COMP2034

Supply Chain Distribution and Transportation System

Tang, Liqi

20259468

11th April 2023

I. Explanation of the C++ program coding

a. Mian.cpp:

```
#include "Generate.h"
#include "Comparison.h"

int main() {
  /* Time function returns the time since the
   Epoch(jan 1 1970). Returned time is in seconds. */
  time_t start, end;
  /* You can call it like this : start = time(NULL);
   in both the way start to contain the total time in seconds
   since the Epoch. */
  time(&start);
  // Seed the random number generator
  srand(time(nullptr));
  // Create a vector of RetailerSite objects
  vector<RetailerSite> sites;
  // Populate the vector with randomly generated RetailerSite objects
  generateRetailerSites(sites);
  // Display the information for each RetailerSite in the vector
  DisplayRetailerSites(sites);
  // Create instances of the NonOptimizedRoute, OptimizedRoute, and Greedy classes
  NonOptimizedRoute non = *new NonOptimizedRoute;
  OptimizedRoute Opt = *new OptimizedRoute;
  Greedy gre = *new Greedy;
  // vector to store retailer sites that meet the weight requirement
  vector<RetailerSite> meetReq;
  // Declare variables to store the calculated distance, weight, cost, and profit
  double distance;
  double weight;
  // Declare variable show the best route and highest profit
  string bestRoute;
  // Declare a variable to store the user's choice of route type
  int i;
  string continuenum = "1";
  DisplayRetailers(sites);
  do {    //Use DO-WHILE loop to control the continuation of the program
    // Prompt the user to choose a type of route
    cout<< "What kind of routes do you want to choose?"<< endl;
    cout<< "1. Best Route with Highest Profit"<<endl;
    cout<< "2. Non-Optimized Route"<<endl;
    cout<< "3. Optimized Route"<<endl;
    cout<< "4. Greedy Route"<<endl;
    i = ChekingMian();
    //Switch statements to process different user's requirements
    switch (i) {
      // If the user wants to show the highest profit among all routes
      case 1:
        //Compare all profits that are produced by all routes and get the highest profit and corresponding route
        bestRoute = Comparison(sites);
        //Print out the best route and highest profit
        cout<< bestRoute << endl;
        cout<< endl;
        break;
      // If the user chooses a non-optimized route
      case 2:
        cout<< "Non-Optimised Route:"<<endl;
        //Get the retailers that meet the requirements
        meetReq = non.meetsRequirement(sites);
        meetReq = CombineDisCostProfit(meetReq);
        //Display the list of retailers that meet the requirements
        DisplayRetailers(meetReq);
        // Calculate the total distance and weight for the non-optimized route
        distance = calculateTotalDistance(meetReq);
        weight = calculateWeight(meetReq);
        costTime(Time(distance));
        // Display the total cost, profit, distance, and weight to the user
        Display(distance, weight);
        break;
      // If the user chooses an optimized route
      case 3:
        cout<< "Optimised Route:"<<endl;
        //Get the retailers that meet the requirements
        meetReq = Opt.OptimizedRoutes(sites);
        meetReq = CombineDisCostProfit(meetReq);
        //Display the list of retailers that meet the requirements
        DisplayRetailers(meetReq);
        // Calculate the total distance and weight for the optimized route by using the retailers that meet the
requirements
        distance = calculateTotalDistance(meetReq);
        weight = calculateWeight(meetReq);
        costTime(Time(distance));
        // Display the total cost, profit, distance, and weight to the user
        Display(distance, weight);
        break;
      // If the user chooses a greedy route
      case 4:
        vector<RetailerSite> sorted;    //Selected and Sorted retailers
        sorted = gre.selectRetailers(sites);     // Select 8 retailer from all retailer;
        // Print out the information of the selected retailers
        cout<<"Sorted Retailers:"<<endl;
        DisplayCostEff(sorted); // Use the DisplayRetailers function to display the information of the selected retailer
sites
        // Prompt the user to choose between an optimized route and a greedy route
        cout<<"Apply Non-Optimised or Optimised Route with Greedy Route:"<<endl;
        cout<<"1. Optimized Route"<<endl;
        cout<<"2. Non-Optimised Route"<<endl;
        int userInput = Cheking();
        switch (userInput) {
          case 1:
            cout<< "Greedy with Optimised Route:"<<endl;
            meetReq = Opt.meetsRequirement(sorted);
            meetReq = CombineDisCostProfit(meetReq);
            DisplayRetailers(meetReq);
            // If the user chooses an optimized route
            // Calculate the total distance and weight for the optimized route
            distance = calculateTotalDistance(meetReq);
            weight = calculateWeight(meetReq);
            costTime(Time(distance));
            // Display the total cost, profit, distance, and weight to the user
            Display(distance, weight);
            break;
          case 2:
            cout<< "Greedy with Non-Optimised Route:"<<endl;
            meetReq = non.meetsRequirement(sorted);
            meetReq = CombineDisCostProfit(meetReq);
            DisplayRetailers(meetReq);
            // Calculate the total distance and weight for the greedy route
            distance = calculateTotalDistance(meetReq);
            weight = calculateWeight(meetReq);
            costTime(Time(distance));
            // Display the total cost, profit, distance, and weight to the user
            Display(distance, weight);
```

```
                break;
            }
            break;
        }
        if(i==1 || i==2 || i==3 || i==4){
          cout<<"Do you want to continue? if [yes] press 1, and press ANY OTHER key to exit."<<endl;
          cin >> continuenum;
          if(continuenum != "1"){
            // Ask the user if they want to print the chosen route to a file
            cout<< "Do you want to print out the chosen route into the file? if [yes] press 1, and press ANY OTHER key
to exit."<<endl;
            int p;
            cin >> p;
            if(p == 1){
              // If the user chooses to print to file, ask which route they want to print
              cout<<"Please choose the route that you want to print into the file:"<<endl;
              cout<<"1. Non-Optimized Route"<< endl;
              cout<<"2. Optimized Route"<< endl;
              cout<<"3. Greedy with Non-Optimized Route"<< endl;
              cout<<"4. Greedy with Optimized Route"<< endl;
              int route = ChekingMian();
              vector<RetailerSite> sorted;    //Selected and Sorted retailers
              sorted = gre.selectRetailers(sites);     // Select 8 retailer from all retailer;
              // Depending on the user's choice, calculate the route and display it
              switch (route) {
                case 1:
                  // For the non-optimized route, check if the route meets requirements and combine distance and cost
                  meetReq = non.meetsRequirement(sites);
                  break;
                case 2:
                  // For the optimized route, check if the route meets requirements and combine distance and cost
                  meetReq = Opt.OptimizedRoutes(sites);
                  break;
                case 3:
                  // For the greedy route with the non-optimized route, check if the route meets requirements and
combine distance and cost
                  meetReq = non.meetsRequirement(sorted);
                  break;
                case 4:
                  // For the greedy route with the optimized route, check if the route meets requirements and combine
distance and cost
                  meetReq = Opt.OptimizedRoutes(sorted);
                  break;
              }
              //Put the distance and cost of each retailer into the retailer vector
              meetReq = CombineDisCostProfit(meetReq);
              // Display the selected retailer sites
              DisplayRetailerSites(meetReq);
            }
          }
        }
        } while (continuenum == "1");
    time(&end);     // Recording end time

    // Calculating total time taken by the program.
    double time_taken = double(end - start);
    cout << "Time taken by the program is : " << fixed
    << time_taken<<"  sec " << endl;
    return 0;
}
```

