Garbage Management System

School**: Bsc Computer Science**

Academic year**:** **2023/24**

Student ID: **20364583**

Word count: **3604**

Category	Details
Assignment/Essay Title	**Garbage Management System Report**
Module Code	**COMP2034**
Module Title	**C++**

Table of Contents

1.0 Executive Summary

This report documents the design, implementation, and evaluation of an advanced Garbage Management System (GMS) developed using C++ programming. The system is designed to optimize garbage collection routes and management processes for a fictitious garbage collection company. The primary focus of this project was to create an efficient route optimization algorithm, leveraging **Floyd-Warshall, Dijkstra** and **A*** pathfinding techniques to minimize travel time and operational costs.

The system comprises several integral components including a user-friendly interface, a robust database for tracking garbage collection data, and dynamic route generation algorithms. Throughout the development process, the project utilized object-oriented programming principles and standard C++ libraries to ensure modularity and maintainability.

Key functionalities include **real-time route optimization** based on garbage levels at various locations, **cost analysis** for each route considering factors like fuel, maintenance, and labor, and a comprehensive reporting feature that provides detailed insights into daily operations.

The development of the GMS faced several challenges, particularly in optimizing pathfinding algorithms to handle real-world variables and constraints efficiently. These challenges were addressed through iterative testing and refinement of the algorithms.

2.0 Introduction

2.1 Purpose of the Report

The purpose of this report is to provide a comprehensive analysis of the Garbage Management System developed as part of the academic coursework in computer science. It aims to detail the system's architecture, the technologies used, the challenges encountered, and the solutions implemented. This report serves as a documentation of the development process and evaluates the theoretical and practical aspects of the system.

2.2 Objectives of the System

The Garbage Management System was developed with the following objectives:

Optimize Collection Routes: To reduce the time and cost associated with waste collection by implementing efficient route planning algorithms.

Enhance Operational Efficiency: To streamline operations using automated scheduling and real-time adjustments based on varying waste levels.

Improve Environmental Impact: To contribute to environmental sustainability by reducing fuel consumption and optimizing logistics.

3.0 System Overview

3.1 Components of the System

The GMS is comprised of several key components, each responsible for a distinct aspect of the system's functionality:

GarbageLocation (garbageLocation.cpp/h): Manages details about each garbage collection point, including geographical coordinates, current waste levels, and adjacency relationships with other points. This component is pivotal for calculating distances and determining the feasibility of routes based on current traffic conditions and other dynamic factors.

Routing Components (aStarRoute.cpp/h, dijkstraRoute.cpp/h, optimizedRoute.cpp/h): Each of these components implements a specific algorithm for route optimization:

A **Route** Optimization (aStarRoute.cpp /h):* Utilizes the A* search algorithm to find the shortest path from the central depot to designated collection points considering heuristic estimates of distance.

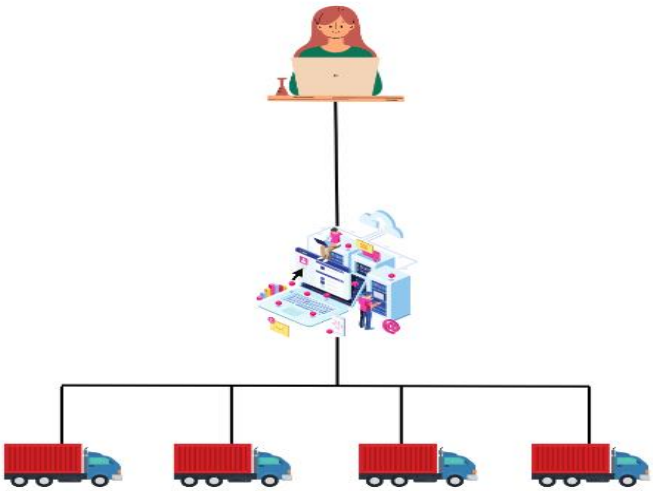
Dijkstra Route Optimization (dijkstraRoute.cpp/h): Implements Dijkstra's algorithm to compute the shortest paths from the source to all other points, used primarily for scenarios requiring comprehensive route coverage.

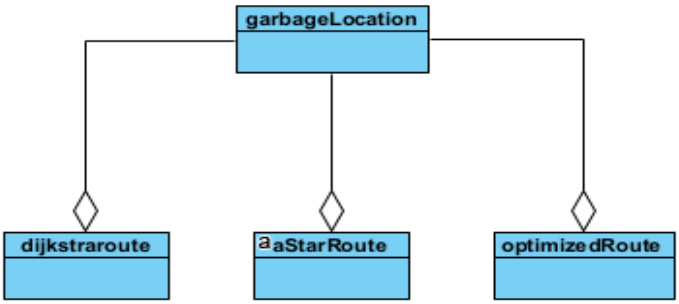
Floyd-Warshall Route Optimization (optimizedRoute.cpp/h): Applies the Floyd-Warshall algorithm for calculating the shortest paths between all pairs of points, useful for pre-computation and multi-stop route planning.

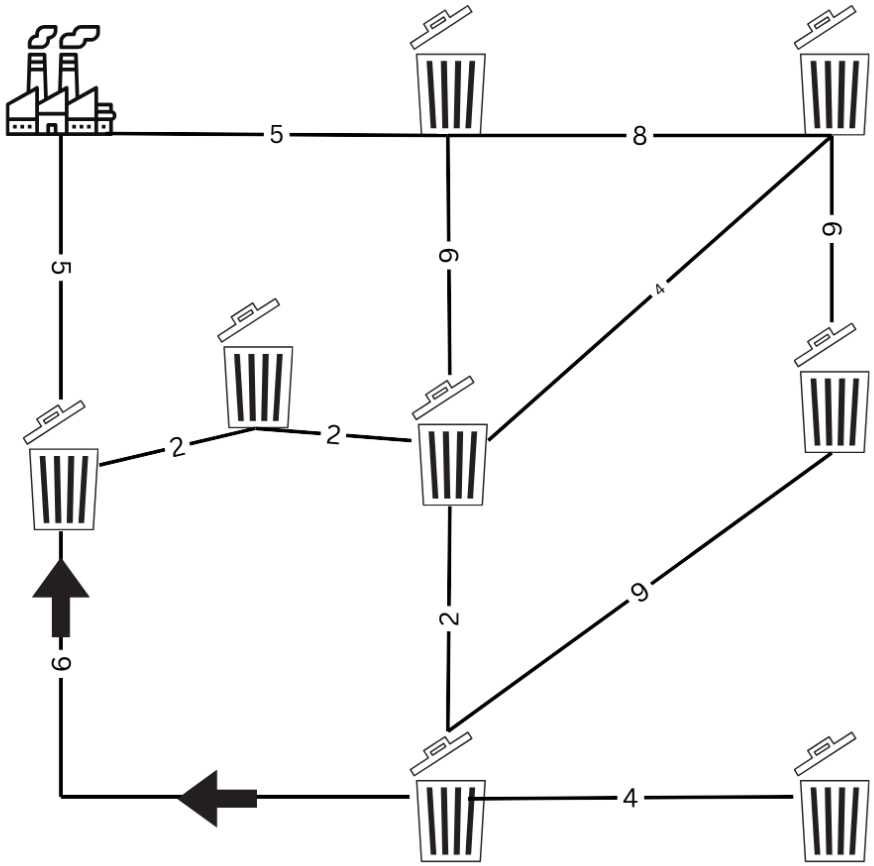
Cost Calculation (routeCost.h/cpp): Calculates the operational costs associated with each route, including fuel consumption, labor, and vehicle maintenance, ensuring the system not only optimizes for time but also for cost efficiency.

