
**FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY**

**DEPARTMENT OF COMMUNICATION TECHNOLOGY AND NETWORK**

**SEMESTER II 2024/2025**

---

**COURSE NAME** : **DESIGN AND ANALYSIS OF ALGORITHM**

**COURSE CODE** : **CSC4202 - 4**

**LECTURER'S NAME** : **DR. NUR ARZILAWATI BINTI MD. YUNUS**

**GROUP NO.** : **5**

**PROJECT TITLE** : **SCHOOL BUS ROUTING FOR RURAL AREA**

**GROUP MEMBERS :**

| NO. | NAME | MATRIC NO. |
|---|---|---|
| 1. | SITI NURINSYIRAH BINTI PAHRIN | 216990 |
| 2. | NURUL FIRZANA ASYIQIN BINTI ABDULLAH | 216663 |
| 3. | NIA SABRINA BINTI NASRUDDIN | 216019 |
| 4. | NUR SYAHIRAH BINTI JAMAL | 216475 |

# TABLE OF CONTENTS

# 1.0 Introduction

## 1.1 Overview of the Issue

In Kampung Sejahtera, a rural village characterized by its scattered residential layout and underdeveloped transport infrastructure, students face significant difficulties in accessing education. The distance between households and schools, combined with a lack of reliable and efficient transport services, has made daily commuting a considerable burden. Most students rely on a limited number of school buses that often operate on fixed routes which do not reflect the current needs of the student population. These routes are inefficient, leading to long wait times, prolonged travel durations, and inconsistent arrival times at school. As a result, students are frequently late or miss school entirely, which negatively impacts their academic performance and overall well-being. The issue highlights a critical need for a data-driven approach to improve route planning and resource allocation in rural school transportation.

## 1.2 Importance of the Study

This study is crucial for enhancing the quality of education in rural areas through improved transportation planning. By optimizing school bus routes, we can significantly reduce travel times, minimize fuel consumption, and ensure timely pick-up and drop-off of students. Efficient routes not only improve punctuality and attendance but also contribute to better academic engagement and reduced fatigue among students. Moreover, a well-structured transportation system promotes equity by ensuring that all students, regardless of their location, have equal access to education. From a logistical standpoint, the study also provides cost-saving benefits for school administrations by optimizing resource usage, including buses, fuel, and driver hours. Ultimately, this work demonstrates how technological solutions can address systemic rural challenges and enhance public service delivery.

### 1.3 Objectives

The primary objective of this project is to develop an algorithm that can efficiently generate optimized school bus routes for students living in Kampung Sejahtera. The algorithm aims to minimize the total travel distance and time required for school buses to pick up and drop off students, while also ensuring that each vehicle operates within its designated capacity. This is particularly important in a rural setting where transportation resources are limited and students are spread across a wide geographical area. By taking into account the locations of students and the constraints of the transportation system, the project seeks to create a solution that enhances punctuality, reduces operational costs, and guarantees fair access to education for all students. The effectiveness of the proposed solution will be demonstrated through an implementation in Java programming, allowing for simulation, testing, and performance evaluation under realistic conditions.

## 2.0 Problem Definition

### 2.1 Scenario Overview

Access to education is a fundamental right, yet in many rural parts of the world, infrastructural limitations continue to hinder students from receiving consistent schooling. In Malaysia, particularly in remote areas like Kampung Sejahtera, students face persistent challenges simply getting to school. The problem stems from the geographically scattered layout of the village, where homes are widely spaced, and roads are often underdeveloped. This rural dispersion, combined with minimal transportation infrastructure, makes school commuting a daily struggle for students. Most affected are those who live far from the main roads or school zones, where public or private transport services are either unavailable or unreliable.

The existing school bus routes intended to address this issue have proven to be highly inefficient. Buses often follow outdated or poorly optimized routes that result in long travel times and frequent delays. In some cases, students are picked up hours before school starts and only return home long after school ends, cutting into their rest and study time. The lack of an efficient system causes not just logistical inconvenience but also academic and psychological strain. Furthermore, limited fuel resources and rising operational costs put pressure on school

administrations, forcing them to make tough decisions about which areas to prioritize or neglect. This could lead to certain students being left out of the service entirely.

The goal of this project is to design and implement an algorithmic solution that can help plan school bus routes more efficiently. By modeling the problem as a resource-constrained path optimization problem, we aim to develop a tool that generates optimized bus routes based on location data, student density, and available resources. The algorithm will aim to maximize coverage while minimizing total distance traveled and delays. This ensures that students who are most in need of transport are prioritized, and resources like fuel and driver time are used wisely.

This project holds great significance, not just as an academic exercise, but also in its potential real-world impact. It demonstrates how computer science, particularly algorithm design and operations research, can be harnessed to solve logistical challenges in rural communities. By applying data-driven approaches to a real social issue, we not only gain practical experience in problem-solving and optimization, but also contribute toward improving educational accessibility for underserved populations. Ultimately, this project reflects the power of technology to bridge gaps and create more equitable systems in society.

## 3.0 Model Development

### 3.1 Data Type Required

In order to simulate Kampung Sejahtera's school bus routing system efficiently, a number of data types are needed to reflect the real-world issue in an organised and computationally accessible manner:

- Villages: Nodes

  In a graph, every village including the school is represented by a node. These can be represented by integers (e.g., 1, 2, 3…) or strings (e.g., “Village A”). Every node represents a real place where students would need to travel.

- Distances: Edges

  Each edge between nodes in the model represents the travel distance in kilometres, and the roads or trails that connect settlements are represented as edges. Finding the quickest or most effective paths between nodes requires this information.

- Bus Object

  A bus is seen as an item with certain characteristics:

  - capacity: the most pupils the bus is able to carry.
  - currentRoute: a sequentially visited list of settlements (nodes).
  - currentLoad: the total number of students enrolled, which cannot be more than the available space.

