appropriate balance by providing a faster arrival at a slightly higher cost. This highlights A*'s strength when moderate cost increases are acceptable for significantly better travel times.

In the balanced path scenario, where both time and cost were given equal importance, A* algorithm demonstrated its advantage by efficiently identifying the optimal route: UPM → KL → Alternative1 → Kuantan → CoastalRoute → Besut. The end results from A* and Dijkstra were the same which is the travel time took 588.65 minutes, the cost were RM172.42, and the total distance were 485.00 km, but A* was able to do this by using a more focused and directed search. It avoided unnecessary paths and minimized processing work due to its heuristic, which included time and cost aspects. Dijkstra also discovered the same route, but he did it by thoroughly examining every option, which is less effective. This result demonstrates that in balanced situations, particularly when time and computational resources are limited, A* may be the preferable option.

To sum up, the testing demonstrates the unique advantages of both Dijkstra's and A* algorithms. By utilizing f(n) = g(n) to exhaustively explore every option that may be taken, Dijkstra's algorithm prioritizes correctness and guarantees that it always discovers the best path for the chosen metric, whether it cost, or time. In view of this, it is extremely dependable in situations where guaranteed optimality is crucial. However, in order to intelligently direct the search and avoid unnecessary paths, the A* method employs f(n) = g(n) + h(n) and integrates heuristics to estimate the remaining distance, time, or cost. Even while A*'s performance is highly dependent on the heuristic's quality, it is faster and more effective, particularly in big or complicated networks. In this instance, A* demonstrated to be more adaptable and effective in the balanced scenario, achieving the same result with less investigation, even if Dijkstra performed better in isolated time and cost scenarios. Therefore, if its heuristic is well-designed, A* is frequently the preferable option in real-world applications where speed and scalability are important considerations.

10.0 Conclusion

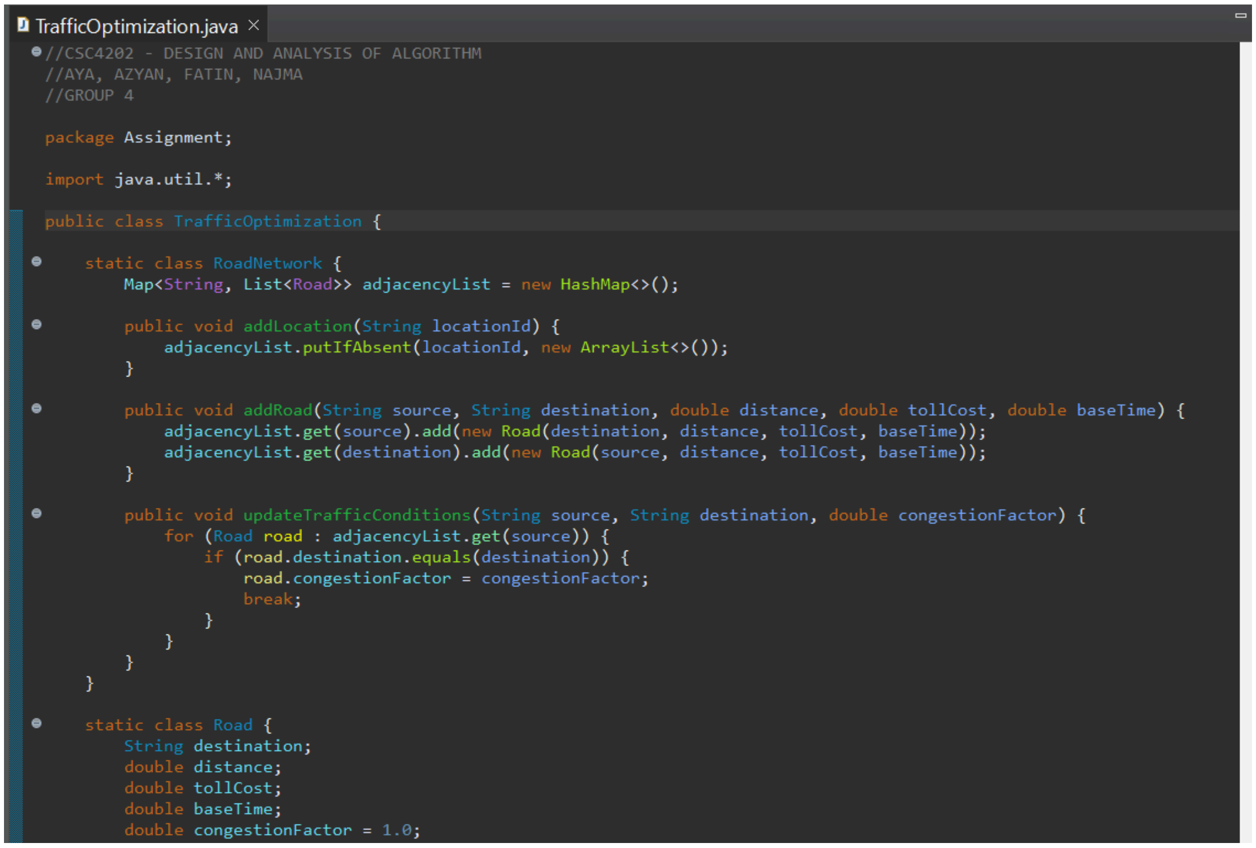
In conclusion, the project "Traffic Optimization from UPM to Besut During Festive Season" effectively illustrated the application of shortest path algorithms to practical traffic routing issues. The study assessed the efficiency of Dijkstra's Algorithm and the A* Algorithm in optimizing routes based on various travel priorities such as time, cost, and a balance of both by mimicking the traffic congestion typically encountered during festive occasions. A realistic testing environment for evaluating route choices was offered via a structured road network with several routes and simulated congestion conditions.