Main (main.cpp): Serves as the entry point of the system, initializing components, handling user inputs, and coordinating the flow of operations across the system.

3.2 Diagrams




garbageLocation
dijkstraroute
aStarRoute
optimizedRoute



4.0 Technical Description

4.1 Programming Environment and Tools Used

The **Garbage Management System (GMS)** was developed using Code::Blocks, an open-source Integrated Development Environment (IDE) that supports the C++ programming language. The system is compiled using the C++17 standard, which offers advanced language features that enhance the capability and efficiency of the developed software. The choice of C++ and Code::Blocks was driven by the need for a robust system with real-time capabilities and the extensive support for object-oriented programming features and standard libraries provided by C++.

4.2 Description of the C++ Program

4.2.1 Main Functions

The main function in the Garbage Management System serves as the central hub for user interaction and system control. It initializes the system, updates traffic conditions, and continuously prompts the user with menu options until the termination command is issued. Here's a breakdown of its critical components:

Initialization

The main function starts by initializing necessary data structures and setting up the system environment:

- **Location and Traffic Initialization:** Initializes a vector of garbageLocation objects with simulated waste levels and updates the traffic conditions randomly affecting the predefined routes.
- **Distance Matrix and Graph Setup:** Prepares a distance matrix from predefined values and converts this matrix into an adjacency list for use with graph-based routing algorithms.

```
    vector<int> affected = garbageLocation::randomizeAffectedLocations(2,
totalLocations);

    garbageLocation::updateTrafficConditions(affected);
    // Update traffic conditions based on weather and time
    vector<garbageLocation> locations =
garbageLocation::initialize_garbageLocation_vector();
```

Menu System and User Input Handling

The core of the main function is a loop that displays a menu and processes user input through nested switch statements. Each case within the switch corresponds to a different function of the system:

- **Top-Level Menu (menuPrint):** Displays the main menu options to the user, where the user's choice determines which part of the program to execute next.

```
    do {
        // Output affected locations for verification
        cout << "High-Traffic Locations Today:";
        for (int loc : affected) {
            cout << garbageLocation::indexToName(loc) << " ";
        }
        cout << "\n";
        cout << "\n";
        garbageLocation::printWasteLevels(locations);
        cout << "\n";
        menu.printMainMenu();
        cin >> option;
        switch(option)
```

Explanation of Switch Statements

The switch statements are structured to handle various functionalities based on user selections:

```
switch(option){
    case 1:{
        system("cls"); // Clear screen
        int mrpm;
        menu.printMultipleRouteMenu();
        cin >> mrpm;
```

```
        switch(mrpm){
            case 1:{
                system("cls");
                float dist[9][9];  // Distance matrix
                int next[9][9];    // Next hop matrix
                cout << "Floyd-Warshall | 60% Garbage Level | <40km" << endl;
                for (int i = 0; i < 9; ++i) {
                    for (int j = 0; j < 9; ++j) {
                        dist[i][j] = garbageLocation::distanceMatrix[i][j];
                        if (i != j && dist[i][j] != INF) {
                            next[i][j] = j;
                        } else {
                            next[i][j] = -1;
                        }
                    }
                }
                // Compute shortest paths using Floyd-Warshall algorithm
                OptimizedRoute::floydWarshall(dist, next);
                // Generate routes based on the Floyd-Warshall algorithm output
                OptimizedRoute::generatefloydWarshall(locations, dist, next);
                cout << "\n";
                if (!menu.printReturn()) {
                    option = false;
                }
                break;
            }
```

```
            case 2:{
                system("cls");
                float dist[9][9];
                int next[9][9];
                cout <<"Generated Djikstra | 40% Garbage Level | <20Km" << endl;
                for (int i = 0; i < 9; ++i) {
                    for (int j = 0; j < 9; ++j) {
                        dist[i][j] = garbageLocation::distanceMatrix[i][j];
                        if (i != j && dist[i][j] != INF) {
                            next[i][j] = j;
                        } else {
                            next[i][j] = -1;
                        }
                    }
                }
                // Create a graph from the distance matrix for Dijkstra's algorithm
                vector<vector<pair<int, float>>> graph = createGraph(dist, 9);
                // Generate routes using Dijkstra's algorithm
                DjikstraRoute::generateDijkstraRoute(locations, graph);
                cout << "\n";
                if (!menu.printReturn()) {
                    option = false;
                }
                break;
```

```
        case 2:{
            system("cls");
            // Generate routes using A* algorithm with a garbage threshold of 50%
            AStarRoute::generateRoutes(locations, 50);
            if (!menu.printReturn()) {
                option = false;
            }
            break;
        }
```

```
            case 3:{
                system("cls");
                // Print the graphical representation of the network
                garbageLocation::printGraph();
                cout << "\n";
                if (!menu.printReturn()) {
                    option = false;
                }
                break;
            }

        }
    } while(option != 9);
    cout << "Exiting..." << endl;  // Exit message
    return 0;
}
```

- **First Level Switch:**

- **Case 1 (Route Menu Generation):** Handles the selection of routing algorithms. It clears the screen and presents another menu specific to routing choices (Floyd-Warshall, Dijkstra). This is managed through another switch statement nested within.
- ***Case 2 (A* Route Generation):*** Directly triggers the generation of routes using the A* algorithm based on the current traffic and waste conditions.
- **Case 3 (Print Graph):** Displays a graphical representation of the locations and their connections, providing a visual map of the routes and distances.