b. RetailerSite.hpp

```
#ifndef RetailerSite_hpp
#define RetailerSite_hpp

#include <stdio.h>
#include <string>
using namespace std;
// Define a class for retailer sites, with variables for name, location, x and y coordinates, distance from the start point, order weight, cost, cost-effectiveness and profit
class RetailerSite {
public:
    string name;
    string location;
    double x, y;
    double distance;
    int orderWeight;
    double cost;
    double CostEffectiveness;
    double profit;
};
#endif /* RetailerSite_hpp */
```

c. Comparison.h

```
#ifndef Comparison_h
#define Comparison_h
#include <stdio.h>
#include <vector>
#include <string>
#include "Cost_Effectiveness.h"
#include "RetailerSite.hpp"
#include "Greedy.h"

// Define a function named "stringComparison" that takes a vector of RetailerSite objects as an argument
string Comparison (vector<RetailerSite> retailers){

    // Create instances of the NonOptimizedRoute, OptimizedRoute, and Greedy classes
    NonOptimizedRoute non = *new NonOptimizedRoute;
    OptimizedRoute Opt = *new OptimizedRoute;
    Greedy gre = *new Greedy;

    // Create a vector to store the selected and sorted retailers
    vector<RetailerSite> sorted;

    // Declare variables to store the calculated distance, weight, and profit, and a string to store the result
    double distance;
    double weight;
    double profit;
    string str;
    string str1;

    // Declare a vector to store the retailers that meet the minimum requirements for distance and weight
    vector<RetailerSite> meetReq;

    // Get retailers that meet Non-optimized route
    meetReq = non.meetsRequirement(retailers);
    // Calculate the distance, weight, and profit for the Non-Optimized Route
    distance = calculateTotalDistance(meetReq);
    weight = calculateWeight(meetReq);
    profit = calculateProfit(weight, distance); //Calculate the Non-Optimized profit
    str = "Non-Optimized Route";    //Assign the str with "Non-Optimized Route"
    str1 = "The Profit of Non-Optimized Route: " + to_string(profit);
    cout<< str1<<endl;

    // Get retailers that meet optimized route
    meetReq = Opt.OptimizedRoutes(retailers);
    // Calculate the distance, weight, and profit for the Optimized Route and compare it with the Non-Optimized Route
    distance = calculateTotalDistance(meetReq);
    weight = calculateWeight(meetReq);
    str1 = "The Profit of Optimized Route: " + to_string(calculateProfit(weight, distance));
    cout<< str1 <<endl;
    if (profit < calculateProfit(weight, distance)) {
        profit = calculateProfit(weight, distance); //Calculate the Optimized profit
        str = "Optimized Route";    //Assign the str with "Optimized Route"
    }else if(profit == calculateProfit(weight, distance)){      //If profit is same as non-optimized route
        str = "No matter Non-Optimized or Optimized Routes";
    }

    // Select the top 8 retailers based on cost-effectiveness and calculate the distance, weight, and profit for the Greedy
with Optimized Route
    sorted = gre.selectRetailers(retailers);
    // Get retailers that meet optimized route
    meetReq = Opt.OptimizedRoutes(sorted);
    distance = calculateTotalDistance(meetReq);
    weight = calculateWeight(meetReq);
    str1 = "The Profit of Greedy with Optimized Route: " + to_string(calculateProfit(weight, distance));
    cout<< str1 <<endl;
    if (profit < calculateProfit(weight, distance)) {
        profit = calculateProfit(weight, distance); //Calculate the Greedy with Optimized profit
        str = "Greedy with Optimized Route";    //Assign the str with "Greedy with Optimized Route"
    }else if(profit == calculateProfit(weight, distance) && str == "Optimized Route"){  //If profit is same as optimized
route
        str = "No matter Optimized or Greedy with Optimized Routes";
    }else if (profit == calculateProfit(weight, distance) && str == "No matter Non-Optimized or Optimized
Routes"){ //If profit is same as non-optimized and optimized routes
        str = "No matter Non-Optimized, Optimized or Greedy with Optimized Routes";
    }

    // Get retailers that meet Non-optimized route
    meetReq = non.meetsRequirement(sorted);
    // Calculate the distance, weight, and profit for the Greedy with Non-Optimized Route
    distance = calculateTotalDistance(meetReq);
    weight = calculateWeight(sorted);
    str1 = "The Profit of Greedy with Non-Optimized Route: " + to_string(calculateProfit(weight, distance));
    cout<< str1 <<endl;
    if (profit < calculateProfit(weight, distance)) {
        profit = calculateProfit(weight, distance); //Calculate the Greedy with Non-Optimized profit
        str = "Greedy with Non-Optimized Route";    //Assign the str with "Greedy with Non-Optimized Route"
    }else if(profit == calculateProfit(weight, distance) && str == "Non-Optimized Route"){  //If profit is same as non-
optimized route
        str = "No matter Optimized or Greedy with Non-Optimized Routes";
    }else if (profit == calculateProfit(weight, distance) && str == "No matter Non-Optimized or Optimized
Routes"){ //If profit is same as non-optimized and optimized routes
        str = "No matter Non-Optimized, Optimized or Greedy with Non-Optimized Routes";
    }else if (profit == calculateProfit(weight, distance) && str == "No matter Non-Optimized, Optimized or Greedy with
Optimized Routes"){  //If profit is same as non-optimized, optimized and greedy with optimized routes
        str = "Profit for all routes are all same";
    }

    // Construct a string containing the best route and the total profit
    str = "Best Route: "+str + "\nThe Highest Profit:"+to_string(profit);

    // Return the result string
    return str;
```

```
}

//calculate the driving time for driver
void costTime(double time){
   if (time > 8.0) {
       cout << endl;
       cout << "Take into account the driver's safety."<<endl;
       cout << "This route is not recommended because the driver will be exhausted after driving for more than 8
hours."<<endl;
   }
}
#endif /* Comparison_h */
```