The results of the tests showed that Dijkstra's Algorithm worked best when cost or time was the primary consideration. Due to its thoroughness, it was able to investigate every potential path and choose the best one using the selected metric. However, both algorithms generated the same path in the balanced situation, when time and cost were equally taken into account. A* did so more quickly by using a heuristic that directed the search towards more promising paths and eliminated unnecessary calculations.

Overall, this project demonstrated both algorithms' advantages and disadvantages. Although Dijkstra's Algorithm ensures the most accurate answer, it uses more computing power, whereas A* Algorithm provides increased efficiency when paired with a thoughtfully crafted heuristic. As long as its heuristic appropriately captures travel conditions, A* is clearly the better option for real-time traffic applications during festive seasons, when speed and responsiveness are crucial. This comparison offers important information on how to choose the best algorithm depending on particular routing objectives.

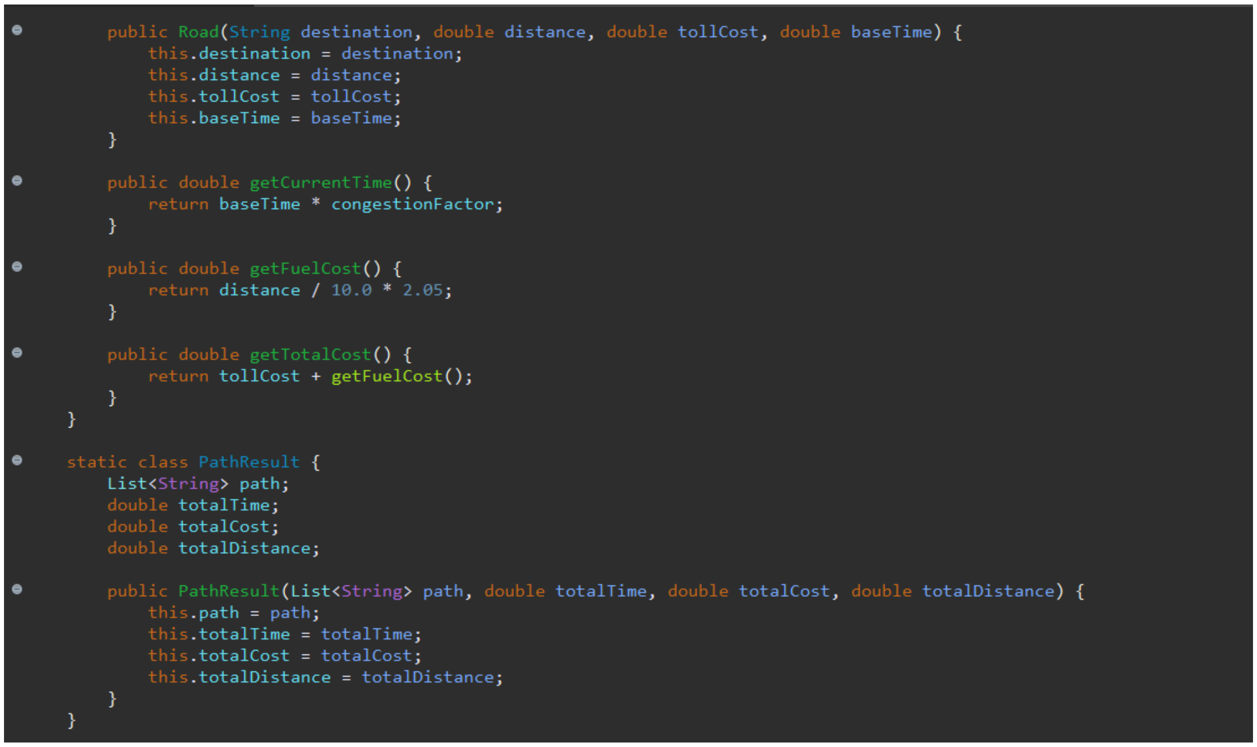
APPENDIX

Code

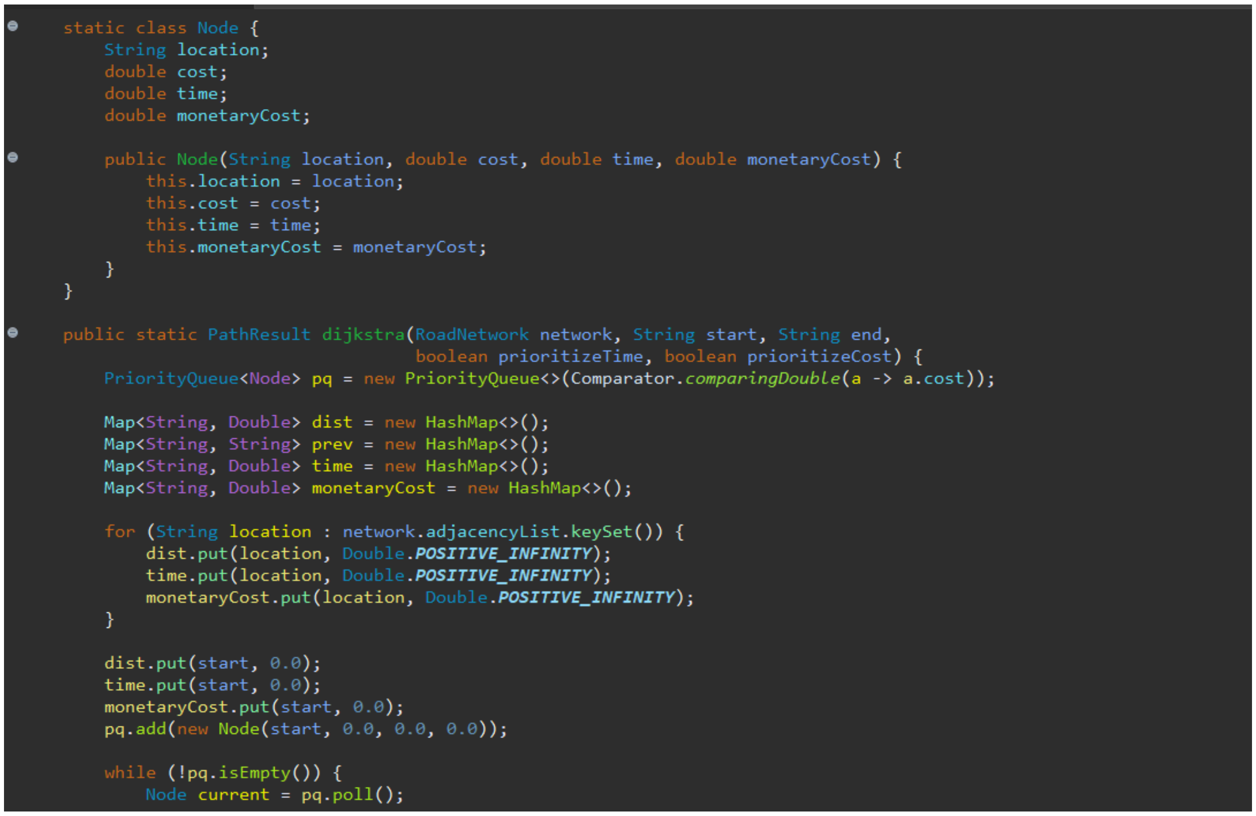


TrafficOptimization.java

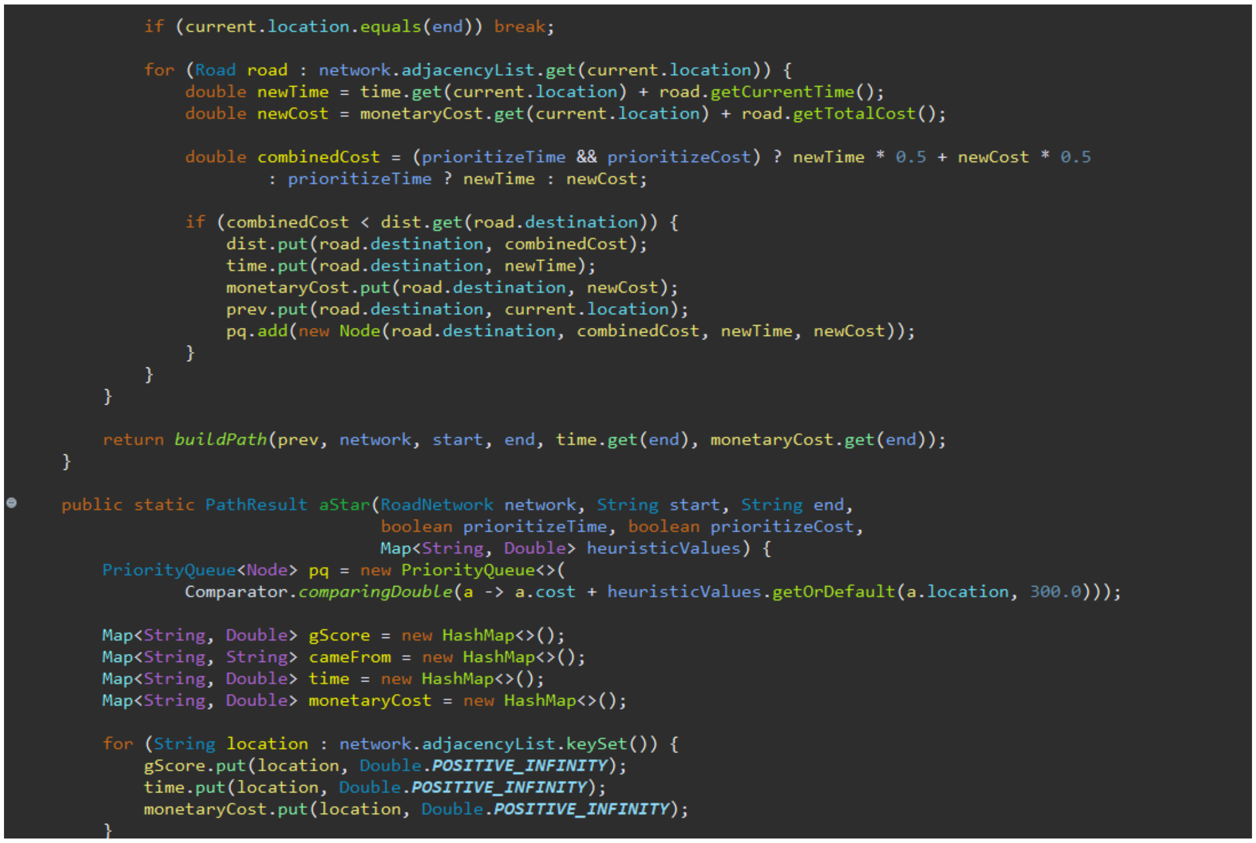
```java
//CSC4202 - DESIGN AND ANALYSIS OF ALGORITHM
//AYA, AZYAN, FATIN, NAJMA
//GROUP 4

package Assignment;

import java.util.*;

public class TrafficOptimization {

    static class RoadNetwork {
        Map<String, List<Road>> adjacencyList = new HashMap<>();

        public void addLocation(String locationId) {
            adjacencyList.putIfAbsent(locationId, new ArrayList<>());
        }

        public void addRoad(String source, String destination, double distance, double tollCost, double baseTime) {
            adjacencyList.get(source).add(new Road(destination, distance, tollCost, baseTime));
            adjacencyList.get(destination).add(new Road(source, distance, tollCost, baseTime));
        }

        public void updateTrafficConditions(String source, String destination, double congestionFactor) {
            for (Road road : adjacencyList.get(source)) {
                if (road.destination.equals(destination)) {
                    road.congestionFactor = congestionFactor;
                    break;
                }
            }
        }
    }

    static class Road {
        String destination;
        double distance;
        double tollCost;
        double baseTime;
        double congestionFactor = 1.0;
```