- **Second Level Switch (Nested within Case 1):**
  - Case 1 (Floyd-Warshall Algorithm): Initializes distance and next-hop matrices, executes the Floyd-Warshall algorithm, and displays the results.
  - **Case 2 (Dijkstra Algorithm): Similar** setup to Floyd-Warshall but utilizes the Dijkstra algorithm to compute the shortest paths from a selected source to all destinations.

Control Flow and Function Invocation

Each case within the switch statements is designed to invoke specific functionalities:

- **Graphical User Interface:** Clears the screen to provide a clean view for each new output, enhancing user experience.
- **Dynamic Response to Input:** Each routing option reflects real-time data processing, accounting for any changes in traffic or waste levels.
- **Loop Continuation:** The loop continues to prompt the user until the exit option (often designated as an option like '9' or similar) is selected, at which point a message confirms the program's termination.

4.2.2 Utility and Helper Functions

Graph Creation (createGraph): A utility function that converts a distance matrix into an adjacency list format, facilitating the application of graph algorithms for route optimization. This function is critical for setting up the graph data structure required by the Dijkstra and A* algorithms.

Distance Matrix Initialization: Within main.cpp, the distance matrix is initialized based on predefined distances between locations, setting the stage for the application of the Floyd-Warshall algorithm for all-pairs shortest path computation.

4.2.3 Data Handling and Management

Dynamic Traffic Updates: The system dynamically updates traffic conditions which affect the route calculations. This is managed by **garbageLocation.cpp**, which adjusts the distance matrix based on random traffic conditions, simulating real-world variability in route planning.

Randomized Location Effects: Utilizes randomized functions to simulate variable waste levels at different locations, influencing the prioritization and scheduling of garbage collection routes.

4.3 Code Snippets and Commentary

4.3.1 Detailed Code Explanations

4.3.1.1 Source Files

aStarRoute.cpp		
Component	**Purpose**	**Implementation Details**
Node Comparison Operator	Defines a custom comparison operator for Node objects in the priority queue.	Overloads the greater-than operator (**operator>()**) to compare nodes based on **fCost** (gCost + heuristic). Returns true if the **fCost** of the current node is greater than that of another node.
Heuristic Function	Estimates the cost to reach the goal from a node.	Uses a simple absolute difference between the indices of the current and goal locations to estimate the cost.
Path Distance Calculation	Calculates the total travel distance for a path.	Iterates through nodes in a path, summing distances between consecutive nodes using **garbageLocation::calculateDistance**, which retrieves distances from a predefined matrix.
AStarRoute::aStar Function	Implements the A* algorithm to find the shortest path between two points.	Starts with the initial node's cost set to zero, pushes it onto a priority queue, processes nodes by lowest **fCost**, updates costs and predecessors for neighbors, reconstructs path from goal to start.
Route Generation	Generates routes connecting multiple garbage collection points based on their waste levels.	Identifies locations with waste levels above a threshold, finds shortest paths sequentially from one point to the next, stops if max distance is exceeded, includes return path to start.
File Output	Writes route details and associated costs to a file for documentation and analysis.	Outputs detailed route segments and total costs, ensuring data is saved persistently.

```
bool AStarRoute::Node::operator>(const Node& other) const {
    return this->fCost > other.fCost;
}

float AStarRoute::heuristic(int from, int to) {
    return std::abs(from - to);
}
float AStarRoute::pathDistance(const std::vector<int>& path, const std::vector<garbageLocation>& locations) {
    float totalDistance = 0.0;
    for (size_t i = 1; i < path.size(); i++) {
        totalDistance += garbageLocation::calculateDistance(path[i-1], path[i]);
    }
    return totalDistance;
}
```

```cpp
std::vector<int> AStarRoute::aStar(int startIdx, int goalIdx, const std::vector<garbageLocation>& locations) {
    std::priority_queue<Node, std::vector<Node>, std::greater<Node>> openSet;
    std::vector<float> gCosts(locations.size(), FLT_MAX);
    std::vector<int> cameFrom(locations.size(), -1);
    std::vector<bool> closedSet(locations.size(), false);

    gCosts[startIdx] = 0;
    openSet.push({startIdx, 0, heuristic(startIdx, goalIdx)});

    while (!openSet.empty()) {
        Node current = openSet.top();
        openSet.pop();

        if (current.index == goalIdx) {
            std::vector<int> path;
            for (int at = goalIdx; at != startIdx; at = cameFrom[at]) {
                path.push_back(at);
            }
            path.push_back(startIdx);
            std::reverse(path.begin(), path.end());
            return path;
        }

        closedSet[current.index] = true;

        for (int i = 0; i < locations.size(); i++) {
            if (closedSet[i] || garbageLocation::calculateDistance(current.index, i) == INF) continue;

            float tentativeGCost = current.gCost + garbageLocation::calculateDistance(current.index, i);
            if (tentativeGCost >= gCosts[i]) continue;

            cameFrom[i] = current.index;
            gCosts[i] = tentativeGCost;
            float fCost = tentativeGCost + heuristic(i, goalIdx);
            openSet.push({i, tentativeGCost, fCost});
        }
    }

    return {}; // return empty path if no path is found
}
```

```cpp
void AStarRoute::generateRoutes(const std::vector<garbageLocation>& locations, float maxTotalDistance) {
    std::ofstream outFile("AStarRoute.txt", std::ios::app);  // File to write the routes and costs

    std::vector<int> eligibleLocations;
    const int startIdx = 0; // Assuming 'S' is always at index 0
    routeCost::RouteCost costs; // Create an instance of RouteCost to track the costs

    // Identify eligible locations
    for (int i = 1; i < locations.size(); i++) {
        if (locations[i].wasteLevel > 50) {
            eligibleLocations.push_back(i);
        }
    }

    // Attempt to create a path that visits all eligible locations
    std::vector<int> tour;
    std::unordered_set<int> visited;
    float currentDistance = 0;
    int currentLocation = startIdx;

    while (!eligibleLocations.empty()) {
        float minDistance = FLT_MAX;
        int nextLocation = -1;
        std::vector<int> path;

        // Find the nearest unvisited eligible location using a modified A* that considers already visited nodes
        for (int loc : eligibleLocations) {
            std::vector<int> tempPath = AStarRoute::aStar(currentLocation, loc, locations);
            float pathDistance = AStarRoute::pathDistance(tempPath, locations);
            if (pathDistance < minDistance && currentDistance + pathDistance < maxTotalDistance) {
```

```
        currentLocation = nextLocation;
        currentDistance += minDistance;
        costs.updateCosts(minDistance);
        visited.insert(nextLocation);
        costs.setSegmentDistance(currentDistance);
        eligibleLocations.erase(std::remove(eligibleLocations.begin(), eligibleLocations.end(), nextLocation), eligibleLocations.end());
    }

    // Add return to start if possible and update costs
    std::vector<int> returnPath = AStarRoute::aStar(currentLocation, startIdx, locations);
    float returnDistance = AStarRoute::pathDistance(returnPath, locations);
    if (currentDistance + returnDistance <= maxTotalDistance) {
        returnPath.erase(returnPath.begin()); // Remove the duplicate start node
        tour.insert(tour.end(), returnPath.begin(), returnPath.end());
        currentDistance += returnDistance;
        costs.updateCosts(returnDistance);
    } else {
        outFile << "Cannot return to start within distance limit." << std::endl;
        std::cout << "Cannot return to start within distance limit." << std::endl;
    }

    // Write and print the route information
    outFile << "--->50%---|" << "A* Route" << "|---" << maxTotalDistance << "Km---" << std::endl;
    outFile << "Route: ";
    std::cout << "--->50%---|" << "A* Route" << "|---" << maxTotalDistance << "Km---" << std::endl;
    std::cout << "Route: ";
    for (int i = 0; i < tour.size(); i++) {
        if (i > 0) {
            outFile << " -> ";
            std::cout << " -> ";
        }
        outFile << locations[tour[i]].garbageLocation_name;
        std::cout << locations[tour[i]].garbageLocation_name;
    }
    outFile << std::endl;
    std::cout << std::endl;

    // Print the total costs for the route
    costs.printCosts( outFile);
    outFile.close();  // Close the file after writing all details
}
```