d. <u>Display.h</u>

```
#ifndef Display_h
#define Display_h
#include <iostream>
#include <vector>
#include <string>
#include <iostream>
#include <fstream>
#include "RetailerSite.hpp"
#include "Cost_Effectiveness.h"
#include "Checking.h"
using namespace std;

// This function takes two parameters as input - distance and weight
void Display(double distance, double weight){
   double cost;
   double profit;
   // Calculate the cost based on the total distance
   cost = Cost(distance);
   // Calculate the profit based on the weight and distance
   profit = calculateProfit(weight, distance);
   // Display the total distance, weight, cost, and profit to the user
   cout<< endl;
   cout<< "Total Distance: "<< fixed << setprecision(2)<< distance<< endl;
   cout<< "Total Weight: "<< fixed << setprecision(2)<< weight<< endl;
   cout<< "Total Cost:" << fixed<< setprecision(2)<< cost << endl;
   cout<< "Total Profit: "<< fixed << setprecision(2)<< profit <<endl;
}

// This function takes a vector of RetailerSite objects as input and displays their information in a formatted table to the
console.
void DisplayRetailers(vector<RetailerSite> retailers){
   bool empty = isVectorEmpty(retailers);  // Check if any retailers meet the requirements
   if(!empty){ // If there are retailers that meet the requirements
       // Print out all the information about the retailers that meet the requirements
       cout<<setw(fieldWidth)<<"Name"
       << setw(fieldWidth)<< "Location"
       << "\t\t\t\t\t\tDistance"
       << setw(fieldWidth)<< "OrderWeight"
       << setw(fieldWidth)<< "Cost"
       << setw(fieldWidth)<< "Profit"<<endl;
       for(int i = 0; i< retailers.size(); i++){ // Loop through each retailer and display its information
           // Print out the name, location, distance, order weight, cost, and profit of the current retailer
           cout<< setw(fieldWidth) <<retailers[i].name
           <<setw(fieldWidth) << retailers[i].location<< "("<< fixed<< setprecision(2)<< retailers[i].x<<","<<fixed<<
setprecision(2)<< retailers[i].y<<")"
           <<setw(fieldWidth) << retailers[i].distance
           <<setw(fieldWidth) << retailers[i].orderWeight
```

```
            <<setw(fieldWidth) << retailers[i].cost
            <<setw(fieldWidth) << retailers[i].profit <<endl;
        }
    }else{  // If there are no retailers that meet the requirements
        cout<< endl;
        cout<< "No Retailer Meet the Requirements."<<endl;   // Print out a message indicating that no retailer meets the
requirements
    }
}

// This function takes a vector of RetailerSite objects as input and displays them in a formatted table in a text file.
void DisplayRetailerSites(const vector<RetailerSite>& sites) {
    // Check whether the file already exists or not
    FileExists();
    // Open a file named "RL_InitialRoute.txt" for writing and create an ofstream object named "outfile".
    ofstream outfile("RL_InitialRoute.txt"); //open the file for writing
    if (outfile.is_open()) { //if the file is opened
        // Write the header row to the file, with appropriate spacing and field width
        outfile << setw(fieldWidth) << "Name"
        <<setw(fieldWidth)<< "Location"
        << setw(fieldWidth) << "X"
        << setw(fieldWidth) << "Y"
        << setw(fieldWidth) << "Order Weight"
        << setw(fieldWidth) << "Cost"
        << setw(fieldWidth)<< "Profit"
        << setw(fieldWidth) << "Distance"<<endl;

        // Loop through each RetailerSite object in the vector and write its attributes to the file, with appropriate spacing
and field width
        for (const RetailerSite& site : sites) {
            outfile << setw(fieldWidth) << site.name
            << setw(fieldWidth) << site.location
            << setw(fieldWidth) << site.x
            << setw(fieldWidth) << site.y
            << setw(fieldWidth) << site.orderWeight
            << setw(fieldWidth) << site.cost
            <<setw(fieldWidth) << site.profit
            << setw(fieldWidth) << site.distance<< endl;
        }

        // Close the file and print a message indicating that the information has been written to the file
        outfile.close();
        cout<< "Information is already written into the file."<<endl;
    } else { //if the file cannot be opened
        cout << "Error: Unable to open file for writing.\n";
    }
}

// This function takes a vector of RetailerSite objects as input and displays them in a formatted table in the console,
including the cost-effectiveness metric.
void DisplayCostEff(const vector<RetailerSite>& retailers) {
// print out all the information about the retailers that meet the requirements
    cout<<setw(fieldWidth)<<"Name"
    << setw(fieldWidth)<< "Location"
    << "\t\t\t\t\t\tDistance"
    << setw(fieldWidth)<< "OrderWeight"
    << setw(fieldWidth)<< "Cost"
    << setw(fieldWidth)<< "Profit"
    << setw(fieldWidth1.5)<< "CostEffctiveness"<<endl;
    for(int i = 0; i< retailers.size(); i++){ //for-loop to calculate the total distance, distance between retailers that meet the
requirements and costs of each retailer
    //Print out all the information about the retailers that meet the requirements
```

```
        cout<< setw(fieldWidth) <<retailers[i].name
        <<setw(fieldWidth) << retailers[i].location<< "("<< fixed<< setprecision(2)<< retailers[i].x<<","<<fixed<< 
setprecision(2)<< retailers[i].y<<")"
        <<setw(fieldWidth) << retailers[i].distance
        <<setw(fieldWidth) << retailers[i].orderWeight
        <<setw(fieldWidth) << retailers[i].cost
        <<setw(fieldWidth) << retailers[i].profit
        <<setw(fieldWidth1.5) << retailers[i].CostEffectiveness<<endl;
    }
}
#endif /* Display_h */
```