- Student Distribution

  This map connects the amount of students in need of transport to each village. It aids the algorithm in comprehending demand at every site and choosing which villages to give priority while routing.

### 3.2 Objectives Function & Constraints

#### 3.2.1 Objective Function

The main objective is to minimize the total distance traveled by all buses. This is formally written as:

$$\min \sum_{i=1}^{n} \sum_{(u,v)\in R_i} \text{distance}(u, v)$$

Where *Ri* is the set of edges (routes) used by bus *i*, and distance *(u,v)* represents the distance between two connected nodes.

#### 3.2.2 Constraints

- Every kid has to be fetched up from their community.
- At no time throughout the journey may the bus's capacity be surpassed.
- To save duplication and needless fuel use, each community is only visited once.
- The school serves as the central depot and is where all buses must start and stop.
- To guarantee the most efficient travel, the shortest routes should be used between nodes that have been visited.

### 3.3 Example of Traffic Data

The main structure for route optimisation is a weighted undirected graph, which is constructed using this distance table. Dijkstra's algorithm uses this graph to determine the shortest routes between each hamlet and the school, hence promoting the greedy route assignment technique.

| From | To | Distance (km) |
|---|---|---|
| School | Village A | 4 |
| Village A | Village B | 6 |
| Village B | Village C | 3 |
| School | Village C | 7 |

## 4.0 Algorithm Specification

To find the best method for addressing the school bus routing issue, this section examines and assesses a variety of algorithm design paradigms. The challenge is to use a small number of buses to effectively pick up students from dispersed towns without going over capacity and while minimizing the overall distance travelled.

### 4.1 Comparison of Algorithm Paradigms

The optimization strategy used for school bus routes must be able to account for real-world factors such as capacity and time, proximity-based selection, and graph traversal. The following table compares various algorithm paradigms:

| Algorithm Paradigm | Strengths | Weaknesses | Suitability |
|---|---|---|---|
| Greedy | Simple to implement, fast for decisions | May not find the global optimal solution | Good |
| Dynamic Programming | Excellent for optimization with overlapping subproblems | High time and space complexity for large datasets | Limited |
| Divide and Conquer | Efficient in recursive splitting | Poor fit for continuous routing with dependencies | Poor |
| Sorting | Helps with data organization | Not applicable to routing-based problems | Not useful |
| Dijkstra Algorithm | Excellent for finding shortest paths | Needs combination with higher-level routing strategy | Best fit |

### 4.2 Chosen Algorithm Approach

After evaluating many algorithm paradigms, including greedy, dynamic programming, divide and conquer, and graph-based algorithms, we concluded that a hybrid strategy combining Dijkstra's algorithm with a greedy approach is the most efficient and best solution to the school bus routing challenge in remote areas like Kampung Sejahtera, Nabawan.

Here the greedy approach is used to progressively assign communities to buses depending on proximity. That is to say, every bus will constantly pick the next unvisited village up to its maximum capacity. This method simplifies the decision-making process and ensures that every bus selects the best local route by avoiding the analysis of all possible route combinations (as would be required in more sophisticated algorithms like full dynamic programming).

To enhance the selfish selection process, we use Dijkstra's algorithm to find the quickest path between the bus's present location and the next chosen village. Because Dijkstra's algorithm properly identifies the shortest path between nodes in a weighted graph, like our villages and highways, it is perfect for this one. Integrating Dijkstra's algorithm into the route planning mechanism guarantees that every path a bus follows is not only straight and practical but also economical regarding journey distance.

This combination was chosen based on the equilibrium between computational efficiency and solution quality. Even if it might not always provide the best solution overall, combining Dijkstra's approach with a strictly greedy one ensures that every leg of the trip is as efficient as possible. Additionally, our chosen strategy has a lot less computational complexity than whole dynamic programming methods, that is, those required to solve the Traveling Salesman Problem hence it may be employed to real-world circumstances involving dozens of villages.

Finally, the chosen algorithmic approach uses Dijkstra's shortest path algorithm's dependability and optimality to generate a practical and effective solution for school bus routing in rural Malaysia combining the speed and simplicity of use of greedy logic.

## 5.0 Algorithm Design

### 5.1 Algorithm Overview

**Dijkstra's algorithm** is a graph-based approach that determines the shortest route between a given starting node and every other node in a weighted graph.
It constantly selects the node with the shortest total distance and repeats this procedure until it determines the shortest route to each node.

It is precise and ensures the best course of action, but it takes more time and space.
It's perfect for complicated maps where route accuracy is crucial, such as in rural locations with inconsistent road distances.

**Steps to Implement Dijkstra:**

1. Model the villages and school as graph nodes.
   Each village and the school are represented as nodes in a graph. Distances between them are the edge weights.

2. Build an adjacency list or matrix.
   This stores all the roads and their distances.

3. Apply Dijkstra's algorithm from the school node (0).
   Calculate the shortest path to all villages.

4. Use the shortest path result to determine the most efficient routes.
   This ensures minimum travel distance per bus.

5. Integrate with a route planner that respects bus capacity (e.g., only add to route if space allows).

The **Greedy algorithm** addresses the routing issue by choosing the closest unvisited community at each stage.
It concentrates on the best short-term decision rather than considering the overall picture.

This makes coding really simple and quick, and it's ideal for small to medium-sized rural communities with simple road systems.

Since it does not compare all options, though, it might not always provide the quickest overall path.

**Steps to Implement Greedy:**

1. Start the route at the school.
   Begin with a bus at the school (node 0).

2. While the bus has capacity, choose the nearest unvisited village.
   Use the distance matrix or a helper method to find the closest unvisited node.