djikstraRoute.cpp		
Function	**Purpose**	**Key Operations and Logic**
generateDijkstraRoute	Generate routes between a central depot and multiple garbage locations using Dijkstra's algorithm.	1. Initialize distance and predecessor vectors. 2. Execute Dijkstra's algorithm from the central depot to all nodes. 3. For each node meeting certain waste and distance criteria, calculate the round trip route and costs. 4. Output route details to both the console and a file.
dijkstra	Implements Dijkstra's algorithm to find the shortest path from a single source to all other vertices.	1. Initialize all distances as infinite, except for the source node. 2. Use a priority queue to process nodes in order to increase distance. 3. Update distances and predecessors if a shorter path is found.

		4. Continue until all nodes are processed or no shorter paths are found.
printPath	Prints the optimal path from the source to a target node and calculates costs.	1. Check for a valid path using the predecessor vector. 2. Construct the path from the target back to the source using the predecessor data. 3. Print the path and update transportation costs along the way using the distances between successive nodes. 4. Display the route using the names of the locations at each node in the path.

```cpp
// Generate djikstra Route
void DjikstraRoute::generateDijkstraRoute(const std::vector<garbageLocation>& locations, const std::vector<std::vector<std::pair<int, float>>>& graph) {
    int source = 0;  // Usually the central depot or main station

    for (size_t i = 1; i < locations.size(); i++) {
        // Run Dijkstra from the source to all other nodes to find shortest path to i
        std::vector<float> distOutbound(locations.size(), std::numeric_limits<float>::max());
        std::vector<int> prevOutbound(locations.size(), -1);
        dijkstra(graph, source, distOutbound, prevOutbound);

        // Check if collection is needed and the destination is within allowable distance
        if (locations[i].wasteLevel > 40 && distOutbound[i] < 20) {

            // Initialize a new RouteCost object to track total costs for the round trip
            routeCost::RouteCost cost;

            // Specifically set the segment distance from S to location i
            cost.setSegmentDistance(distOutbound[i]);

            std::cout << "-------*" << "Location " << locations[i].garbageLocation_name << "*-------" << std::endl;
            std::cout << "Route to location:\n";
            printPath(i, prevOutbound, locations, cost, distOutbound);
            std::cout << "\n";

            // Run Dijkstra from location i back to the source for the return journey
            std::vector<float> distReturn(locations.size(), std::numeric_limits<float>::max());
            std::vector<int> prevReturn(locations.size(), -1);
            dijkstra(graph, i, distReturn, prevReturn);

            std::cout << "Return route:\n";
            printPath(source, prevReturn, locations, cost, distReturn);
            std::cout << std::endl;

            std::ofstream outFile("DijkstraRoute.txt", std::ios::app);
            // Print the total costs for this round trip
            cost.printCosts(outFile);
            std::cout << std::endl; // Separate different location reports with a newline
        }
    }
}
```

```cpp
// Implements Dijkstra's algorithm to compute the shortest paths from a source node to all other nodes in a graph.
void DjikstraRoute::dijkstra(const std::vector<std::vector<std::pair<int, float>>>& graph, int src, std::vector<float>& dist, std::vector<int>& prev) {
    int n = graph.size(); // Number of nodes in the graph
    dist.assign(n, INF); // Initialize the distance to each node as infinity
    prev.assign(n, -1); // Initialize the previous node for each node as -1 (undefined)
    dist[src] = 0; // Set the distance from the source to itself to zero

    // Priority queue to store the nodes to explore, sorted by distance from the source
    std::priority_queue<std::pair<float, int>, std::vector<std::pair<float, int>>, std::greater<std::pair<float, int>>> pq;
    pq.push({0, src}); // Push the source node onto the queue with distance 0

    // Loop until there are no more nodes to process
    while (!pq.empty()) {
        auto [d, u] = pq.top(); // Get the node with the smallest distance
        pq.pop(); // Remove this node from the queue

        if (d > dist[u]) // If the recorded distance is greater than the current distance, skip processing
            continue;

        // Iterate through each adjacent node
        for (auto& edge : graph[u]) {
            int v = edge.first; // Node at the end of the edge
            float weight = edge.second; // Weight of the edge
            if (dist[u] + weight < dist[v]) { // If the new distance to node v is shorter, update it
                dist[v] = dist[u] + weight;
                prev[v] = u; // Record the previous node
                pq.push({dist[v], v}); // Push the updated distance to the priority queue
            }
        }
    }
}
```

```cpp
// Prints the optimal path from the source to a target node and calculates the transportation costs based on distances.
void DjikstraRoute::printPath(int target, const std::vector<int>& prev, const std::vector<garbageLocation>& locations, routeCost::RouteCost& cost, const std::vector<float>& dist) {
    if (prev[target] == -1) {
        std::cout << " No path";
        return;
    }
    std::vector<int> path;
    for (int v = target; v != -1; v = prev[v]) {
        path.push_back(v);
    }
    reverse(path.begin(), path.end());
    for (size_t i = 0; i < path.size(); i++) {
        if (i > 0) std::cout << " -> ";
        std::cout << locations[path[i]].garbageLocation_name;
        if (i > 0) {
            cost.updateCosts(dist[path[i]] - dist[path[i - 1]]);
        }
    }
}
```