e. Cost_Effectiveness.h

```
#ifndef Cost_Effectiveness_h
#define Cost_Effectiveness_h
#include <cmath>
using namespace std;

// Define a function to calculate the cost of a trip based on the distance travelled, fuel consumption, and wage costs
const double TIME_PER_KM = 3.0 / 60.0; // in hours
const double FUEL_COST = 2.5; // in RM per km
const double WAGE_PER_HOUR = 15.0; // in RM per hour
const int fieldWidth = 15; // Set the field width for output formatting

// Define a function to calculate the cost of a trip based on the distance travelled, fuel consumption, and wage costs
double Cost(double distance) {
    double time = distance * TIME_PER_KM;
    double fuelCost = distance * FUEL_COST;
    double wageCost = time * WAGE_PER_HOUR;
    return fuelCost + wageCost;
}

// Define a function to calculate the profit margin gained by the supplier and/or wholesaler based on the weight of 
goods being ordered and the delivery distance
double calculateProfit(double weight, double distance) {
    double deliveryCost = Cost(distance);
    double sellingPrice = weight * 2.0; // assume selling price is 2 RM per kg
    return sellingPrice - deliveryCost;
}

// Define a function to calculate the Euclidean distance between two points in 2D space
double calculateDistance(double x1, double y1, double x2, double y2) {
    double dx = x2 - x1;
    double dy = y2 - y1;
    return sqrt(dx*dx + dy*dy);
}

// This function finds the retailer in the given vector with the smallest distance to the given "next" retailer.// Find the 
smallest distance between this retailer and to next retailer
RetailerSite findSmallestDistance(RetailerSite next, vector<RetailerSite> retailers) {
    double smallestDistance = numeric_limits<double>::max();  // Initialize the smallestDistance to the maximum value 
for a double
    double distance;    // Distance between retailers
    RetailerSite nextR;
    // Iterate through the retailers' vector and calculate the distance to each retailer
    for(int i = 0; i < retailers.size(); i++) {
        distance = calculateDistance(next.x, next.y, retailers[i].x, retailers[i].y);   //Calculate the distances
        // If the distance is smaller than the current smallest distance, update the smallest distance and the next retailer
        if(distance < smallestDistance && retailers[i].name != next.name) {     // if distance smaller than the smallest 
distance and retailer is not itself
            nextR = retailers[i];   //Find the nearest retailer
```

```
                smallestDistance = distance;    // Assign the distance as the smallest distance
            }
        }
    // Return the retailer with the smallest distance to the given "next" retailer
    return nextR;
}


//function declaration that takes a vector of RetailerSite objects as an argument and returns a double.
double calculateWeight(vector<RetailerSite> meetReq) {
    double weight = 0.0;    //initializes a double variable named weight with a value of 0.0.
    // Calculate total distance, fuel cost, and wage cost for each retailer on the route
    for (int i = 0; i < meetReq.size(); i++) {    //starts a for loop that iterates over the retailers' vector, using the variable i
as the index.
            weight += meetReq[i].orderWeight;    //if the retailer meets the weight requirement, the weight of the retailer's
order is added to the weight variable.
    }
    return weight;  //returns the total weight of orders for retailers that meet the weight requirement.
}

//function declaration that takes a vector of RetailerSite objects as an argument and returns a double.
double calculateTotalDistance(vector<RetailerSite> meetReq) {
    double distance = 0.0;      //Initial the total distance as 0.0
    // Calculate total distance, fuel cost, and wage cost for each retailer on the route
    for(int i = 0; i< meetReq.size(); i++){ //for-loop to calculate the total distance, distance between retailers that meet
the requirements and costs of each retailer
        if (i == 0) {   //For the first retailer
            meetReq[i].distance = calculateDistance(0, 0, meetReq[i].x, meetReq[i].y);
        }else if(i == meetReq.size()){  //For last retailer
            meetReq[i].distance = calculateDistance(meetReq[i-1].x, meetReq[i-1].y, meetReq[i].x, meetReq[i].y);
            distance += calculateDistance(0, 0, meetReq[i].x, meetReq[i].y);    //Distance between the last retailer and the
wholesaler
        }else{  // Other retailers
            meetReq[i].distance = calculateDistance(meetReq[i-1].x, meetReq[i-1].y, meetReq[i].x, meetReq[i].y);
        }
        // Add the retailers' distance to total distance
        distance += meetReq[i].distance;
    }
    return distance;
}

//This function takes the cost and profit as input and returns the cost-effectiveness as output.
double calculateCostEffectiveness(double cost, double profit) {
    //This checks if the cost is not equal to zero using an if statement. If the cost is not equal to zero, the block of code
within the curly braces {} is executed.
    if (cost != 0) {
        //If the cost is not zero, this line calculates the ratio of profit to cost and returns it as the output of the function.
        return profit / cost;
    } else {
        // Return a large negative number if cost is zero
        return -numeric_limits<double>::infinity();
    }
}
#endif /* Cost_Effectiveness_h */
```

f. Generate.h

```
#ifndef Generate_h
#define Generate_h
#include <iostream>
#include <fstream>
#include <vector>
```

```
#include <string>
#include "RetailerSite.hpp"
#include "Cost_Effectiveness.h"
using namespace std;

void generateRetailerSites(vector<RetailerSite>& sites) {
   const int numSites = 10; // the number of sites to generate
   const double maxX = 40.0; // the maximum x coordinate
   const double maxY = 40.0; // the maximum y coordinate

   // Resize the vector to the number of sites to be generated
   sites.resize(numSites);

   // Generate a random set of coordinates and order weights for each site
   for (int i = 0; i < numSites; i++) {
      RetailerSite& site = sites[i];
      site.name = "Retailer " + to_string(i+1); // set the name of the site
      site.location = "Location " + to_string(i+1); // set the location of the site
      site.x = (double) rand() / RAND_MAX * maxX; // generate a random x coordinate
      site.y = (double) rand() / RAND_MAX * maxY; // generate a random y coordinate
      site.orderWeight = rand() % 201; // generate a random order weight between 0 and 200 kg
      site.cost = 0.0; // initialize the delivery cost to 0
      site.profit = 0.0;  // initialize the delivery profit to 0
      site.distance = calculateDistance(0, 0, site.x, site.y); // initialize the delivery distance to the distance between
the wholesaler and retailers
   }
}

// This function takes a vector of RetailerSite objects as input and returns a vector of the same objects after
updating their distance and cost properties
vector<RetailerSite> CombineDisCostProfit(vector<RetailerSite> meetReq) {
   for(int i = 0; i< meetReq.size(); i++){ // Loop through the meetReq vector
      if (i == 0) {   // Check if it's the first retailer
         meetReq[i].distance = calculateDistance(0, 0, meetReq[i].x, meetReq[i].y);  // Calculate the distance
between the starting point (0,0) and this retailer's location
      }else{  // For other retailers
         meetReq[i].distance = calculateDistance(meetReq[i-1].x, meetReq[i-1].y, meetReq[i].x, meetReq[i].y);    //
Calculate the distance between this retailer and the previous one in the meetReq vector
      }
      meetReq[i].cost = Cost(meetReq[i].distance);  // Calculate the cost of visiting this retailer, based on its
distance
      meetReq[i].profit = calculateProfit(meetReq[i].orderWeight, meetReq[i].distance);
   }
   return meetReq; // Return the updated vector of RetailerSite objects
}
#endif /* Generate_h */
```