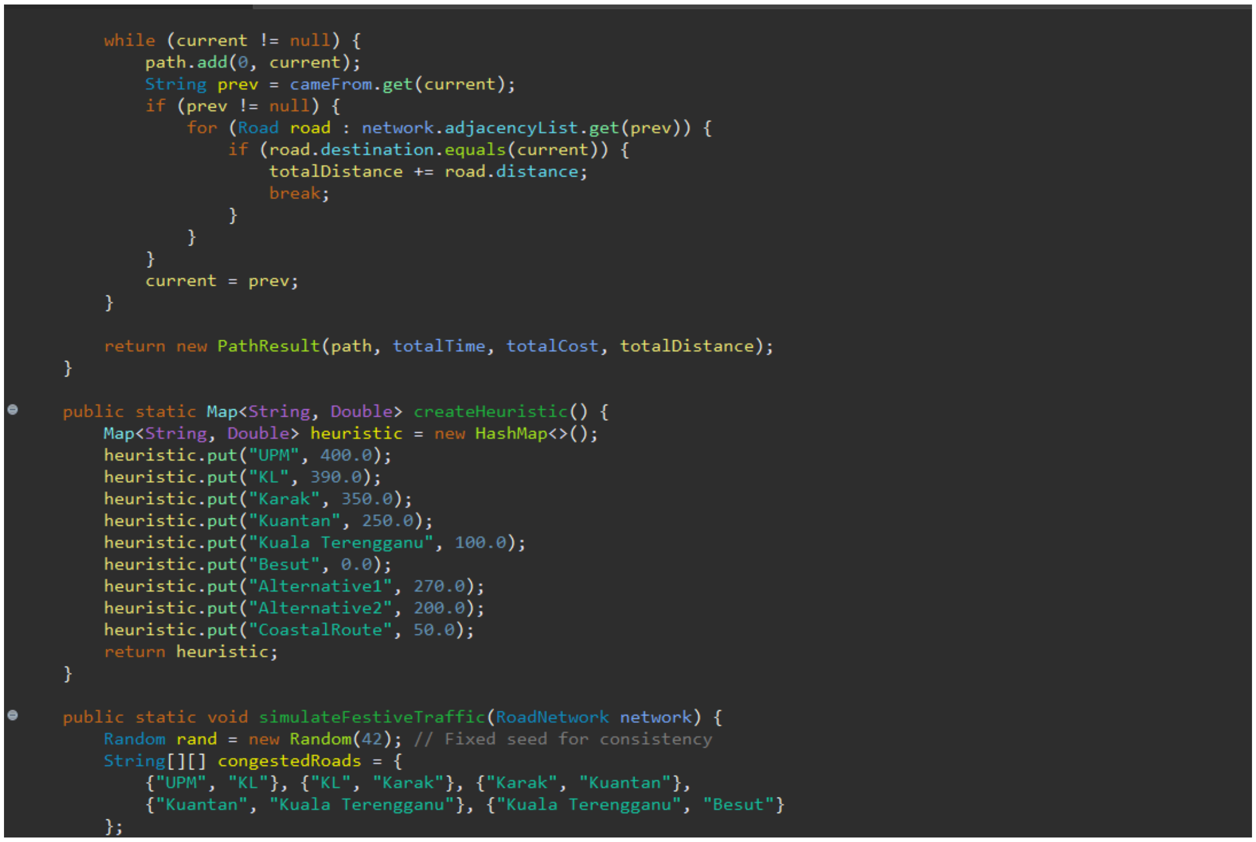
```java
        public Road(String destination, double distance, double tollCost, double baseTime) {
            this.destination = destination;
            this.distance = distance;
            this.tollCost = tollCost;
            this.baseTime = baseTime;
        }

        public double getCurrentTime() {
            return baseTime * congestionFactor;
        }

        public double getFuelCost() {
            return distance / 10.0 * 2.05;
        }

        public double getTotalCost() {
            return tollCost + getFuelCost();
        }
    }

    static class PathResult {
        List<String> path;
        double totalTime;
        double totalCost;
        double totalDistance;

        public PathResult(List<String> path, double totalTime, double totalCost, double totalDistance) {
            this.path = path;
            this.totalTime = totalTime;
            this.totalCost = totalCost;
            this.totalDistance = totalDistance;
        }
    }
```