garbageLocation.cpp		
Function	**Purpose**	**Key Operations and Logic**
printGraph	Outputs a visual representation of the garbage collection graph.	- Prints a multi-line string representation of nodes and edges in the graph, with labels indicating distances.
garbageLocation (constructor)	Constructs a garbageLocation object with a specified name and initializes its waste level.	- Sets the garbageLocation_name. - Initializes wasteLevel randomly for all locations except the station ('S'), which has a waste level of 0.
initialize_garbageLocation_vector	Initializes and returns a vector of garbageLocation objects representing different locations.	- Seeds the random number generator. - Creates and returns a vector containing garbageLocation objects for each location named from "S" to "G".
printWasteLevels	Prints the waste level for each location in a vector of garbageLocation objects.	- Iterates through the vector of locations and prints the garbageLocation_name along with its wasteLevel.

calculateDistance	Calculates and returns the distance between two specified locations using a distance matrix.	- Accesses the static distance matrix to get the distance between two indices. - Returns the distance or indicates infinity (INF) if no direct path exists.
indexToName	Converts an index to the corresponding location name.	- Uses a static array of strings to map indices to location names. - Returns the name if the index is within range; otherwise, returns "Invalid index".
updateTrafficConditions	Updates traffic conditions by applying a multiplier to the distances between specified locations.	- Applies a traffic multiplier to the distances in the distance matrix for specified locations to simulate traffic conditions affecting travel times.
randomizeAffectedLocations	Randomly selects a specified number of locations from a total set.	- Initializes a list of location indices and randomly rearranges them using a uniform distribution. - Resizes the list to include only the specified number of locations, effectively selecting a random subset of locations to be considered affected.

```cpp
void garbageLocation::printGraph() {
    std::string line1  = "S-----5-----A---8---B ";
    std::string line2  = "|           |      /| ";
    std::string line3  = "|           |     / 6 ";
    std::string line4  = "|           |    /  | ";
    std::string line5  = "5           6   4   G ";
    std::string line6  = "|           |  /   /  ";
    std::string line7  = "|           | /   /   ";
    std::string line8  = "|           |/   /    ";
    std::string line9  = "D-->2-E->2--C   9     ";
    std::string line10 = "^           |  /      ";
    std::string line11 = "|           2 /       ";
    std::string line12 = "^           |/        ";
    std::string line13 = "6-<-<-<-<-<-F         ";

    // Print each line
    std::cout << line1 << std::endl;
    std::cout << line2 << std::endl;
    std::cout << line3 << std::endl;
    std::cout << line4 << std::endl;
    std::cout << line5 << std::endl;
    std::cout << line6 << std::endl;
    std::cout << line7 << std::endl;
    std::cout << line8 << std::endl;
    std::cout << line9 << std::endl;
    std::cout << line10 << std::endl;
    std::cout << line11 << std::endl;
    std::cout << line12 << std::endl;
    std::cout << line13 << std::endl<< std::endl;
}
garbageLocation::garbageLocation(std::string name) : garbageLocation_name(name) {
    // Implementation
    if (name == "S") {
        wasteLevel = 0;
    } else {
        wasteLevel = (std::round(rand() % 100));
    }
}
```

```
float garbageLocation::distanceMatrix[9][9] = {
    //S   A    B    C    D    E    F    G    H
    {0,   5,  INF, INF,  5 , INF, INF, INF, INF }, // S
    {5,   0,   8 ,  6 , INF, INF, INF, INF, INF }, // A
    {INF, 8,   0 ,  4 , INF, INF, INF,  6 , INF }, // B
    {INF, 6,   4 ,  0 , INF, INF,  2 , INF, INF }, // C
    {5 , INF, INF, INF,  0 ,  2,  INF, INF, INF }, // D
    {INF,INF, INF,  2,  INF,  0,  INF, INF, INF }, // E
    {INF,INF, INF,  2 ,  6,  INF,  0,  9 ,   4  }, // F
    {INF,INF,  6,  INF, INF, INF,  9,  0 ,  INF }, // G
    {INF,INF, INF, INF, INF, INF,  4, INF,   0  }, // H
};
```

```
std::vector<garbageLocation> garbageLocation::initialize_garbageLocation_vector() {
    srand(time(NULL)); // Initialize random seed
    std::vector<garbageLocation> garbageLocations;
    garbageLocations.push_back(garbageLocation("S")); // Assuming S is the station
    garbageLocations.push_back(garbageLocation("A"));
    garbageLocations.push_back(garbageLocation("B"));
    garbageLocations.push_back(garbageLocation("C"));
    garbageLocations.push_back(garbageLocation("D"));
    garbageLocations.push_back(garbageLocation("E"));
    garbageLocations.push_back(garbageLocation("F"));
    garbageLocations.push_back(garbageLocation("G"));

    return garbageLocations;
}
```

```cpp
void garbageLocation::printWasteLevels(const std::vector<garbageLocation>& garbageLocations) {
    for (const auto& location : garbageLocations) {
        std::cout << "Location " << location.garbageLocation_name << " has garbage level: " << location.wasteLevel << std::endl;
    }
}
float garbageLocation::calculateDistance(int from, int to) {
    float baseDistance = garbageLocation::distanceMatrix[from][to];
    if (baseDistance == INF) return INF;
    return baseDistance;
}
// In garbageLocation.cpp or where you implement the methods
std::string garbageLocation::indexToName(int index) {
    static const std::string names[9] = {"S", "A", "B", "C", "D", "E", "F", "G", "H"};
    if (index >= 0 && index < 9) {
        return names[index];
    }
    return "Invalid index";
}

void garbageLocation::updateTrafficConditions(const std::vector<int>& locations) {
    float trafficMultiplier = 1.25f;
    for (int from : locations) {
        for (int to = 0; to < 9; ++to) {
            if (from != to && distanceMatrix[from][to] != INF) {
                distanceMatrix[from][to] *= trafficMultiplier;
            }
        }
    }
}
std::vector<int> garbageLocation::randomizeAffectedLocations(int numLocations, int totalLocations) {
    std::vector<int> locations(totalLocations);
    std::iota(locations.begin(), locations.end(), 0);

    std::random_device rd;
    std::mt19937 g(rd());

    for (int i = 0; i < numLocations; ++i) {
        std::uniform_int_distribution<> dis(i, totalLocations - 1);
        int j = dis(g);
        std::swap(locations[i], locations[j]);
    }

    locations.resize(numLocations);

    return locations;
}
```