g. Checking.h

```
#ifndef Checking_h
#define Checking_h
#include <fstream>
#include <iostream>

//check the validation of user input
int Cheking(){
   int userInput;      //Declaration of userInput
   do {
      cout << "Please enter 1 or 2: "; // prompt user for input
      cin >> userInput; // read user input

      // check if the input failed or is not a number
```

```
        if (cin.fail()) {
            cin.clear(); // clear the input stream
            cin.ignore(numeric_limits<streamsize>::max(), '\n'); // ignore any characters in the input buffer
            cout << "Invalid input. Please enter a number.\n"; // display an error message
        }
        // check if the input is out of range
        else if (userInput < 1 || userInput > 2) {
            cout << "Invalid input. Please enter 1 or 2.\n"; // display an error message
        }
    } while (userInput < 1 || userInput > 2); // loop until valid input is entered
    return userInput;
}

int ChekingMian(){
    int userInput;      //Declaration of userInput
    do {
        cout << "Please enter 1, 2, 3 or 4: "; // prompt user for input
        cin >> userInput; // read user input

        // check if the input failed or is not a number
        if (cin.fail()) {
            cin.clear(); // clear the input stream
            cin.ignore(numeric_limits<streamsize>::max(), '\n'); // ignore any characters in the input buffer
            cout << "Invalid input. Please enter a number.\n"; // display an error message
        }
        // check if the input is out of range
        else if (userInput < 1 || userInput > 4) {
            cout << "Invalid input. Please enter 1, 2, 3 or 4.\n"; // display an error message
        }
    } while (userInput < 1 || userInput > 4); // loop until valid input is entered
    return userInput;
}

//check if the vector is empty
bool isVectorEmpty(vector<RetailerSite> vec) {
    return vec.empty();
}

void FileExists(){
    string filename = "RL_InitialRoute.txt";
    // Check if the file exists
    ifstream file(filename);
    if (file.good()) {
        cout << "File exists." << endl;
    } else {
        // If the file does not exist, create a new one
        ofstream newfile(filename);
        if (newfile.good()) {
            cout << "New file created." << endl;
        } else {
            cerr << "Error creating file." << endl;
        }
    }
}
#endif /* Checking_h */
```

h. DeleteRetailers.h

```
#ifndef DeleteRetailers_h
#define DeleteRetailers_h
#include "RetailerSite.hpp"
```

```
void deleteRetailer(vector<RetailerSite>& retailers, string retailerName) {
    for (auto it = retailers.begin(); it != retailers.end(); ++it) {
        if (it->name == retailerName) {
            retailers.erase(it);
            break;
        }
    }
}

#endif /* DeleteRetailers_h */
```

i. Non-Optimised.h

```
#ifndef Non_Optimised_h
#define Non_Optimised_h
#include <stdio.h>
#include <vector>
#include <string>
#include "Cost_Effectiveness.h"
#include "RetailerSite.hpp"
#include "Display.h"

//Defines a class named NonOptimizedRoute.
class NonOptimizedRoute {
public:
    // Function to check if a retailer meets the weight requirement and is within a certain distance of the depot
    vector<RetailerSite> meetsRequirement(vector<RetailerSite> r) {
        vector<RetailerSite> meetReq;   // vector to store retailer sites that meet the weight requirement
        for (int i = 0; i < r.size(); i++) {     //for-loop to check retailers one by one
            if (r[i].orderWeight >= 50) {    // Add retailer sites that meet the weight
                meetReq.push_back(r[i]);   //  if retailer meet the requirements store in the meetReq vector
            }
        }
        return meetReq; //Return the retailers that store in a vector meet the requirements
    }
};
#endif /* Non_Optimised_h */
```

j. Optimised.h

```
#ifndef Optimised_h
#define Optimised_h
#include <stdio.h>
#include <vector>
#include <string>
#include "Cost_Effectiveness.h"
#include "RetailerSite.hpp"
#include "Display.h"
#include "DeleteRetailers.h"
using namespace std;

class OptimizedRoute {
public:
    // Function to check if a retailer meets the weight requirement and is within a certain distance of the depot
    vector<RetailerSite> meetsRequirement(vector<RetailerSite> r) {
        vector<RetailerSite> meetReq;   // vector to store retailer sites that meet the weight requirement
        double distance;
        for (int i = 0; i < r.size(); i++) {     //for-loop to check retailers one by one
            distance = calculateDistance(0, 0, r[i].x, r[i].y); // Calculate distance from depot to retailer
            if (r[i].orderWeight >= 100 && distance <= 30) {    // Add retailer sites that meet the weight and distance
                meetReq.push_back(r[i]);   //  if retailer meet the requirements store in the meetReq vector
            }
        }
        return meetReq; //Return the retailers that store in a vector meet the requirements
    }

     // Function to apply with the optimized route which always finds the nearest retailer
    vector<RetailerSite> OptimizedRoutes(vector<RetailerSite> sites) {
        vector<RetailerSite> meetReq = meetsRequirement(sites); // Select retailers that meet the requirements
        int length = meetReq.size();    //Length of the meetReq vector
        vector<RetailerSite> sorted; // Vector to store the sorted retailers
        RetailerSite nextR; // Store the next retailer to add to the sorted list
        RetailerSite preR; // Store the previous retailer added to the sorted list

        // Find the retailer with the smallest distance to the depot
        for (int i=0; i<length; i++) {
            if (i == 0) { // If it's the first retailer, find the smallest distance to the depot
                double smallestDistance = numeric_limits<double>::max(); // Initialize the smallest distance as the
maximum value of double
                for (int j = 0; j < meetReq.size(); j++) {
                    double distance = meetReq[j].distance; // Get the distance of the current retailer to the depot
                    if (distance < smallestDistance) { // If it's smaller than the current smallest distance, update the next
retailer to add
                        nextR = meetReq[j];
                        smallestDistance = distance;
                    }
                }
            } else { // For the remaining retailers, find the smallest distance to the previous retailer added
                nextR = findSmallestDistance(preR, meetReq); // Find the next retailer with the smallest distance to the
previous retailer added
                deleteRetailer(meetReq, preR.name); // Remove the previous retailer from the list to avoid adding it
again
            }
            sorted.push_back(nextR); // Add the next retailer with the smallest distance to the sorted list
            preR = nextR; // Update the previous retailer added to the sorted list
        }
        return sorted; // Return the sorted list of retailers
        }
    };

#endif /* Optimised_h */
```