3. Add the selected village to the bus route.
   Mark it as visited and reduce the remaining capacity.

4. Repeat until the bus is full or all villages are visited.

5. Return to school.
   Once a bus route is completed, return to the school and start the next bus if needed.

**5.2 Pseudocode for Route Optimization**

**Dijkstra Algorithm**

```
# Initialize graph as an adjacency list with V vertices
# Each list index represents a node, and stores a list of (neighbor, weight) pairs
Initialize graph as an adjacency list with V vertices

# Add undirected edges between connected villages with respective distances (weights)
For each village connection:
    Add edge from u to v with weight w
    Add edge from v to u with weight w (since it's undirected)

# Dijkstra's algorithm to find shortest paths from a starting node
Function dijkstra(start, graph, dist[], prev[]):
    # Initialize all distances to ∞ (unknown) and previous nodes to -1 (no parent)
```

```
    Set all dist[] = ∞ and prev[] = -1

    # Distance from start to itself is 0
    Set dist[start] = 0

    # Use a priority queue (min-heap) to always expand the nearest node first
    Initialize priority queue pq with (start, 0)

    While pq is not empty:
        # Get the node with the smallest known distance
        Pop (node, d) from pq

        # Skip this node if a shorter path to it has already been found
        If d > dist[node], skip

        # Explore all neighbors of the current node
        For each neighbor in graph[node]:
            # If a shorter path to neighbor is found
            If dist[neighbor] > dist[node] + edge weight:
                # Update the distance to this neighbor
                Update dist[neighbor] = dist[node] + edge weight

                # Record the current node as the parent of this neighbor
                Set prev[neighbor] = node

                # Add the neighbor to the priority queue for further exploration
                Add (neighbor, new dist) to pq

# Helper function to reconstruct the shortest path to a target node
Function getPath(target, prev[]):
    # Start from the target node and walk backward using the prev[] array
    Initialize empty path list
    While target is not -1:
        Add target to path list
        target = prev[target]

    # Reverse the path to get the correct order from start to target
    Reverse and return path

# Main execution
Main:
    # Create a graph with 6 nodes (e.g., node 0 = school, nodes 1-5 = villages)
    Create graph with 6 nodes

    # Add edges to represent roads with distances between school and villages
    Add sample edges with distances between nodes

    # Print all direct roads (edges) from the school (node 0)
    Print direct connections from school
```

```
# Run Dijkstra's algorithm from the school to compute shortest paths
Run dijkstra(0, graph, dist, prev)

# For each village (node 1 to 5):
For each village:
    # Print shortest distance from school to this village
    Print shortest distance from school

    # Reconstruct and print the full path from school to village
    Print full path from school using getPath(village, prev)
```

**Greedy Algorithm**

```
Define class Village:
    Attributes: id (int), students (int)

Function greedyRoute(dist[][], villages[], busCapacity):
    n ← number of nodes (villages + school)
    visited[] ← false for all n nodes
    Mark school (index 0) as visited

    totalStudents ← sum of students in all villages (excluding school)
    Print total number of students

    Initialize:
        totalDistance ← 0
        current ← 0 (start from school)
        load ← 0 (number of students currently in bus)
        trips ← 0 (bus trips counter)
        route ← [0] (start from school)

    While true:
        nearest ← -1
        minDist ← ∞

        # Find nearest unvisited village that can be picked up within bus capacity
        For i from 1 to n-1:
            If village i is not visited AND
               dist[current][i] < minDist AND
               load + villages[i].students ≤ busCapacity:
                nearest ← i
                minDist ← dist[current][i]

        # If no suitable village found, return to school
        If nearest = -1:
            If current ≠ 0:
```

```
            totalDistance += dist[current][0]
            Add school (0) to route
            trips += 1
            Print route and stats
            Reset route to [0], load ← 0, current ← 0

        # Check if all villages are visited
        allVisited ← true
        For i from 1 to n-1:
            If visited[i] = false:
                allVisited ← false
                Break

        If allVisited = true: break
        Else: continue

    # Move to nearest village
    totalDistance += dist[current][nearest]
    Add nearest to route
    load += villages[nearest].students
    visited[nearest] ← true
    current ← nearest

  # If last trip ends away from school, return to school
  If current ≠ 0:
    totalDistance += dist[current][0]
    Add school (0) to route
    trips += 1
    Print final route and stats

  Print total trips and total distance

Function formatRoute(route[]):
  Return a string representing each stop: School or Village i

Function getTripDistance(route[], dist[][]):
  total ← 0
  For i in 0 to route.size - 2:
    total += dist[route[i]][route[i+1]]
  Return total

Main:
  Define graph as distance matrix dist[6][6]
  Define villages[] with id and number of students
  Define busCapacity
  Call greedyRoute(dist, villages, busCapacity)
```

## 6.0 Implementation and Testing

### 6.1 Java Code Development

To solve the school bus routing problem in Kampung Sejahtera, two algorithms were developed in Java: Dijkstra's Algorithm and a Greedy Algorithm. Both aim to help a school bus pick up students from five villages and return to the school, using different routing strategies.

The DijkstraRouting program calculates the shortest path from the school to each village. It uses an adjacency list to represent the graph and a priority queue to select the nearest unvisited node. It shows the minimum distance and the exact path to reach each village from the school.

The GreedyRouting program simulates a bus route that picks up students based on the nearest village that can still fit within the bus capacity. It uses a distance matrix and marks visited villages to avoid repeated stops. The program outputs each trip, the number of students picked up, and the total distance traveled.

Both codes are developed in a way that allows easy comparison between the two approaches.

### 6.2 Challenges Faced

Several challenges were encountered during development:

- **Data Structure Differences**: Dijkstra's algorithm uses a graph with edges, while the Greedy algorithm uses a 2D distance matrix. Careful attention was needed to ensure the distances in both representations matched.

- **Handling Capacity Limits**: In the Greedy algorithm, the bus has a fixed capacity. The route had to be split into multiple trips, and the logic to track the number of students and reset the route was challenging.

- **Village Visit Tracking**: Ensuring each village was visited once and not skipped or repeated required extra checks in the Greedy method.

- **Unreachable Routes**: Dijkstra's algorithm had to handle the possibility that some villages were not directly connected to the school.

- **Readable Output**: For both codes, additional effort was made to display the paths clearly, using helper methods to reconstruct and format the route.