optimizedRoute.cpp		
Function	**Purpose**	**Key Operations and Logic**
floydWarshall	Implements the Floyd-Warshall algorithm to calculate the shortest paths between all pairs of nodes in a graph.	- Iterates through all possible pairs of nodes and updates the shortest path distances between them using an intermediate node if it offers a shorter path. - Updates the path reconstruction matrix 'next' to store the intermediate nodes for path tracing.
printPath	Outputs the shortest path from a specified start node to an end node based on the 'next' matrix.	- Starts at the beginning node and follows the 'next' matrix to trace the path to the destination, outputting each location's name. - Continues until the destination node is reached, concatenating location names to form the complete path.

printAndCalculatePath	Outputs the shortest path and calculates transportation costs based on distances for routing optimization.	- Like printPath, but additionally updates transportation costs for each segment of the journey using the distance between nodes. - Outputs the path to both the console and a file. - Calls the method to update transportation costs with each traveled segment, ensuring cost calculations are accurate for the entire route.
generatefloydWarshall	Generates optimized routes using the Floyd-Warshall algorithm for locations based on waste levels and distances.	- Iterates over each location, checking if it meets specific waste level and distance criteria. - For eligible locations, calculates and prints the route from the central depot to the location and back, tracking costs. - Outputs detailed route and cost information to a file and console, closing the file after all entries.

```cpp
// Implementation of the Floyd-Warshall algorithm to find the shortest paths between all pairs of vertices
void OptimizedRoute::floydWarshall(float dist[][9], int next[][9]) {
    const int V = 9; // Number of vertices in the graph
    // Loop through each combination of vertices as intermediate, start, and end points
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                // Update the shortest path and the intermediate vertex if a shorter path is found
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    next[i][j] = next[i][k];
                }
            }
        }
    }
}
```

```cpp
// Prints and calculates the optimal path from 'start' to 'end' using precomputed information
void OptimizedRoute::printAndCalculatePath(int next[][9], int start, int end, const float dist[][9], routeCost::RouteCost& cost, std::ofstream& outFile) {
    if (next[start][end] == -1) {
        std::cout << " No path";  // Indicate no path exists
        outFile << " No path";
        return;
    }
    int u = start;
    std::cout << "S";  // Start from the 'start' location
    outFile << "S";
    while (u != end) {
        int v = next[u][end];
        // Output each step in the path
        std::cout << " -> " << garbageLocation::indexToName(v);
        outFile << " -> " << garbageLocation::indexToName(v);
        // Update cost calculations for this segment
        cost.updateCosts(dist[u][v]);
        u = v;
    }
}
```

```cpp
// Generates route with floydWarshall
void OptimizedRoute::generatefloydWarshall(const std::vector<garbageLocation>& locations, const float dist[][9], int next[][9]) {
    std::ofstream outFile("FloydWarshallRoute.txt", std::ios::app);
    for (size_t i = 1; i < locations.size(); ++i) {
        if (locations[i].wasteLevel > 60 && dist[0][i] < 40) {  // Check waste level and distance criteria
            std::cout << "-------*" << "Location " << locations[i].garbageLocation_name << "*-------" << std::endl;
            outFile << "-------*" << "Location " << locations[i].garbageLocation_name << "*-------" << std::endl;

            // Initialize a new RouteCost object to track total costs for the round trip
            routeCost::RouteCost cost;

            // Specifically set the segment distance from S to the initial location
            cost.setSegmentDistance(dist[0][i]);

            // Calculate and print the path from S to location i
            std::cout << "Start: \n";
            outFile << "Start: \n";
            printAndCalculatePath(next, 0, i, dist, cost, outFile);
            std::cout << std::endl;
            outFile << std::endl;

            // Calculate and print the return path from location i to S
            std::cout << "Return\n";
            outFile << "Return\n";
            printAndCalculatePath(next, i, 0, dist, cost, outFile);
            std::cout << std::endl;
            outFile << std::endl;

            // Print the total costs after both trips are completed
            cost.printCosts(outFile);
            std::cout << "\n";
            outFile << "\n";
        }
    }
    outFile.close(); // Close the file after writing all details
}
```

4.3.1.2 Header Folder

Header files play a crucial role in C++ projects by declaring the interfaces of classes and functions. This separation of declaration from implementation helps manage large codebases by making them easier to navigate and maintain. Below, I describe the specific purposes and contents of each header file involved in the waste management routing system:

1. AStarRoute.h

Purpose: Declares the AStarRoute class, which is responsible for implementing the A* algorithm, a popular pathfinding and graph traversal method. This class finds the most cost-effective route between two nodes. Node struct with cost properties and comparison operator.

Methods for executing the A* algorithm, generating routes, and calculating path distances.

2. DjikstraRoute.h

Purpose: Defines the DjikstraRoute class for applying Dijkstra's algorithm to find the shortest paths from a single source to all other vertices in a graph. Static methods for running the algorithm, printing paths, and generating routes tailored to garbage collection locations.

3. GarbageLocation.h

Purpose: Establishes the garbageLocation class to model individual collection points with attributes like waste levels and methods for handling distances and traffic updates. Attributes for location name, waste level, and visitation. Methods for initializing location data, calculating distances, and printing garbage levels.

