A·P·I·I·T
Inspire love for learning

STAFFORDSHIRE UNIVERSITY

BSc (Hons) in Computing

Level 5

INDIVIDUAL ASSIGNMENT

Module Code & Title:

COMP50004 – Databases and Data Structures- 2

Prepared By:

Student Number	Student Name	Cohort
CB010205	Pathirage Pasan Dulmin Perera	IF2321COM

Date of Submission: 25 Sep 2023

Instructor: Mr. Guhanathan Poravi

MARKING CRITERIA	%	MARKS OBTAINED
TOTAL (%)		

Table of Contents

1 Introduction for Milestone 1

This report addresses a common challenge faced by our company in organizing goods within the delivery vans. The objective is to create an optimal loading sequence that ensures both time and space efficiency. To tackle this issue, we propose a solution based on a set of well-defined functions and data structures, which will enable us to generate a structured order document for goods in the van and provide an invoice for each customer, ultimately enhancing the precision and professionalism of our delivery services.

2 Analysis for Milestone 1

2.1 Data Structure

A data structure is a set of data with well-defined actions and attributes. To solve the above scenario selected data structure are arrays and stack. Advantages of arrays, they are used to store and retrieve data in a specific order. Stack data structure is easy to implement using arrays and also they are used to store function calls and their states

Data stored in the data structures

Data Structure	Relevant Data
Arrays	To store details of the deleivey packages. (packageId, customerName, cutomerId,, Dimensions,)
Stack	To load packages to the delivery van

2.2 Sample Data

Position of the package in the van.

Package id	Customer Id	Customer Name	Description	Position
P1	C1	Jane Stone	Clothes	1
P2	C2	Dehan Perera	Books	3
P3	C3	Henry Fernando	TV	5
P4	C4	Amal Fernando	Oven	4
P5	C5	Pasan Dulmin	Beater	2
P6	C6	Mary Perera	Clothes	7
P7	C7	Mary Rodrigo	Bag	6
P8	C8	Joel Jayawardena	Mobile phone	8

Invoice

InvoiceID IN1

CustomerID C1

CustomerName Jane Stone

CustomerAddress No: 2, Riverside, Colombo

CustomerContact jane@123mail.com

PackageID P1,

Charge. Rs.50

Description Clothes

InvoiceID IN2

CustomerID CU2

CustomerName Dehan Perera

CustomerAddress No:55, Park Street, Maharagama.

CustomerContact dehan@gmail.com

PackageID PR002

Charge Rs. 80

Description Books

2.3 Pseudo Code

```
function find_optimal_route(loaded_goods):
    current_location = starting_location
    optimal_route = []

    while loaded_goods:
        nearest_good = find_nearest_good(current_location, loaded_goods)
        optimal_route.append(nearest_good)
        loaded_goods.remove(nearest_good)
        current_location = nearest_good.destination

    return optimal_route

function find_nearest_good(current_location, loaded_goods):
    nearest_good = None
    min_distance = infinity

    for good in loaded_goods:
        distance = calculate_distance(current_location, good.destination)
        if distance < min_distance:
            min_distance = distance
            nearest_good = good

    return nearest_good

# Load goods in the order of the optimal route
optimal_route = find_optimal_route(loaded_goods)

# Sample data
starting_location = { coordinates: (0, 0) }
```

```
# Load goods into the van following the optimal route
van_load = []
current_location = starting_location

for good in optimal_route:
    van_load.append(good)
    current_location = good.destination

# Generate order document for loaded goods
order_document = generate_order_document(van_load)

# Generate invoices for each customer
customer_invoices = []
for good in van_load:
    order = find_order_by_good(orders, good)
    invoice = generate_customer_invoice(order)
    customer_invoices.append(invoice)

print("Optimal Route:")
for good in optimal_route:
    print(good.name)

print("Order Document:")
print(order_document)

for invoice in customer_invoices:
    print("Customer Invoice:")
    print(invoice)
```

2.4 Issues that Pseudo Code will not solve

No Consideration for Time Windows: The pseudocode doesn't consider time windows for deliveries. In real-world scenarios, deliveries must be made within specific time slots, which can significantly impact the route.

Assumes Static Data: It assumes that the data (e.g., distances, goods' locations) doesn't change during the delivery process. Data can change due to traffic, weather, or customer cancellations.

No Handling of Failed Deliveries: There's no provision for handling failed deliveries (e.g., if the customer is not available). Real-world delivery systems need mechanisms for dealing with such scenarios.

2.5 Assumption

Assumptions -

Euclidean Distance: The pseudocode assumes that distances between locations are calculated using Euclidean distance. Road networks and travel times may not adhere to this idealized distance measure.