### 6.3 Example Test Cases and Results

**Test Configurations**

- Total Nodes: 6 (1 school + 5 villages)
- Bus Capacity: 9 students
- Student Distribution:

| Locations | Number of Students |
|---|---|
| School | - |
| Village 1 | 5 |
| Village 2 | 4 |
| Village 3 | 6 |
| Village 4 | 2 |
| Village 5 | 3 |
| Total | 20 |

## Results

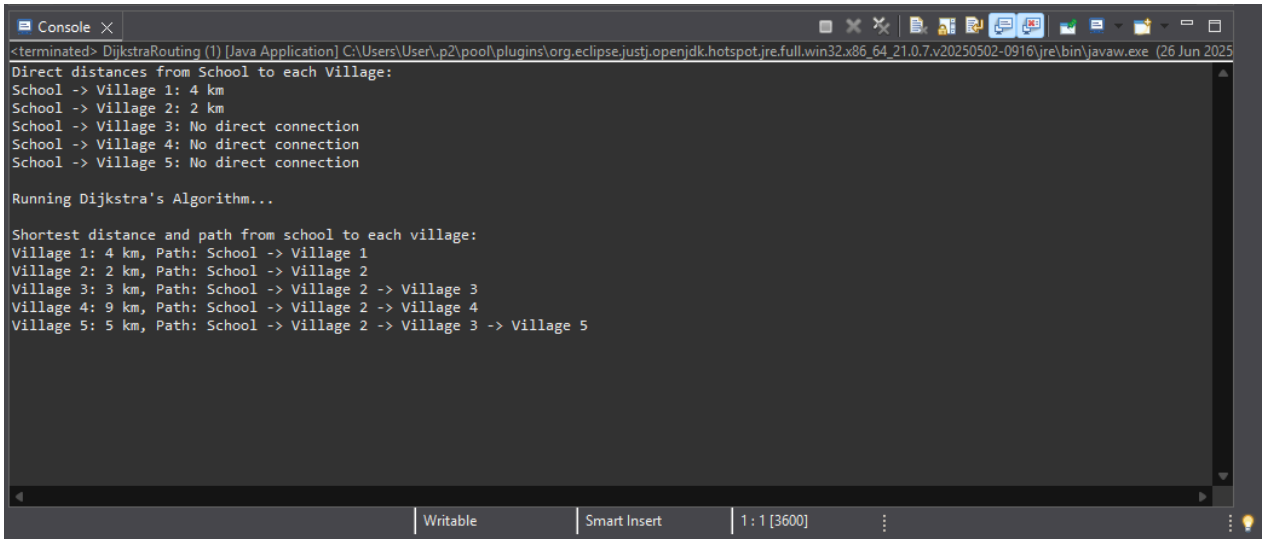


Figure 1. Output for Dijkstra Algorithm

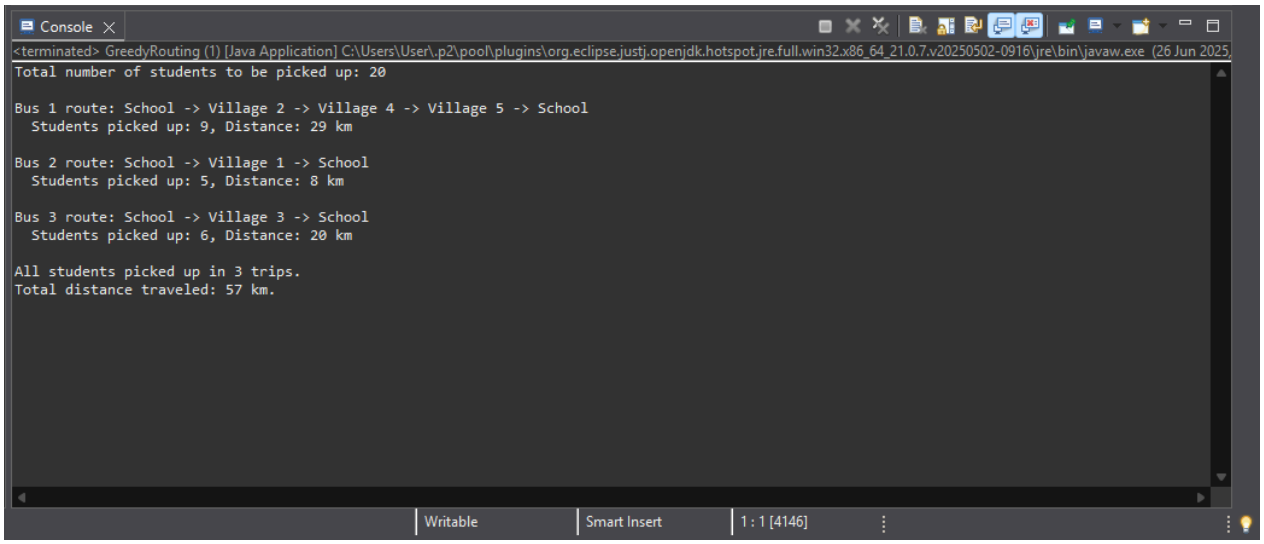


Figure 2. Output for Greedy Algorithm

## 7.0 Performance Analysis

### 7.1 Proof of Algorithm Correctness

**Why the Greedy Algorithm is Correct**

**A. Greedy Choice Step-by-Step**

The closest unexplored village that still fits under the bus's capacity is always chosen by the algorithm when making judgements. Without examining every possible combination, this localised decision-making process guarantees quick and straightforward routing decisions. It produces outcomes quickly and efficiently.

**B. Fulfills Routing Constraints**

Logic is used in each stage to determine whether the chosen hamlet is larger than the bus's remaining capacity. The bus goes back to the school before proceeding if it does. No hamlet is served outside bus restrictions thanks to this stringent verification, and the approach complies with practical limitations.