k. <u>Greedy.h</u>

```
#ifndef Greedy_h
#define Greedy_h
#include <stdio.h>
#include <vector>
#include <string>
#include "Cost_Effectiveness.h"
#include "RetailerSite.hpp"
#include "Display.h"
#include "Optimised.h"
#include "Non-Optimised.h"

class Greedy{
public:
    static const int NUM_DELIVERIES = 8;    // Select the 8 retailers from all retailers
    //This is a function declaration for a function called sortByCostEff. It takes in two arguments of type
RetailerSite passed by reference and returns a bool
    static bool sortByCostEff(const RetailerSite& lhs, const RetailerSite& rhs) {
        // Sort the retailer sites by their cost-effectiveness, in descending order
        return lhs.CostEffectiveness > rhs.CostEffectiveness;
    }
```

```
   //Function used to sort and selected the retailers by the greedy algorithm and return the sorted and selected
retailers
   vector<RetailerSite> selectRetailers(vector<RetailerSite>& retailers) {
      vector<RetailerSite> SortRe;  // Initialize a new vector to store sorted retailers
      // Calculate the distance and cost for each retailer site
      SortRe = retailers;  // Copy the original retailers vector to the SortRe vector
      for(int i=0;i<SortRe.size();i++){
         SortRe[i].cost = Cost(SortRe[i].distance);  // Calculate the cost of delivering to the retailer
         SortRe[i].profit = calculateProfit(SortRe[i].orderWeight, SortRe[i].distance);  //Calculate the profit of
delivering to the retailer
         SortRe[i].CostEffectiveness = calculateCostEffectiveness(SortRe[i].cost, SortRe[i].profit); //Calculate the
CostEffectiveness of delivering to the retailer
      }
      // Sort the retailer sites by their cost, in ascending order
      sort(SortRe.begin(), SortRe.end(), sortByCostEff);  // Use the sortByCostEff function to sort the retailer sites
by their CostEffectiveness
      SortRe.resize(NUM_DELIVERIES);  // Resize the SortRe vector to contain only the first
NUM_DELIVERIES elements (i.e., the top NUM_DELIVERIES cheapest retailer sites)
      return SortRe;  // Return the vector of selected retailer sites
   }
};
#endif /* Greedy_h */
```

l. <u>RL_InitialRoute.txt</u>

This program gets total 12 files, `Comparison.h`, `Checking.h`, `Display.h`, `RetailerSite.hpp`, `Cost_Effectiveness.h`, `Generate.h`, `DeleteRetailers.h`, `Non-Optimised.h`, `Optimised.h`, `Greedy.h`, `RL_InitialRoute.txt` and `Main.cpp` to process the whole system, where ` Comparison.h` is used to compare and get the highest profit among all three routes, `Checking.h` file is used to check the user input, `Display.h` is used to display the information that users can see, `RetailerSite.hpp` is used to define the basic class structure components of retailer site, `Cost_Effectiveness.h` is used to calculate cost effectiveness, distance, profit of all retailers, `Generate.h` is used to generate the random retailers information and combine cost, distance and profit after applying the routes of all retailers, `DeleteRetailers.h` is used to delete the retailers from the retailer vector, `Non-Optimised.h` is used to accomplish non-optimised algorithm, ` Optimised.h` is used to accomplish optimised algorithm, `Greedy.h` is used to accomplish greedy algorithm, `RL_InitialRoute.txt` is used to store the retailers' information.

The code is an implementation of a program that generates a list of RetailerSites and allows the user to choose between different types of routes for visiting these RetailerSites. The main function starts by initializing two time_t variables, start and end, that will be used to time the execution of the program. Then, the random number generator is seeded with the current time. The program creates a vector of RetailerSite objects and populates it with randomly generated RetailerSite objects using the generateRetailerSites function. The DisplayRetailerSites function is then called to display the information for each RetailerSite in the vector.

The program creates instances of the NonOptimizedRoute, OptimizedRoute, and Greedy classes, and declares a vector to store RetailerSites that meet a weight requirement. It also declares variables to store the calculated distance, weight, cost, and profit, as well as a string variable to show the best route and highest profit.

The program uses a do-while loop to control the flow of the program and prompt the user to choose between different types of routes. The ChekingMian function is called to validate the user's input. The user can choose between four options: 1) show the highest profit among all routes, 2) choose a non-optimized route, 3) choose an optimized route, and 4) choose a greedy route.

If the user chooses option 1, the program calls the Comparison function to compare all profits produced by all routes and get the highest profit and corresponding route. The best route and highest profit are then printed to the console.

If the user chooses option 2, the program calls the meetsRequirement function of the NonOptimizedRoute class to get the RetailerSites that meet the weight requirement, combines the distance, cost, and profit of these RetailerSites, and displays the list of RetailerSites that meet the requirement using the DisplayRetailers function. The program then calculates the total distance and weight for the non-optimized route using the calculateTotalDistance and calculateWeight functions and displays the total cost, profit, distance, and weight to the user.

If the user chooses option 3, the program calls the OptimizedRoutes function of the OptimizedRoute class to get the RetailerSites that meet the weight requirement and optimize the route, combines the distance, cost, and profit of these RetailerSites, and displays the list of RetailerSites that meet the requirement using the DisplayRetailers function. The program then calculates the total distance and weight for the optimized route using the calculateTotalDistance and calculateWeight functions and displays the total cost, profit, distance, and weight to the user.

If the user chooses option 4, the program creates an instance of the Greedy class and calls the selectRetailers function to select and sort eight RetailerSites from all RetailerSites. The DisplayCostEff function is then used to display the information of the selected RetailerSites. The user is prompted to choose between an optimized route and a non-optimized route. Depending on the user's choice, the program either calls the meetsRequirement function of the OptimizedRoute class or the meetsRequirement function of the NonOptimizedRoute class to get the RetailerSites that meet the weight requirement, combines the distance, cost, and profit of these RetailerSites, and displays the list of RetailerSites that meet the requirement using the DisplayRetailers function. The program then calculates the total distance and weight for the route using the calculateTotalDistance and calculateWeight functions and displays the total cost, profit, distance, and weight to the user.

Finally, the user is prompted to choose whether to continue with the program or not. If the user enters "1", the program continues with another iteration of the do-While loop. If the user enters anything else, the program exits the loop and the message "Thank you for using!" is printed to the console. If the user enters a value other than "1" for `**continuenum**`, the program will skip this block of code entirely. Otherwise, it will prompt the user to choose whether or not to print the chosen route to a file. If the user enters "1" to indicate they want to print the route to a file, the program will prompt the user to choose which type of route they want to print (non-optimized, optimized, etc.) and then calculate and display the selected route based on the user's choice.