4. MenuPrint.h

Purpose: Provides the MenuPrint class which handles all user interface components related to menu selections and navigation within the application. Methods for printing different menus (main, multiple route generation, single route generation) and handling return navigation.

5. OptimizedRoute.h

Purpose: This header file defines the OptimizedRoute class, which is primarily designed to handle the computation of optimized routes using the Floyd-Warshall algorithm. This algorithm finds the shortest paths between all pairs of nodes in a weighted graph, which is essential for planning efficient routes in a network of garbage collection locations. Static methods for executing the Floyd-Warshall algorithm (floydWarshall) to compute the shortest paths matrix. Methods like printPath and printAndCalculatePath to display routes and calculate associated costs based on the path distances. A method (generatefloydWarshall) to generate optimized routes based on specific criteria such as distance thresholds and waste levels at different locations..

6. RouteCost.h

Purpose: Defines the routeCost namespace and its nested RouteCost struct, which are used to manage and calculate various costs associated with the routes, such as fuel costs, maintenance costs, and labor costs. This cost management is critical for optimizing and evaluating the economic efficiency of garbage collection routes. The RouteCost struct encapsulates various cost metrics and provides methods to update these costs based on travel distances and times. Methods to calculate costs for each segment of a journey (updateCosts), set and get segment-specific distances, and handle additional costs incurred during collection stops. A method (printCosts) to output detailed cost information to both the console and a file, ensuring that route costs are transparent and easily reportable.

5.0 Data Structures and Algorithms

5.1 Overview of Data Structures Used

n the routing system, various data structures are employed to efficiently manage and process the data associated with the network of garbage collection points. The choice of data structures is critical to supporting the functionality and performance of the pathfinding and optimization algorithms used.

- **Graphs:** The graph is the fundamental data structure used to represent the collection points and pathways between them, with nodes symbolizing locations and edges denoting the accessible routes between these locations. This representation is crucial for both pathfinding and optimization algorithms.
- **Priority Queues:** Utilized primarily in Dijkstra's algorithm, priority queues help manage the nodes to be processed. Nodes are prioritized based on the shortest distance from the source, enabling efficient determination of the next node to process.
- **Arrays and 2D Arrays:** These are used extensively to store distances between locations. In the Floyd-Warshall algorithm, a 2D array holds the shortest distances between all pairs of nodes, facilitating quick updates and path reconstruction.
- **Vectors:** Serve as dynamic arrays to manage lists of nodes, distances, and other pertinent data. Vectors are especially useful due to their dynamic sizing capabilities, which accommodate the variable number of locations and route calculations.

5.2 Algorithms Description

5.2.1 Pathfinding Algorithms (Dijkstra, A*)

- **Dijkstra's Algorithm:** This algorithm is instrumental in finding the shortest path from a single source node to all other nodes. It is particularly useful in scenarios where the shortest paths need to be recalculated frequently due to dynamic conditions such as traffic or road closures. The algorithm efficiently handles these updates by recalculating paths from the source using a priority queue to always extend the shortest path found so far.
- *A* Algorithm:* The A* algorithm enhances the basic pathfinding approach by using heuristics to estimate the cost from the current node to the goal, reducing the number of explorations needed. This project employs a heuristic based on the geometric distance (e.g., Euclidean or Manhattan distance), appropriate for estimating distances in urban layouts where garbage collection routes are plotted.

5.2.2 Optimization Techniques

- **Floyd-Warshall Algorithm:** Used for its ability to compute the shortest paths between all pairs of nodes within a weighted graph. This capability is essential for precomputing and storing route distances in a central depot, allowing quick adjustments and decision-making for route optimization.
- **Traffic Adjustment Techniques:** The system incorporates dynamic adjustments to route calculations based on traffic conditions. By altering the weights of edges in the graph (representing the routes between nodes), the system can adapt to real-time traffic data, optimizing routes to avoid delays and reduce operational costs.

6.0 Evaluation

6.1 Results

6.1.2 Main Menu

```
High-Traffic Locations Today:E F

Location S has garbage level: 0
Location A has garbage level: 32
Location B has garbage level: 67
Location C has garbage level: 4
Location D has garbage level: 87
Location E has garbage level: 17
Location F has garbage level: 51
Location G has garbage level: 69

*-*-*-**-*-*-**-*-*-**-*-*-**-*-*-**-*-*-**-*-*-**-*-*-**-*
*           Please enter a command:                *
*--------------------------------------------------*   *
*      ( 1 ) Multiple Route Generation             * * * *
*--------------------------------------------------* M * A
*      ( 2 ) Single Route Generation               *   *
*--------------------------------------------------* R * Z
*      ( 3 ) Print Map                             * * * *
*--------------------------------------------------*   *
*      ( 9 ) Exit                                  *
*-*-*-**-*-*-**-*-*-**-*-*-**-*-*-**-*-*-**-*-*-**-*-*-**-*
```

6.1.3 Multiple-route menu

```
*-*-*-**-*-*-**-*-*-**-*-*-**-*-*-**-*-*-**-*-*-**-*-*-**-*
*            Please enter a command:                 *
*----------------------------------------------------*   *
*      ( 1 ) Floyd-Warshall Rotue                    * M   A
*----------------------------------------------------*   *
*      ( 2 ) Djikstra Route                          *   *
*----------------------------------------------------* R   Z
*      ( 9 ) Return to Main Menu                     * *   *
*----------------------------------------------------*   *
```

6.1.4 Floyd-Warshall Route

```
Floyd-Warshall | 60% Garbage Level | <40km
-------*Location B*-------
Start:
S -> A -> B
Return
S -> A -> S
Distance from Station: 13 km
Total Distance Inc Return: 26 km
Total Fuel Cost: $8.67
Total Maintenance Cost: $18.20
Total Labor Cost: $6.50
Total Time: 1.30 hours

-------*Location D*-------
Start:
S -> D
Return
S -> S
Distance from Station: 5.00 km
Total Distance Inc Return: 10.00 km
Total Fuel Cost: $3.33
Total Maintenance Cost: $7.00
Total Labor Cost: $2.50
Total Time: 0.50 hours

-------*Location G*-------
Start:
S -> A -> B -> G
Return
S -> B -> A -> S
Distance from Station: 19.00 km
Total Distance Inc Return: 38.00 km
Total Fuel Cost: $12.67
Total Maintenance Cost: $26.60
Total Labor Cost: $9.50
Total Time: 1.90 hours


*      ( Y ) Return to Main Menu                  *
*-------------------------------------------------*
*      ( N ) Exit                                 *
```