**C. Single Visit Guarantee**

Every town is only visited once thanks to the visited array. This keeps repeat pickups and needless diversions at bay. It ensures complete coverage of every location, which is a crucial prerequisite for school bus scheduling.

**D. Local Optimality Per Trip**

The Greedy algorithm aims to make every bus journey efficient, even when it isn't globally optimum. It reduces the trip distance for each route by always selecting the nearest settlement. This makes every leg of the trip distance-optimal when paired with Dijkstra.

**E. Practical Efficiency**

Compared to dynamic programming or exhaustive search, greedy is quicker and simpler to implement. It performs well on graphs of a moderate size and provides results that are adequate, particularly when dealing with transportation issues in rural areas where stringent limits apply and less-than-ideal routes are acceptable.

**Limitations of Dijkstra's Algorithm Compared to Greedy Algorithm**

**A. Capacity Awareness**

The number of pupils that a bus can accommodate is not taken into account by Dijkstra. Even if the community is too crowded to service, it might guide a bus to the quickest route.

**B. No Control Over Grouping**

It doesn't decide which combination of villages should be served in a single journey; instead, it operates at the individual path level.

**C. Not a Route Planner by Itself**

Dijkstra calculates pathways rather than complete pickup routes. To choose which places to visit and in what sequence, it has to be combined with a more complex algorithm (such as Greedy).

**7.2 Greedy Algorithm**

**A. Time Complexity**

The number of villages (n) that must be visited and the distribution of the pupils determine the Greedy Algorithm's time complexity.

- Worst Case: $O(n^2)$

  Due to capacity restrictions, the bus must, in the worst scenario, make several visits. During each journey, the algorithm searches through all of the villages that have not yet been visited in order to identify the closest legitimate one. For instance, if n is the number of villages and the bus only chooses one at a time (because of capacity constraints), the loop may run n times, scanning up to n nodes each time $\rightarrow O(n^2)$.

- Average Case: $O(n \log n)$

  The bus picks up several towns during a trip in normal circumstances, when there is a balanced distribution of students and a fair capacity. Sorting or priority queues, if used, can frequently be used more effectively to find the next village, cutting down on scanning time. Overall, this results in fewer iterations.

- Best Case: O(n)

  The method merely completes one full scan and constructs the route in a single loop, passing through every village in a single pass, providing linear time, if all the villages can be visited in a single trip (for example, due to low student numbers).

**B. Space Complexity**

Several data structures are used by the Greedy Algorithm to control decision-making and routing logic. The distance matrix, a 2D array that stores the distances between every pair of nodes, including the school and every settlement, takes up the most space. When choosing the closest village, this matrix allows for constant-time lookups and takes $O(n^2)$ space, where n is the total number of nodes. The program tracks which towns have previously been served using a visited array in addition to the matrix. O(n) space is needed for this array as well as variables for the current bus load, trip routes, and trip counts. The total space complexity is $O(n^2)$ as the distance matrix dominates the memory consumption. This works well for small to medium-sized village networks, but in bigger graphs, particularly in sparse rural road networks, it might be further optimised by utilising an adjacency list rather than a complete matrix.

**C. Scalability**

The Greedy technique is appropriate for school districts or rural planning since it is straightforward and effective for issues involving 10 – 100 villages. If the distance matrix is stored in memory (quadratic space), it does not scale well to hundreds or thousands of villages. It operates quickly since it doesn't go back or try different combinations, which is a nice balance between speed and solution quality in real-world applications.

### 7.3 Dijkstra’s Algorithm

#### A. Time Complexity

The number of vertices (V = villages + school) and edges (E = roads) determines Dijkstra's method.

- Worst Case: O((V + E) log V)
  Every node is added, removed, and the edge is relaxed once using a binary heap (priority queue). The priority queue operations are the source of the logarithmic term.

- Best Case: O(V log V)
  Fewer edge relaxations are required when the majority of nodes are directly related or when the first few pathways discovered are ideal. Since E ≈ V for sparse graphs (few roads), it approaches O(V log V).

- Average Case: O((V + E) log V)
  Assuming the graph is moderately sparse but not entirely linked (such as rural road networks), this is the usual runtime. For graphs with up to several thousand nodes, performance is still quick.

#### B. Space Complexity

Because of its memory-efficient architecture, Dijkstra's Algorithm works well with big graphs. The shortest known distances between the source node and any other node in the network are stored in a distance array of size O(V), where V is the number of vertices (villages plus the school). In addition, by tracking each node's parent throughout traversal, a prior node array of the same size, thus O(V), is utilized to rebuild the shortest path. An adjacency list is used to store the graph itself, which is very effective for sparse graphs such as rural transportation networks. O(V + E) space is needed for this representation, where E is the number of edges (roads). O(V + E) is the overall space complexity when all components are combined. Allowing Dijkstra's Algorithm to be scalable and appropriate for geographic applications with numerous nodes and connections in the actual world.

**C. Scalability**

When calculating the shortest path over huge maps or areas, Dijkstra is very scalable. Nevertheless, it just computes distances and ignores route restrictions (such as bus capacity, total load, and multi-stop journeys). Therefore, as in your project, it has to be paired with Greedy or another layer of decision-making to provide a complete routing solution.

## 8.0 Conclusion

### 8.1 Summary of Findings

We employed a combination of a greedy algorithm and a graph model to address the issue of school bus routing in this project. The distances between each town and the school were represented in a weighted graph as edges, and each village and the institution were represented as nodes. We were able to choose the closest unvisited village at each stage using the greedy method, while the graph made sure the distance calculations were correct.

Using this method, we were able to create effective and realistic bus routes that took into account capacity restrictions and reduced the total travel distance. It offered a straightforward but practical method for scheduling routes for several buses, particularly in rural regions where there are fewer road alternatives.