II. Description of all the components, structures and data structures, basic and application functions

A. RetailerSite.hpp

This is a class called "RetailerSite", which represents a retailer's physical location. It contains the following components:

- **name**: a string representing the name of the retailer
- **location**: a string representing the physical location of the retailer
- **x** and **y**: doubles representing the coordinates of the retailer's location
- **distance**: a double representing the distance between the retailer's location and a given point (e.g., a customer's location)
- **orderWeight**: an integer representing the weight of an order that a customer may place with this retailer
- **cost**: a double representing the cost for the retailer to fulfil an order of a given weight
- **CostEffectiveness**: a double representing the cost-effectiveness of fulfilling an order with this retailer (i.e., how much profit the retailer can make per unit cost)
- **profit**: a double representing the profit the retailer will make from fulfilling an order of a given weight.

B. Cost_Effectiveness.h

The code defines several functions to calculate different aspects of a delivery route for a wholesaler that has to deliver goods to several retailers.

1. **Cost(double distance)** - calculates the cost of a trip based on the distance travelled, fuel consumption, and wage costs.

2. **calculateProfit(double weight, double distance)** - calculates the profit margin gained by the supplier and/or wholesaler based on the weight of goods being ordered and the delivery distance.

3. **calculateDistance(double x1, double y1, double x2, double y2)** - calculates the Euclidean distance between two points in 2D space.

4. **findSmallestDistance(RetailerSite next, vector<RetailerSite> retailers)** - finds the retailer in the given vector with the smallest distance to the given "next" retailer.

5. **calculateWeight(vector<RetailerSite> meetReq)** - calculates the total weight of orders for retailers that meet the weight requirement.

6. **calculateTotalDistance(vector<RetailerSite> meetReq)** - calculates the total distance and cost for a delivery route for the given vector of retailers that meet the weight requirement.

7. **calculateCostEffectiveness(double cost, double profit)** - calculates the cost-effectiveness of a delivery route by calculating the ratio of profit to cost, provided that the cost is not zero.

<u>The data structures used:</u>

- **distance** - represents the distance travelled in miles
- **weight** - represents the weight of the goods being ordered in pounds
- **x1, y1, x2, y2** - represent the coordinates of two points in 2D space
- **next** - represents a RetailerSite object that is used as a reference point to find the retailer with the smallest distance
- **retailers** - represents a vector of RetailerSite objects
- **meetReq** - represents a vector of RetailerSite objects that meet the weight requirement
- **cost** - represents the cost of a delivery route
- **profit** - represents the profit margin gained by the supplier and/or wholesaler

C. <u>Display.h</u>

1. **Display(double distance, double weight)**: This function takes two parameters as input, **distance** and **weight**, and calculates the cost and profit based on the distance and weight. The function then displays the total distance, weight, cost, and profit to the user in the console.

2. **DisplayRetailers(vector<RetailerSite> retailers)**: This function takes a vector of **RetailerSite** objects as input and displays their information in a formatted table to the console. The function first checks whether any retailers meet the requirements and if there are, it prints out all the information about the retailers that meet the requirements, including their name, location, distance, order weight, cost, and profit. If no retailers meet the requirements, the function prints out a message indicating that no retailer meets the requirements.

3. **DisplayRetailerSites(const vector<RetailerSite>& sites)**: This function takes a vector of **RetailerSite** objects as input and displays their information in a formatted table in a text file named "RL_InitialRoute.txt". The function first checks whether the file already exists or not, and if it does, it overwrites the file. The function then opens the file for writing and writes the header row to the file, with appropriate spacing and field width. After that, the function loops through each **RetailerSite** object in the vector and writes its attributes to the file, with appropriate spacing and field width. Finally, the function closes the file and prints a message indicating that the information has been written to the file.

4. **DisplayCostEff(const vector<RetailerSite>& retailers)**: This function takes a vector of **RetailerSite** objects as input and displays their information in a formatted table in the console, including the cost-effectiveness metric. The function first prints out the header row to the console, with appropriate spacing and field width. After that, the function loops through each **RetailerSite** object in the vector and prints out its attributes, including its name, location, distance, order weight, cost, profit, and cost-effectiveness metric, which is calculated as the profit divided by the cost.

The data structures used:

- **distance**: a double variable representing the total distance.
- **weight**: a double variable representing the weight of the order.
- **retailers**: a vector of RetailerSite objects representing the retailers to be displayed.
- **sites**: a vector of RetailerSite objects representing the retailer sites to be displayed and written to the text file.

D. Generate.h

Function:

1. **generateRetailerSites(vector<RetailerSite>& sites):** takes a vector of RetailerSite objects as input and generates a set of random coordinates and order weights for each site. It then initializes the delivery cost, profit, and distance properties for each site based on the distance between the wholesaler and retailers.

2. **CombineDisCostProfit** function takes a vector of RetailerSite objects as input and updates their distance, cost, and profit properties based on the distance between each retailer and the previous one, the distance-based cost of visiting each retailer, and the profit generated from each retailer's order weight and distance. It then returns the updated vector of RetailerSite objects.

The data structures used:

The **RetailerSite** structure represents a retailer site and contains the following properties:

- **name**: the name of the retailer site
- **location**: the location of the retailer site
- **x**: the x coordinate of the retailer site
- **y**: the y coordinate of the retailer site
- **orderWeight**: the order weight of the retailer site
- **cost**: the delivery cost of visiting the retailer site
- **profit**: the profit generated from the retailer site's order weight and distance
- **distance**: the distance between the retailer site and the previous retailer site in the delivery route.

E. Checking.h

The several functions to perform different tasks:

1. **Cheking()**: checks the validation of user input. It prompts the user to enter either 1 or 2 and loops until a valid input is entered. It returns the valid user input.

2. **ChekingMian()**: it is similar to **Cheking()** but prompts the user to enter either 1, 2, 3 or 4 and loops until a valid input is entered. It returns the valid user input.

3. **isVectorEmpty(vector<RetailerSite> vec)**: checks whether a vector of RetailerSite objects is empty or not. It returns a boolean value indicating whether the vector is empty.

4. **FileExists():** checks whether a file with the name "RL_InitialRoute.txt" exists or not. If the file exists, it prints a message indicating that the file exists. Otherwise, it creates a new file with the same name and prints a message indicating that a new file is created. If there is any error creating the file, it prints an error message.

The data structures used:

1. **vector<RetailerSite>** is a vector of **RetailerSite** objects. It is used to store information about the retailer sites.
2. int **userInput** is an integer variable used to store the user input.
3. string **filename** is a string variable used to store the name of the file to be checked or created.
4. **ifstream file** and **ofstream newfile** are file input and output stream objects used to check the existence of a file and create a new file, respectively.

F. DeleteRetailer.h

Function:

1. **deleteRetailer(vector<RetailerSite>& retailers, string retailerName):** takes a vector of RetailerSite objects and a string representing the name of a retailer to be deleted from the vector. It uses a for loop to iterate over each element of the vector using an iterator (auto it = retailers.begin()), checking if the name of the current element (it->name) matches the retailer name parameter passed to the function. If a match is found, the element is erased from the vector using the erase() function and the loop is exited using the break statement.