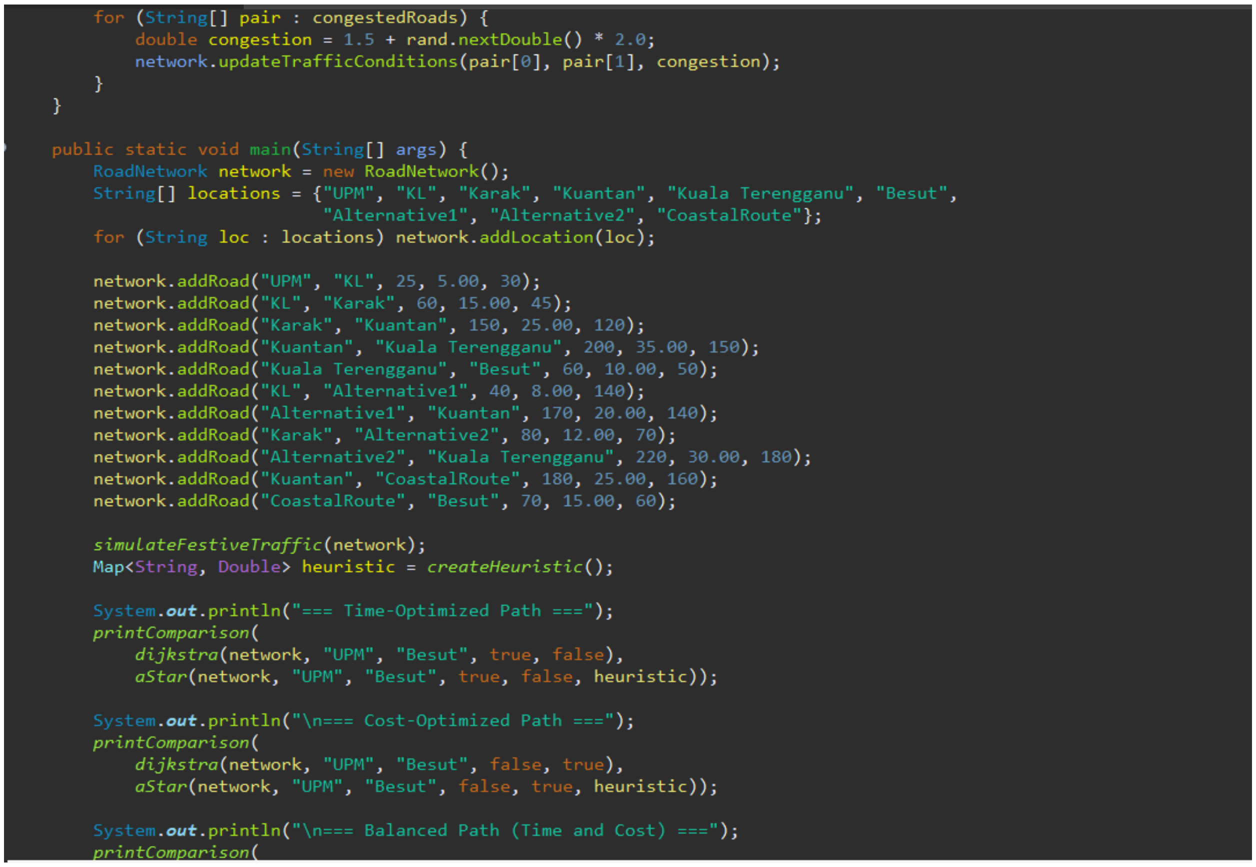
```java
    static class Node {
        String location;
        double cost;
        double time;
        double monetaryCost;

        public Node(String location, double cost, double time, double monetaryCost) {
            this.location = location;
            this.cost = cost;
            this.time = time;
            this.monetaryCost = monetaryCost;
        }
    }

    public static PathResult dijkstra(RoadNetwork network, String start, String end,
                                      boolean prioritizeTime, boolean prioritizeCost) {
        PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingDouble(a -> a.cost));

        Map<String, Double> dist = new HashMap<>();
        Map<String, String> prev = new HashMap<>();
        Map<String, Double> time = new HashMap<>();
        Map<String, Double> monetaryCost = new HashMap<>();

        for (String location : network.adjacencyList.keySet()) {
            dist.put(location, Double.POSITIVE_INFINITY);
            time.put(location, Double.POSITIVE_INFINITY);
            monetaryCost.put(location, Double.POSITIVE_INFINITY);
        }

        dist.put(start, 0.0);
        time.put(start, 0.0);
        monetaryCost.put(start, 0.0);
        pq.add(new Node(start, 0.0, 0.0, 0.0));

        while (!pq.isEmpty()) {
            Node current = pq.poll();
```