### 8.2 Limitations and Improvements

Although the suggested method produces realistic school bus routes, it has a number of drawbacks that may be overcome with future development. First off, the system ignores real-time traffic circumstances. The real-world deployment of the routes may be impacted by delays caused by road closures, traffic congestion, or weather disturbances, which could compromise their precision and dependability. The system would be more dynamic and responsive if real-time data from traffic APIs or GPS were integrated.

Second, the current approach addresses each bus route individually, without any coordination between them. When several buses operate in the same area, this might result in route inefficiencies or overlapping paths. Future improvements might include route sharing or merging

techniques to minimize redundancy. Thirdly, before routing, our method does not do any zoning or clustering. Using clustering algorithms like KMeans might aid in dividing the region into logical zones and allocating buses more efficiently, thereby maintaining even loads and reducing needless travel.

Lastly, although the acquisitive strategy is quick and straightforward, it may not always lead to the best course of action globally since it only considers the best option right now when making choices. Combining the present strategy with optimization methods like metaheuristics (e. g. , simulated annealing or genetic algorithms) may enhance the solution's overall quality and efficiency for bigger or more complicated regions.

### 8.3 Future Applications

The routing algorithm created by this study might have applications outside of school transportation. Emergency evacuation preparation is one essential area. This algorithm can aid in the development of rapid and effective evacuation routes in the event of natural disasters, such as floods, landslides, or wildfires, particularly in remote or difficult-to-reach locations where timing and coordination are essential.

Moreover, the approach may be used to improve logistical and delivery activities in rural locations. Route planning that takes into account distance and capacity limits might be beneficial for mobile clinics, healthcare supply chains, and delivery services, similar to the school bus example. The creation of intelligent urban transportation infrastructure is another potential use. Particularly in suburban or low-traffic locations, effective routing will be crucial for ridesharing services, autonomous vehicle fleets, and public buses as cities become more intelligent.

Finally, this project offers a solid educational basis for students studying algorithm design, optimization, and real-world problem-solving. It illustrates how fundamental concepts of computer science can be used to solve real-world problems that affect communities.

## 9.0 References


*A modified Dijkstra's algorithm for solving the problem of finding the maximum load path*. (2019, March 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/8711024

*Public transport route planning: Modified dijkstra's algorithm*. (2017, October 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/8093444

*Research on Application Technology of Expert Knowledge Graph based on greedy Iterative search Algorithm*. (2024, June 14). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/10692634

*Two heuristic algorithms for the minimum weighted connected vertex cover problem under greedy strategy*. (2022). IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/document/9938972

*Path-finding and Planning in a 3D environment an analysis using bidirectional versions of Dijkstra's, Weighted A*, and Greedy Best First search algorithms*. (2022, August 26). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/9909251

## Appendix

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

<table>
<tr><td>Group Name</td><td>Group 5</td></tr>
<tr><td>Members</td><td>
<table>
<tr><th>Name</th><th>Email</th><th>Phone number</th></tr>
<tr><td>SITI NURINSYIRAH BINTI PAHRIN</td><td>216990@student.upm.edu.my</td><td>010-6617935</td></tr>
<tr><td>NURUL FIRZANA ASYIQIN BINTI ABDULLAH</td><td>216663@student.upm.edu.my</td><td>011-69798327</td></tr>
<tr><td>NIA SABRINA BINTI NASRUDDIN</td><td>216019@student.upm.edu.my</td><td>011-37031816</td></tr>
<tr><td>NUR SYAHIRAH BINTI JAMAL</td><td>216475@student.upm.edu.my</td><td>011-19495767</td></tr>
</table>
</td></tr>
<tr><td>Problem scenario description</td><td>In rural districts such as Kampung Sejahtera, many students live far from school and rely on limited bus services. The current routes are inefficient, resulting in late arrivals and overuse of fuel and time. Optimizing the school bus routes is crucial to improve punctuality and maximize bus usage.</td></tr>
<tr><td>Why it is important</td><td>1. Reduces travel time and operational costs.<br>2. Ensures students arrive on time and safely.<br>3. Helps education departments make informed transportation decisions.<br>4. Supports equal access to education for rural students.</td></tr>
<tr><td>Problem specification</td><td>Input: Set of villages with student counts, distance matrix between villages and school, number of buses with capacities.<br>Constraints:<br>● Each student must be picked up.<br>● Each bus has a fixed capacity.<br>● All buses start and end at the school.<br>Output: Optimized routes for each bus that minimize travel distance and ensure constraints are met.</td></tr>
</table>

| | |
|---|---|
| Potential solutions | **Greedy + Graph Algorithm:** Use Dijkstra's algorithm to find the shortest paths and greedy logic to allocate students to buses.<br><br>**TSP Heuristic:** For scenarios with fewer villages, use traveling salesman approach for single-bus routing.<br><br>**Dynamic Programming (if needed):** For optimization with larger datasets (optional due to complexity). |
| Sketch (framework, flow, interface) | **Framework:**<br><br>● Java-based console application with modular code (Graph, Bus, RoutePlanner classes).<br>● Uses adjacency matrix or list for distance representation.<br><br>**Flow:**<br><br>1. Load village and distance data.<br>2. Assign villages to buses based on proximity and capacity.<br>3. Plan the shortest route for each bus using Dijkstra.<br>4. Output routes and total distance.<br><br>**Interface:**<br><br>● Text-based input/output in console.<br>● Optional: Display map via graph visualizer or text route output. |