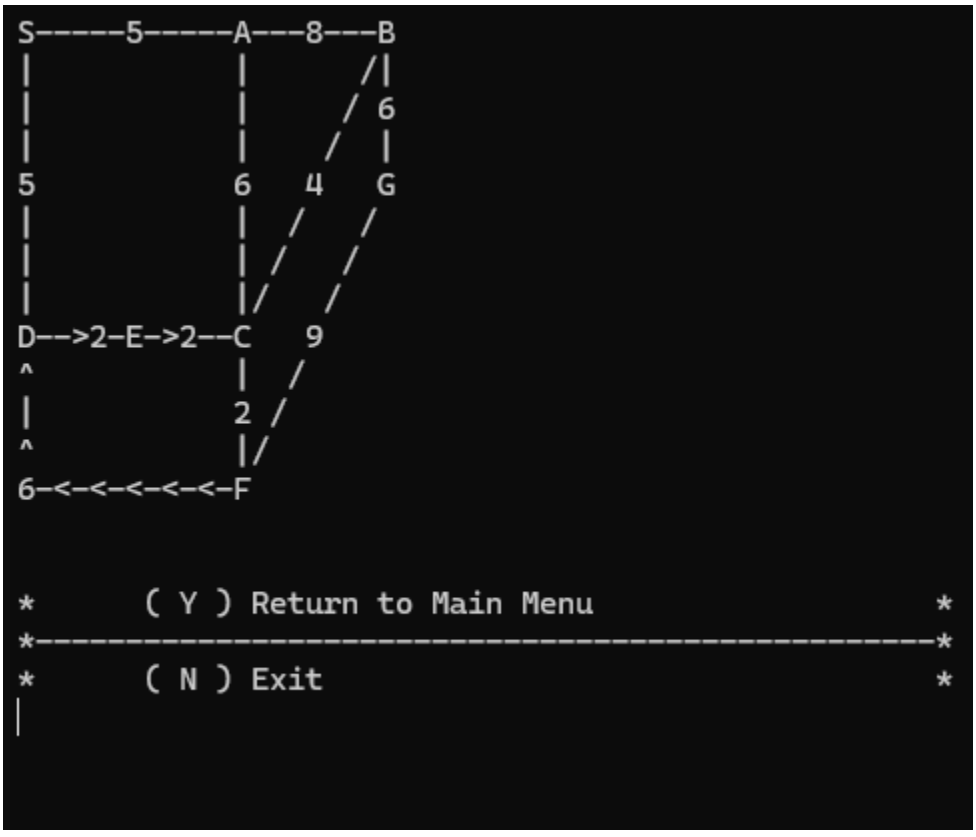
6.1.5 Dijkstra Route

```
Generated Djikstra | 40% Garbage Level | <20Km
-------*Location B*-------
Route to location:
S -> A -> B
Return route:
B -> A -> S
Distance from Station: 13.00 km
Total Distance Inc Return: 26.00 km
Total Fuel Cost: $8.67
Total Maintenance Cost: $18.20
Total Labor Cost: $6.50
Total Time: 1.30 hours

-------*Location D*-------
Route to location:
S -> D
Return route:
D -> S
Distance from Station: 5.00 km
Total Distance Inc Return: 10.00 km
Total Fuel Cost: $3.33
Total Maintenance Cost: $7.00
Total Labor Cost: $2.50
Total Time: 0.50 hours

-------*Location F*-------
Route to location:
S -> D -> E -> C -> F
Return route:
F -> D -> S
Distance from Station: 11.50 km
Total Distance Inc Return: 24.00 km
Total Fuel Cost: $8.00
Total Maintenance Cost: $16.80
Total Labor Cost: $6.00
Total Time: 1.20 hours

-------*Location G*-------
Route to location:
S -> A -> B -> G
Return route:
G -> B -> A -> S
Distance from Station: 19.00 km
Total Distance Inc Return: 38.00 km
Total Fuel Cost: $12.67
Total Maintenance Cost: $26.60
Total Labor Cost: $9.50
Total Time: 1.90 hours


*      ( Y ) Return to Main Menu                  *
*-------------------------------------------------*
*      ( N ) Exit                                 *
```

6.1.6 A* Single Route

```
--->50%---|A* Route|---50.00Km---
Route: S -> D -> E -> C -> F -> C -> B -> G -> B -> A -> S
Distance from Station: 24.00 km
Total Distance Inc Return: 43.00 km
Total Fuel Cost: $14.33
Total Maintenance Cost: $30.10
Total Labor Cost: $10.75
Total Time: 2.15 hours
*      ( Y ) Return to Main Menu                *
*-----------------------------------------------*
*      ( N ) Exit                               *
```

6.1.7 Print Map



7.0 Personal Reflections

7.1 Practical Experiences and Challenges

Throughout this project, I faced a variety of practical challenges, particularly in integrating real-time traffic data into the routing algorithms. This required not only technical adjustments within the algorithms but also a deep understanding of how traffic patterns can dynamically affect routing decisions. Handling the

complexities of data structures like graphs and priority queues in real scenarios tested my problem-solving skills extensively.

7.2 Theoretical Insights and Application

This project provided a rich opportunity to apply theoretical concepts from computer science, especially graph theory and algorithms, to a tangible problem. I discovered firsthand how algorithms like Dijkstra's and A* can be adapted for specific applications like route optimization, offering insights into their practical efficiency and limitations in real-world applications.

7.3 Learning Outcomes

One of the key learning outcomes of this project was understanding the importance of algorithm efficiency in systems that require real-time processing. I also learned to evaluate and choose the appropriate data structures based on the specific needs and constraints of the application.

7.4 Skills Acquired

Throughout the project, I honed my technical skills in programming with C++, especially in complex data structure manipulation and algorithm optimization. Additionally, I developed a stronger capability in debugging and refining code, which are crucial skills in software development.

7.5 Personal Growth and Development

This project was instrumental in enhancing my resilience and adaptability. Facing and overcoming the various challenges throughout the project helped me develop a more methodical approach to problem-solving and improved my ability to function effectively under pressure, preparing me for future professional challenges.