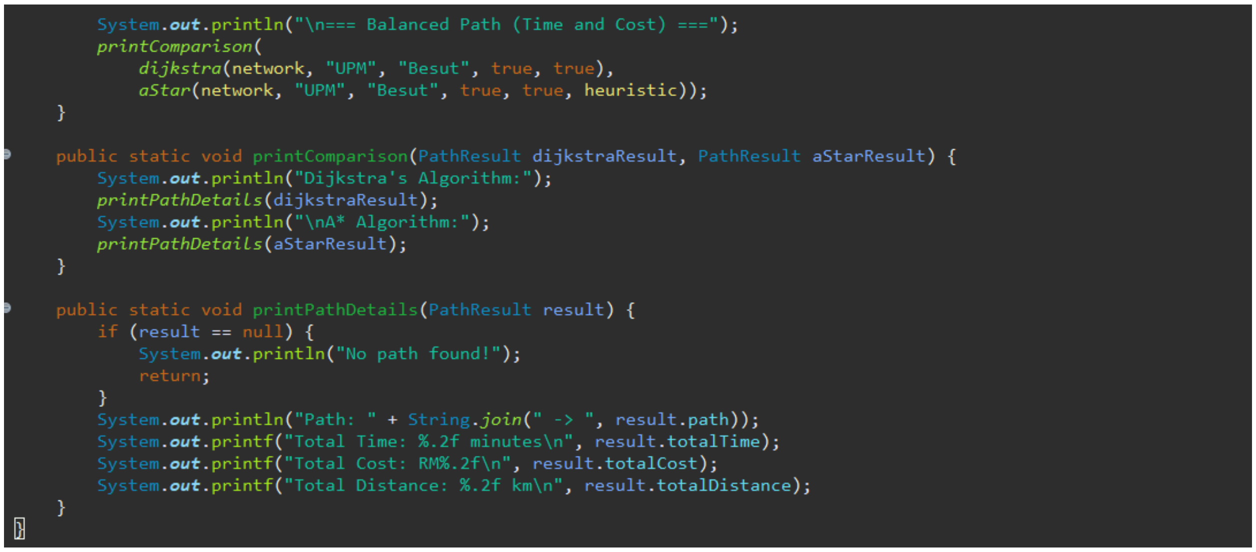
```
            if (current.location.equals(end)) break;

            for (Road road : network.adjacencyList.get(current.location)) {
                double newTime = time.get(current.location) + road.getCurrentTime();
                double newCost = monetaryCost.get(current.location) + road.getTotalCost();

                double combinedCost = (prioritizeTime && prioritizeCost) ? newTime * 0.5 + newCost * 0.5
                        : prioritizeTime ? newTime : newCost;

                if (combinedCost < dist.get(road.destination)) {
                    dist.put(road.destination, combinedCost);
                    time.put(road.destination, newTime);
                    monetaryCost.put(road.destination, newCost);
                    prev.put(road.destination, current.location);
                    pq.add(new Node(road.destination, combinedCost, newTime, newCost));
                }
            }
        }

        return buildPath(prev, network, start, end, time.get(end), monetaryCost.get(end));
    }

    public static PathResult aStar(RoadNetwork network, String start, String end,
                                   boolean prioritizeTime, boolean prioritizeCost,
                                   Map<String, Double> heuristicValues) {
        PriorityQueue<Node> pq = new PriorityQueue<>(
                Comparator.comparingDouble(a -> a.cost + heuristicValues.getOrDefault(a.location, 300.0)));

        Map<String, Double> gScore = new HashMap<>();
        Map<String, String> cameFrom = new HashMap<>();
        Map<String, Double> time = new HashMap<>();
        Map<String, Double> monetaryCost = new HashMap<>();

        for (String location : network.adjacencyList.keySet()) {
            gScore.put(location, Double.POSITIVE_INFINITY);
            time.put(location, Double.POSITIVE_INFINITY);
            monetaryCost.put(location, Double.POSITIVE_INFINITY);
        }
```



```
        while (current != null) {
            path.add(0, current);
            String prev = cameFrom.get(current);
            if (prev != null) {
                for (Road road : network.adjacencyList.get(prev)) {
                    if (road.destination.equals(current)) {
                        totalDistance += road.distance;
                        break;
                    }
                }
            }
            current = prev;
        }

        return new PathResult(path, totalTime, totalCost, totalDistance);
    }

    public static Map<String, Double> createHeuristic() {
        Map<String, Double> heuristic = new HashMap<>();
        heuristic.put("UPM", 400.0);
        heuristic.put("KL", 390.0);
        heuristic.put("Karak", 350.0);
        heuristic.put("Kuantan", 250.0);
        heuristic.put("Kuala Terengganu", 100.0);
        heuristic.put("Besut", 0.0);
        heuristic.put("Alternative1", 270.0);
        heuristic.put("Alternative2", 200.0);
        heuristic.put("CoastalRoute", 50.0);
        return heuristic;
    }

    public static void simulateFestiveTraffic(RoadNetwork network) {
        Random rand = new Random(42); // Fixed seed for consistency
        String[][] congestedRoads = {
            {"UPM", "KL"}, {"KL", "Karak"}, {"Karak", "Kuantan"},
            {"Kuantan", "Kuala Terengganu"}, {"Kuala Terengganu", "Besut"}
        };
```



```java
        for (String[] pair : congestedRoads) {
            double congestion = 1.5 + rand.nextDouble() * 2.0;
            network.updateTrafficConditions(pair[0], pair[1], congestion);
        }
    }

    public static void main(String[] args) {
        RoadNetwork network = new RoadNetwork();
        String[] locations = {"UPM", "KL", "Karak", "Kuantan", "Kuala Terengganu", "Besut",
                              "Alternative1", "Alternative2", "CoastalRoute"};
        for (String loc : locations) network.addLocation(loc);

        network.addRoad("UPM", "KL", 25, 5.00, 30);
        network.addRoad("KL", "Karak", 60, 15.00, 45);
        network.addRoad("Karak", "Kuantan", 150, 25.00, 120);
        network.addRoad("Kuantan", "Kuala Terengganu", 200, 35.00, 150);
        network.addRoad("Kuala Terengganu", "Besut", 60, 10.00, 50);
        network.addRoad("KL", "Alternative1", 40, 8.00, 140);
        network.addRoad("Alternative1", "Kuantan", 170, 20.00, 140);
        network.addRoad("Karak", "Alternative2", 80, 12.00, 70);
        network.addRoad("Alternative2", "Kuala Terengganu", 220, 30.00, 180);
        network.addRoad("Kuantan", "CoastalRoute", 180, 25.00, 160);
        network.addRoad("CoastalRoute", "Besut", 70, 15.00, 60);

        simulateFestiveTraffic(network);
        Map<String, Double> heuristic = createHeuristic();

        System.out.println("=== Time-Optimized Path ===");
        printComparison(
            dijkstra(network, "UPM", "Besut", true, false),
            aStar(network, "UPM", "Besut", true, false, heuristic));

        System.out.println("\n=== Cost-Optimized Path ===");
        printComparison(
            dijkstra(network, "UPM", "Besut", false, true),
            aStar(network, "UPM", "Besut", false, true, heuristic));

        System.out.println("\n=== Balanced Path (Time and Cost) ===");
        printComparison(
```



```java
        System.out.println("\n=== Balanced Path (Time and Cost) ===");
        printComparison(
            dijkstra(network, "UPM", "Besut", true, true),
            aStar(network, "UPM", "Besut", true, true, heuristic));
    }

    public static void printComparison(PathResult dijkstraResult, PathResult aStarResult) {
        System.out.println("Dijkstra's Algorithm:");
        printPathDetails(dijkstraResult);
        System.out.println("\nA* Algorithm:");
        printPathDetails(aStarResult);
    }

    public static void printPathDetails(PathResult result) {
        if (result == null) {
            System.out.println("No path found!");
            return;
        }
        System.out.println("Path: " + String.join(" -> ", result.path));
        System.out.printf("Total Time: %.2f minutes\n", result.totalTime);
        System.out.printf("Total Cost: RM%.2f\n", result.totalCost);
        System.out.printf("Total Distance: %.2f km\n", result.totalDistance);
    }
}
```