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

<table>
<tr><td>Group Name</td><td>Group 5</td></tr>
<tr><td>Members</td><td>
<table>
<tr><th>Name</th><th>Role</th></tr>
<tr><td>SITI NURINSYIRAH BINTI PAHRIN</td><td>Java Development and Testing</td></tr>
<tr><td>NURUL FIRZANA ASYIQIN BINTI ABDULLAH</td><td>Analysis, Portfolio, Admin</td></tr>
<tr><td>NIA SABRINA BINTI NASRUDDIN</td><td>Report Writing: Problem and Summary</td></tr>
<tr><td>NUR SYAHIRAH BINTI JAMAL</td><td>Algorithm Design and Model</td></tr>
</table>
</td></tr>
<tr><td>Problem statement</td><td>Rural students often experience late arrivals and long commutes due to unoptimized school bus routes. There is a need to develop an algorithm that efficiently plans routes for school buses while respecting bus capacity and time constraints.</td></tr>
<tr><td>Objectives</td><td>● To minimize total travel time and distance for school buses.<br>● To ensure all students are picked up without exceeding bus capacities.<br>● To implement the solution in Java and validate its correctness and efficiency.<br>● To present the project using an online portfolio and demonstration.</td></tr>
<tr><td>Expected output</td><td>● A set of optimized bus routes (lists of stops per bus)<br>● Total distance traveled by each bus<br>● Java-based console output<br>● Graph or table representation of routes in portfolio</td></tr>
<tr><td>Problem scenario description</td><td>In Kampung Sejahtera, students are distributed across distant villages. The school has limited buses. Current bus routes are inefficient and lead to late student arrivals. Our project aims to simulate and optimize this routing problem using graph-based algorithm design.</td></tr>
<tr><td>Why it is important</td><td>1. Helps reduce operational costs and delays<br>2. Supports equal access to education in rural areas<br>3. Can be expanded for larger transport or emergency routing systems<br>4. Promotes use of computational algorithms for real-world problem-solving</td></tr>
<tr><td>Problem specification</td><td>Inputs:<br><br>● List of villages and number of students in each<br>● Distance matrix between locations</td></tr>
</table>

<table>
<tr><td></td><td>

- Number of available buses and their capacity

**Constraints:**

- Each bus can only pick up a certain number of students
- All buses must start and end at the school
- Each student must be assigned to exactly one bus

**Output:**

- Optimized routes for each bus (as a list of villages)
- Total distance for each route

</td></tr>
<tr><td>Potential solutions</td><td>

**Greedy + Dijkstra's Algorithm:** Assign villages to buses based on proximity and capacity, and find shortest paths using graph traversal.

**TSP Heuristic (optional):** For optimal routing within one bus group.

**Dynamic Programming (optional):** For exact TSP but only feasible for small node counts.

</td></tr>
<tr><td>Sketch (framework, flow, interface)</td><td>

**Framework:** Java console application with modular classes
**Flow:**

1. Input village and distance data
2. Allocate villages to buses
3. Compute optimized route using Dijkstra
4. Output routes and distance

**Interface:** Text-based CLI output and structured report on GitHub portfolio

</td></tr>
<tr><td>Methodology</td><td>

| Milestone | Time |
|---|---|
| scenario refinement | wk10 |
| find example solutions and suitable algorithms. Discuss in group why that solution and the example problems relate to the problem in the project | wk11 |
| edit the coding of the chosen problem and complete the coding. Debug | wk12 |
| conduct analysis of correctness and time complexity | wk13 |
| prepare online portfolio and presentation | wk14 |

</td></tr>
</table>

**Project Progress (Week 10 – Week 14)**

<table>
<tr><td>Milestone 1</td><td>Scenario refinement, find example solutions and suitable algorithms. Discuss in group why that solution and the example problems relate to the problem in the project</td></tr>
<tr><td>Date (week)</td><td>Week 10–11</td></tr>
<tr><td>Description/ sketch</td><td>● Finalized project scenario: School bus routing in rural areas<br>● Identified main goals: minimize route distance, ensure all students are picked up, avoid overcapacity<br>● Selected suitable algorithm: Greedy + Dijkstra<br>● Created project plan and divided team roles<br>● Rough design of the route planning logic and class structure (Java)<br>● Initial sketch of graph: Villages as nodes, distances as weighted edges</td></tr>
<tr><td>Role</td><td><table><tr><td>Member 1</td><td>Member 2</td><td>Member 3</td><td>Member 4</td></tr><tr><td>Supported planning and discussed Java structure</td><td>Documented scenario and organized team notes</td><td>Drafted initial report and project description</td><td>Designed algorithm model and planned pseudocode</td></tr></table></td></tr>
</table>

<table>
<tr><td>Milestone 2</td><td>Eedit the coding of the chosen problem and complete the coding, conduct analysis of correctness and time complexity</td></tr>
<tr><td>Date (Wk)</td><td>Week 12–13</td></tr>
<tr><td>Description/ sketch</td><td>● Completed Java implementation: route allocation, shortest path with Dijkstra<br>● Performed testing with different village and student data<br>● Finalized algorithm analysis: time complexity and correctness<br>● Set up online GitHub portfolio and uploaded content (report, code, test cases)<br>● Drafted final report and filled project forms<br>● Sketched route output table and graph map for visualization</td></tr>
<tr><td>Role</td><td><table><tr><td>Member 1</td><td>Member 2</td><td>Member 3</td><td>Member 4</td></tr><tr><td>Developed and tested Java code</td><td>Set up portfolio, uploaded files, completed appendix</td><td>Wrote final sections: introduction, problem and conclusion</td><td>Finalized pseudocode, flowchart, time complexity</td></tr></table></td></tr>
</table>