Simplified Goods Representation: The loaded goods are represented as objects with names and destination coordinates. This simplification may not capture the complexity of real-world goods, such as varying sizes, weights, or special handling requirements.

Limitations -

Greedy Approach: The algorithm relies on a greedy approach to find the nearest good at each step, which may not guarantee the optimal solution in all cases.

Capacity Constraints: The pseudocode does not check or manage the capacity constraints of the delivery van. It assumes that all goods will fit within the van, which may not be the case in practice.

3 Introduction for Milestone 2

While commercial route planning software offers sophisticated solutions, the associated costs may not always be justifiable for every business. In response to this challenge, we present an algorithmic approach to van delivery scheduling. This report outlines the proposed algorithm, the necessary data structures, sample data for testing, and a pseudocode structure. By offering an alternative solution, this report aims to assist the company in making an informed decision on whether to employ a developer for their delivery scheduling needs, while also highlighting potential limitations and issues that the algorithm may encounter.

4 Analysis for Milestone 2

4.1 Data Structure

Data structures are a collection of basic and complicated forms that are used to organize data for a certain purpose. Data structures selected for the above is arayList, 2d arrays, and stack. When we add or delete elements from arraylists, their capacity adjusts automatically. As a result, arraylists are often referred to as dynamic arrays. 2D array is a array of array. arrays are useful when you want to store data as a tabular form, like a table with rows and columns. The stack is a linear data structure for storing a collection of things. It is based on the Last-In-First-Out (LIFO) principle.

Data Structure	Relevant Data
ArrayList	To store details of the Packages. (packagerId, customerName, cutomerId, Distances, Dimensions)
Two Dimensional Arrays	To store distances between each location
Stack	To create the delivery order. To decide unloading and loading of packages.

4.2 Sample Data

The delivery route

Package Id	**Customer Id**	**Customer Name**	**Delivery address**	**Description**	**Coordinated (Latitude, longitude)**
P1	C1	Jane Stone	No: 2, Riverside, Colombo	Clothes	39.99 58.99
P2	C2	Dehan Perera	No:55, Park Street, Maharagama.	Books	59.028 49.839
P3	C3	Henry Fernando	No:45, Lake Road, Kotte	TV	62.884 -87.99

P4	C4	Amal Fernando	No:99, flower road, Colombo	Oven	59.783 29.77

A list of grid co-ordinates, goods etc that would act as a backup if their IT failed

Package Id	**Customer Id**	**Customer Name**	**Location**	**Latitude**	**Longitude**
P1	C1	Jane Stone	No: 2, Riverside, Colombo	39.99	58.99
P2	C2	Dehan Perera	No:55, Park Street, Maharagama.	59.028	49.839
P3	C3	Henry Fernando	No:45, Lake Road, Kotte	62.884	-87.99
P4	C4	Amal Fernando	No:99, flower road, Colombo	59.783	-29.77

4.3 Functionality

```
2 usages
public class VanDeliverySchedule {
    3 usages
    private List<Integer> deliveryLocations = new ArrayList<>();
    8 usages
    private List<Integer> visitedLocations = new ArrayList<>();
    4 usages
    private int currentLocation;
    4 usages
    private double totalDistance;
    4 usages
    private int[][] distanceMatrix;

```

```
1 usage
public VanDeliverySchedule(int[][] distanceMatrix) {
    this.distanceMatrix = distanceMatrix;
    int numLocations = distanceMatrix.length;
    for (int i = 0; i < numLocations; i++) {
        deliveryLocations.add(i + 1); // Locations are 1-based
    }
    visitedLocations.add(1); // Start at the first location (assuming 1 as the starting point)
    currentLocation = 1; // Set the current location as the starting point
    totalDistance = 0;
}

```

```
    2 usages
    private double calculateDistance(int point1, int point2) {
        // Subtract 1 from point1 and point2 to account for 1-based indexing
        int adjustedPoint1 = point1 - 1;
        int adjustedPoint2 = point2 - 1;

        // Check if the adjusted points are within the bounds of the matrix
        if (adjustedPoint1 >= 0 && adjustedPoint1 < distanceMatrix.length &&
                adjustedPoint2 >= 0 && adjustedPoint2 < distanceMatrix.length) {
            return distanceMatrix[adjustedPoint1][adjustedPoint2];
        } else {
            // Handle an invalid request (points outside the matrix)
            throw new IllegalArgumentException("Invalid points");
        }
    }
```

```
    public void findOptimalRoute() {
        while (visitedLocations.size() < deliveryLocations.size()) {
            double minDistance = Double.POSITIVE_INFINITY;
            int nearestLocation = -1;

            for (int location : deliveryLocations) {
                if (!visitedLocations.contains(location)) {
                    double distanceToLocation = calculateDistance(currentLocation, location);
                    if (distanceToLocation < minDistance) {
                        minDistance = distanceToLocation;
                        nearestLocation = location;
                    }
                }
            }

            if (nearestLocation != -1) {
                visitedLocations.add(nearestLocation);
                totalDistance += minDistance;
                currentLocation = nearestLocation;
            }
        }

        // Complete the loop by returning to the starting location
        totalDistance += calculateDistance(currentLocation, visitedLocations.get(0));
        visitedLocations.add(visitedLocations.get(0));

        // Output the results
        System.out.println("Optimal Delivery Route:");
        for (int location : visitedLocations) {
            System.out.println(location);
        }

        System.out.println("Total Distance Traveled: " + totalDistance);
    }
}
```