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

Group Name	4
Members	

Name	Email	Phone number
AYA SOFEA	214973@student.upm.edu.my	019-6068160
AZYAN SYAZWANI	215014@student.upm.edu.my	018-3181565
FATIN NASUHA	215190@student.upm.edu.my	011-55022981
SITI NAJMA IZZATY	216922@student.upm.edu.my	017-5977433

Problem scenario description	During major festive seasons such as Hari Raya Aidilfitri and Chinese New Year, traffic along the Lebuhraya Pantai Timur (LPT) becomes severely congested as thousands of travelers make their way from the Klang Valley area, including Universiti Putra Malaysia (UPM), to their hometowns in the east coast states like Terengganu, Kelantan, and Pahang. This seasonal surge results in long travel delays, especially at known bottlenecks such as Gombak Toll, Karak Highway, and critical highway junctions.

Why it is important	An optimized solution can help: ● Reduce travel time for students and families heading home. ● Minimize fuel consumption due to prolonged idling in traffic. ● Dynamically reroute vehicles to avoid severe congestion zones. ● Improve road safety by reducing driver stress and fatigue.
Problem specification	**Input**: ● Starting location (e.g., UPM) ● Destination (e.g., Besut) ● Departure time ● Vehicle type (optional: car, motorcycle, etc.) **Output**: ● Optimized route recommendation avoiding congested or high-delay areas ● Estimated travel time and route overview **Constraints**: ● Real-time traffic data from major road sensors or apps ● Toll charges along different routes ● Availability of alternative paths and rest stops ● Prediction of delays based on time and location
Potential solutions	● *Graph Algorithm:** Determines shortest paths based on live congestion reports. ● Greedy Approach: Allocates alternative routes dynamically to reduce bottlenecks. ● Dynamic Programming: Uses historical traffic trends to predict the best departure times.
Sketch (framework, flow, interface)	

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

Group Name	4
Members	

Name	Role
AYA SOFEA	
AZYAN SYAZWANI	
FATIN NASUHA	
SITI NAJMA IZZATY	

Problem statement	Traffic congestion along Lebuhraya Pantai Timur (LPT) during festive holidays causes long travel delays, increased fuel consumption, and driver frustration. Students and families traveling from UPM to Besut are among those heavily affected. There is a critical need for an optimized route-planning solution that can dynamically adapt to congestion conditions and improve travel efficiency during peak periods.
Objectives	● To develop a route optimization algorithm that suggests the most efficient travel path based on real-time and predictive traffic data. ● To implement a Java-based route planner that incorporates live traffic updates, toll information, and travel time estimation. ● To evaluate the performance of the algorithm using asymptotic analysis to ensure scalability and efficiency under varying traffic conditions.

Expected output	● Travelers will receive optimized route suggestions that avoid known congestion hotspots between UPM and Besut. ● The system will dynamically reroute vehicles based on live conditions, helping reduce total congestion. ● A fully documented online portfolio will showcase the system, algorithm design, test cases, performance results, and future recommendations.
Problem scenario description	During major Malaysian festivals like Hari Raya Aidilfitri and Chinese New Year, LPT experiences an enormous traffic surge. This is particularly true for routes from UPM to Terengganu, Kelantan, and Pahang, where students and families return to their hometowns. The sudden influx of vehicles causes massive traffic jams, especially near Gombak Toll, Karak Highway, and key junctions. These conditions result in journeys being delayed by several hours, stressing both drivers and infrastructure.
Why it is important	An optimized solution can help: ● Reduce total travel time and driver stress. ● Lower fuel usage and environmental impact. ● Enable smoother traffic flow by distributing vehicle load across alternate routes. ● Enhance road safety by reducing fatigue-related accidents and bottleneck frustration.
Problem specification	**Input:** ● Starting point (e.g., UPM) ● Destination (e.g., Besut) ● Departure time ● Vehicle type (optional) **Output:** ● Real-time optimized route suggestion ● Estimated travel duration ● Dynamic rerouting alert (if needed) **Constraints:** ● Live traffic data integration ● Toll charges ● Multiple alternative paths ● Festive traffic predictions ● Data accuracy and response time