**Coding**

**Dijkstra's Algorithm**

```
package testing2;

import java.util.*;

public class DijkstraRouting {
   static class Edge {
      int to, weight;
      Edge(int t, int w) { to = t; weight = w; }
   }

   public static void dijkstra(int start, List<List<Edge>> graph, int[] dist, int[] prev) {
      Arrays.fill(dist, Integer.MAX_VALUE);
      Arrays.fill(prev, -1);
      dist[start] = 0;

      PriorityQueue<int[]> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a[1]));
      pq.offer(new int[]{start, 0});

      while (!pq.isEmpty()) {
         int[] curr = pq.poll();
         int node = curr[0], d = curr[1];

         if (d > dist[node]) continue;

         for (Edge edge : graph.get(node)) {
            int next = edge.to, weight = edge.weight;
            if (dist[next] > dist[node] + weight) {
```

```
                dist[next] = dist[node] + weight;
                prev[next] = node;
                pq.offer(new int[]{next, dist[next]});
            }
        }
    }
}

public static List<Integer> getPath(int target, int[] prev) {
    List<Integer> path = new ArrayList<>();
    for (int at = target; at != -1; at = prev[at]) {
        path.add(at);
    }
    Collections.reverse(path);
    return path;
}

public static void main(String[] args) {
    int V = 6; // School + 5 villages
    List<List<Edge>> graph = new ArrayList<>();
    for (int i = 0; i < V; i++) graph.add(new ArrayList<>());

    // Sample village graph
    graph.get(0).add(new Edge(1, 4)); graph.get(1).add(new Edge(0, 4));
    graph.get(0).add(new Edge(2, 2)); graph.get(2).add(new Edge(0, 2));
    graph.get(1).add(new Edge(3, 3)); graph.get(3).add(new Edge(1, 3));
    graph.get(2).add(new Edge(3, 1)); graph.get(3).add(new Edge(2, 1));
    graph.get(2).add(new Edge(4, 7)); graph.get(4).add(new Edge(2, 7));
    graph.get(3).add(new Edge(5, 2)); graph.get(5).add(new Edge(3, 2));
```

```
int[] dist = new int[V];
int[] prev = new int[V];

// 🔹 Print direct connections from school
System.out.println("Direct distances from School to each Village:");
for (int i = 1; i <= 5; i++) {
    boolean directFound = false;
    for (Edge edge : graph.get(0)) {
        if (edge.to == i) {
            System.out.println("School -> Village " + i + ": " + edge.weight + " km");
            directFound = true;
            break;
        }
    }
    if (!directFound) {
        System.out.println("School -> Village " + i + ": No direct connection");
    }
}

System.out.println("\nRunning Dijkstra's Algorithm...\n");

dijkstra(0, graph, dist, prev);

System.out.println("Shortest distance and path from school to each village:");
for (int i = 1; i < V; i++) {
    if (dist[i] == Integer.MAX_VALUE) {
        System.out.println("Village " + i + " is unreachable from school.");
    } else {
```

```
            List<Integer> path = getPath(i, prev);
            System.out.print("Village " + i + ": " + dist[i] + " km, Path: ");
            for (int j = 0; j < path.size(); j++) {
                if (j > 0) System.out.print(" -> ");
                System.out.print(path.get(j) == 0 ? "School" : "Village " + path.get(j));
            }
            System.out.println();
        }
    }
  }
}
```

**Greedy Algorithm**

```
package testing2;

import java.util.*;

public class GreedyRouting {
   static class Village {
      int id, students;
      Village(int id, int students) {
         this.id = id;
         this.students = students;
      }
   }

   public static void greedyRoute(int[][] dist, Village[] villages, int busCapacity) {
      int n = dist.length;
      boolean[] visited = new boolean[n];
      visited[0] = true;

      int totalStudents = 0;
      for (int i = 1; i < n; i++) {
         totalStudents += villages[i].students;
      }

      System.out.println("Total number of students to be picked up: " + totalStudents + "\n");

      int totalDistance = 0;
      int current = 0;
      int load = 0;
```

```
        int trips = 0;
        List<Integer> route = new ArrayList<>();
        route.add(0);

        while (true) {
            int nearest = -1;
            int minDist = Integer.MAX_VALUE;

            for (int i = 1; i < n; i++) {
                if (!visited[i] && dist[current][i] < minDist && load + villages[i].students <= busCapacity) {
                    nearest = i;
                    minDist = dist[current][i];
                }
            }

            if (nearest == -1) {
                if (current != 0) {
                    totalDistance += dist[current][0];
                    route.add(0);
                    System.out.println("Bus " + (++trips) + " route: " + formatRoute(route));
                    System.out.println("  Students picked up: " + load + ", Distance: " + getTripDistance(route,
dist) + " km\n");
                    route.clear();
                    route.add(0);
                    current = 0;
                    load = 0;
                }

                boolean allVisited = true;
```

```
            for (int i = 1; i < n; i++) {
                if (!visited[i]) {
                    allVisited = false;
                    break;
                }
            }

            if (allVisited) break;
            else continue;
        }

        totalDistance += dist[current][nearest];
        route.add(nearest);
        load += villages[nearest].students;
        visited[nearest] = true;
        current = nearest;
    }

    if (current != 0) {
        totalDistance += dist[current][0];
        route.add(0);
        System.out.println("Bus " + (++trips) + " route: " + formatRoute(route));
        System.out.println("  Students picked up: " + load + ", Distance: " + getTripDistance(route, dist) + "
km\n");
    }

    System.out.println("All students picked up in " + trips + " trips.");
    System.out.println("Total distance traveled: " + totalDistance + " km.");
  }
```

```
static String formatRoute(List<Integer> route) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < route.size(); i++) {
        if (i > 0) sb.append(" -> ");
        sb.append(route.get(i) == 0 ? "School" : "Village " + route.get(i));
    }
    return sb.toString();
}

static int getTripDistance(List<Integer> route, int[][] dist) {
    int total = 0;
    for (int i = 0; i < route.size() - 1; i++) {
        total += dist[route.get(i)][route.get(i + 1)];
    }
    return total;
}

public static void main(String[] args) {
    int INF = 10;

    int[][] dist = {
        {0,   4,   2, INF, INF, INF},
        {4,   0, INF,   3, INF, INF},
        {2, INF,   0,   1,   7, INF},
        {INF, 3,   1,   0, INF,   2},
        {INF, INF, 7, INF,   0, INF},
        {INF, INF, INF, 2, INF,   0}
    };
```

```
        Village[] villages = new Village[] {
            new Village(0, 0),  // School
            new Village(1, 5),
            new Village(2, 4),
            new Village(3, 6),
            new Village(4, 2),
            new Village(5, 3)
        };

        int busCapacity = 9;
        greedyRoute(dist, villages, busCapacity);
    }
}
```