main.java | VanDeliverySchedule.java

```java
import java.util.ArrayList;
import java.util.List;

public class main {
    public static void main(String[] args) {
        int[][] distanceMatrix = {
                {0, 5, 8, 9, 7},
                {5, 0, 10, 6, 8},
                {8, 10, 0, 4, 12},
                {9, 6, 4, 0, 6},
                {7, 8, 12, 6, 0}
        };

        VanDeliverySchedule van = new VanDeliverySchedule(distanceMatrix);
        van.findOptimalRoute();
    }
}

```

4.4 Pseudo Code

```
# Pseudocode for Van Delivery Schedule - Greedy Approach with Required Serialization/Deserialization

# Define the data structures
Define a list "delivery_locations"  # List of all delivery locations
Define a list "visited_locations"   # List of visited locations
Define a variable "current_location"  # The current location of the van
Define a variable "total_distance"   # Total distance traveled
Define a variable "backup_goods"   # List of backup goods

# Initialize data structures
delivery_locations = []  # Fill this list with the coordinates of delivery locations
visited_locations = [0]  # Start at the first location (assuming 0 as the starting point)
current_location = 0   # Set the current location as the starting point
total_distance = 0    # Initialize the total distance to 0
backup_goods = []    # Initialize the backup goods list

# Function to calculate distance between two points
Function calculate_distance(point1, point2):
   # Pseudocode for calculating Euclidean distance between two points
   ...

# Main algorithm loop
While there are unvisited locations:
   min_distance = Infinity
   nearest_location = None

   For each location in delivery_locations:
      If location is not in visited_locations:
```

```
            distance_to_location = calculate_distance(current_location, location)
            If distance_to_location < min_distance:
                min_distance = distance_to_location
                nearest_location = location

    If nearest_location is not None:
        visited_locations.append(nearest_location)
        total_distance += min_distance
        current_location = nearest_location

# Complete the loop by returning to the starting location
total_distance += calculate_distance(current_location, visited_locations[0])
visited_locations.append(visited_locations[0])

# Output the results
Print("Optimal Delivery Route:")
For each location in visited_locations:
    Print(location)
Print("Total Distance Traveled:", total_distance)

# Store backup goods
backup_goods = []  # Fill this list with backup goods data

# Serialize the state to continue from this point if needed
serialized_state = {
    "visited_locations": visited_locations,
    "current_location": current_location,
    "total_distance": total_distance
}
# ... Store 'serialized_state' in a secure way (e.g., temporary memory) ...
```

```
# Deserialize the state to resume the algorithm (this should be done
automatically)
state_to_resume = {
   "visited_locations": [],
   "current_location": 0,
   "total_distance": 0
}
# ... Load 'serialized_state' from where it was stored ...
state_to_resume = serialized_state

# Set the algorithm state to the deserialized state
visited_locations = state_to_resume["visited_locations"]
current_location = state_to_resume["current_location"]
total_distance = state_to_resume["total_distance"]
```

4.5 Issues that Pseudo Code will not solve

Serialization Limitations: While the pseudocode includes serialization/deserialization for checkpointing, it doesn't address potential issues like data loss or security concerns when saving and loading algorithm states.

Backup Goods Handling: It assumes a simple storage mechanism for backup goods without considering the complexity of managing goods during unexpected events or failures.

Assumption of Continuous Execution: The algorithm assumes continuous execution, which might not be practical in a real-world scenario where the process can be interrupted and resumed at any time.

No Error Handling: It lacks error handling mechanisms for unexpected situations, such as data corruption during serialization/deserialization or failed route calculations.

4.6 Assumption

Assumptions -

Static Data: It assumes that the data, including the coordinates of delivery locations and goods, remains static throughout the execution. In practice, this data can change due to various factors.

Perfect Execution: The pseudocode assumes perfect execution without considering factors such as errors, disruptions, or external influences that can affect delivery operations.

Limitations -

Limited Error Handling: The code lacks error-handling mechanisms for scenarios like data validation errors or network communication failures.

Scalability: Performance may degrade for a large number of delivery locations, as it employs a simple linear search to find the nearest location.