Potential solutions	● Graph-Based Algorithms (Dijkstra's / A* Search): Use a graph representation of the highway system to find the shortest or fastest route. A* can factor in real-time congestion using heuristics. ● Greedy Algorithms: Provide fast rerouting decisions by selecting the best local segment at each junction, useful for quick updates. ● Dynamic Programming: Analyze historical traffic patterns to predict peak times and suggest ideal departure windows to avoid congestion buildup.
Sketch (framework, flow, interface)	
Methodology	

Milestone	Time
Scenario Refinement	Week 10
Algorithm Selection	Week 11
Code Implementation and Debugging	Week 12
Algorithm Analysis	Week 13
Online Portfolio and Presentation	Week 14

Milestone 1	Scenario Refinement
Date (Week)	10
Description/ sketch	The team held a brainstorming session to explore project ideas, considering titles such as *Flood Disaster*, *Traffic Jam on LPT*, and *Traffic Jam in Kuala Lumpur*. After discussion, the group decided to focus on festive season congestion along LPT, finalizing the title: “Traffic Optimization from UPM to Besut During Festive Season.” Key objectives and system scope were defined. Initial tasks were assigned, which included algorithm research, traffic data simulation, and interface flow planning.
Role	(see table below)

Member 1	Member 2	Member 3	Member 4
Fatin Nasuha Binti Zuraidi - Proposed title direction and drafted project scope	Aya Sofea Binti Rosli - Contributed ideas during brainstorming and helped structure early problem statement	Azyan Syazwani Binti Setia - Handled scope refinement and prepared early system design outline	Siti Najma Izzaty Binti Muhd Asyraf - Documented final title, objective, and coordinated team task division

Milestone 2	Algorithm Selection
Date (Week)	11
Description/ sketch	Each group member researched shortest-path algorithms including Dijkstra's, A*, and BFS. After comparing them based on complexity, real-time performance, and suitability for traffic routing, the team selected Dijkstra's Algorithm and A* for the project. Both are effective for weighted graphs that represent road networks with distances, travel times, and tolls. Pseudocode and key data structures like priority queues and distance tables were also prepared for implementation.
Role	

Member 1	Member 2	Member 3	Member 4
Fatin Nasuha Binti Zuraidi - Researched Dijkstra's Algorithm and explained its advantages	Aya Sofea Binti Rosli - Finalized algorithm selection and ensured alignment with the project's goals	Azyan Syazwani Binti Setia - Created a comparison table to evaluate algorithm efficiency	Siti Najma Izzaty Binti Muhd Asyraf - Gathered resources and tutorials on A* Algorithm and heuristic design

Milestone 3	Code Implementation and Debugging
Date (Week)	12
Description/ sketch	Full implementation of both Dijkstra and A* algorithms was completed using Java. The team integrated traffic simulation (random congestion factors) into the route calculation. A display function was created to output the result clearly. Bugs were identified in the congestion update logic and corrected. Heuristic values for A* were tested and adjusted.
Role	(see table below)

Member 1	Member 2	Member 3	Member 4
Fatin Nasuha Binti Zuraidi - Cleaned final code, and ensured consistency in output format	Aya Sofea Binti Rosli - Focused on path output display and code structure	Azyan Syazwani Binti Setia - Debugged time-cost calculation formulas	Siti Najma Izzaty Binti Muhd Asyraf - Implement traffic simulation logic and testing

Milestone 4	Algorithm Analysis
Date (Week)	13
Description/ sketch	Each algorithm was tested for three scenarios: time optimization, cost optimization, and balanced optimization. Output from each algorithm was analyzed and compared. A discussion section was prepared, explaining how each algorithm behaved under different traffic and priority settings. A performance comparison table was created.
Role	

Member 1	Member 2	Member 3	Member 4
Fatin Nasuha Binti Zuraidi - Analyzed time-based results and explained why A* performed better	Aya Sofea Binti Rosli - Studied cost-based output and helped write analysis section	Azyan Syazwani Binti Setia - Created the algorithm comparison table and visual result presentation	Siti Najma Izzaty Binti Muhd Asyraf - Drafted explanation paragraphs for testing and balanced route analysis

Milestone 5	Online Portfolio and Presentation
Date (Week)	14
Description/ sketch	Finalized all components. Compiled source code, flowcharts, testing data, and analysis into a structured online portfolio which is GitHub. Prepared presentation slides showcasing the problem statement, algorithm comparison, performance results and analysis. Conducted the final presentation. Held multiple practice sessions beforehand to ensure smooth delivery.
Role	

Member 1	Member 2	Member 3	Member 4
Fatin Nasuha Binti Zuraidi - Create slides on correctness algorithm and performance analysis.	Aya Sofea Binti Rosli - Create slides on problem definition, model algorithm and chosen algorithm.	Azyan Syazwani Binti Setia - Create slides on comparison between two algorithm and designing algorithm.	Siti Najma Izzaty Binti Muhd Asyraf - Create slides on result, analysis and conclusion.