The data structures used:

- **retailers**: A reference to a vector of **RetailerSite** objects. This vector contains the list of retailers from which we want to delete a particular retailer.
- **retailerName** A string that represents the name of the retailer that we want to delete from the vector of retailers.

G. Non-Optimised. h

Function:

1. **vector<RetailerSite> meets the requirement (vector<RetailerSite> r):** a public method that takes a vector of RetailerSite objects as input and returns a vector of RetailerSite objects that meet the weight requirement.

The class variables:

- "r" vector: a vector of RetailerSite objects that are passed as input to the "meetsRequirement" method.
- "meetReq" vector: a vector that stores retailer sites that meet the weight requirement and is returned by the "meetsRequirement" method.
- "i" integer: a loop counter variable used to iterate through the "r" vector.
- "orderWeight" double: a variable that stores the order weight of a retailer site.
- "50" integer: a constant representing the weight requirement for a retailer site.

H. Optimised. h

The **OptimizedRoute** class defines two member functions:

1. **meetsRequirement**: This function takes a vector of **RetailerSite** objects as input and returns a vector of **RetailerSite** objects that meet the weight and distance requirements. The function checks each retailer site in the input vector one by one, calculates the distance from the depot to the retailer site, and adds the retailer site to the output vector if its order weight is greater than or equal to 100 and its distance from the depot is less than or equal to 30. The function returns the output vector of retailer sites that meet the requirements.

2. **OptimizedRoutes**: This function takes a vector of **RetailerSite** objects as input and returns a vector of **RetailerSite** objects that are sorted based on their distance from the depot. The function first calls the **meetsRequirement** function to select retailer sites that meet the weight and distance requirements. It then finds the retailer site with the smallest distance to the depot and adds it to a new vector. For the remaining retailer sites, it finds the retailer site with the smallest distance to the previous retailer site added and adds it to the new vector. The function repeats this process until all retailer sites have been added to the new vector. The function returns the new vector of retailer sites sorted by distance.

The class variables:

- **vector<RetailerSite>**: A vector of **RetailerSite** objects is used to store the input and output retailer sites.
- **double distance**: A **double** variable is used to store the distance from the depot to each retailer site.
- **int length**: An **int** variable is used to store the length of the input vector of retailer sites.
- **vector<RetailerSite> sorted**: A vector of **RetailerSite** objects is used to store the sorted retailer sites.
- **RetailerSite next**: A **RetailerSite** object is used to store the next retailer site to add to the sorted list.
- **RetailerSite preR**: A **RetailerSite** object is used to store the previous retailer site added to the sorted list.

I. Greedy. h

Function:

1. **static bool sortByCostEff(const RetailerSite& lhs, const RetailerSite& rhs)** - a static member function of the **Greedy** class that takes two arguments of type

RetailerSite by reference and returns a boolean. It sorts the **RetailerSite** objects by their cost-effectiveness in descending order.

2. **vector<RetailerSite> selectRetailers(vector<RetailerSite>& retailers)** - a member function of the **Greedy** class that takes a reference to a vector of **RetailerSite** objects as input and returns a vector of **RetailerSite** objects. It selects the top **NUM_DELIVERIES** (i.e., 8) retailers using a greedy algorithm based on cost-effectiveness, sorts them by cost-effectiveness, and returns the resulting vector.

The class variables:

- **NUM_DELIVERIES**: a constant integer variable set to 8, representing the number of retailers selected from all retailers.
- **SortRe**: a vector of **RetailerSite** objects used to store sorted retailers.
- **i**: an integer variable used as a counter in the for loop to calculate the cost, profit, and cost-effectiveness of delivering to each retailer site.

J. Comparison.h

Function:

1. **string Comparison (vector<RetailerSite> retailers):** it takes a vector of **RetailerSite** objects as its argument. It then creates instances of the **NonOptimizedRoute**, **OptimizedRoute**, and **Greedy** classes. It also creates a vector to store the selected and sorted retailers and declares variables to store the calculated distance, weight, and profit and a string to store the result.
2. **void costTime(double time): c**alculates the driving time for a driver and prints a message if the driving time is more than 8 hours.

The class variables:

1. Parameter: **vector<RetailerSite> retailers** - a vector of RetailerSite objects that contains information about the retailers, such as their location, weight, and profit.
2. Objects:
   - **NonOptimizedRoute non** - an instance of the NonOptimizedRoute class, used to calculate the non-optimized route's distance, weight, and profit.
   - **OptimizedRoute opt** - an instance of the OptimizedRoute class, used to calculate the optimized route's distance, weight, and profit.
   - **Greedy gre** - an instance of the Greedy class, used to select the top 8 retailers based on cost-effectiveness and calculate the greedy route's distance, weight, and profit.
3. Variables:
   - **double distance** - a variable to store the calculated distance.
   - **double weight** - a variable to store the calculated weight.
   - **double profit** - a variable to store the calculated profit.
   - **string str** - a string variable to store the name of the best route.
   - **string str1** - a string variable used to print messages to the console.
   - **double time** – a variable to represent the estimated driving time for the driver.
4. Vectors:

- **vector<RetailerSite> sorted** - a vector used to store the selected and sorted retailers.
- **vector<RetailerSite> meetReq** - a vector used to store the retailers that meet the minimum requirements for distance and weight.

K. Main.cpp

The class variables:

- **start** and **end**: both of type **time_t**, used to store the start and end time of the program.
- **sites**: a vector of **RetailerSite** objects.
- **non**: an instance of the **NonOptimizedRoute** class.
- **Opt**: an instance of the **OptimizedRoute** class.
- **gre**: an instance of the **Greedy** class.
- **meetReq**: a vector of **RetailerSite** objects that meet the weight requirement.
- **distance**, **weight**: doubles are used to store the calculated distance and weight.
- **bestRoute**: a string used to store the best route with the highest profit.
- **i**: an integer used to store the user's choice of route type.
- **continuenum**: a string used to control the continuation of the program.
- **sorted**: a vector of **RetailerSite** objects used to store the selected and sorted retailers.
- **userInput**: an integer used to store the user's choice between an optimized route and a non-optimized route.

III. EXPERIENCE

Theoretically, this curriculum necessitates a firm grasp of mathematical ideas like cost-effectiveness analysis and profit maximization in addition to programming techniques and data structures. I would need to be familiar with programming concepts like functions, classes, objects, and data structures like vectors, arrays, and linked lists to complete this challenge. In the context of supply chain management, I would also need to be conversant with the ideas of optimization and greedy algorithms, cost-effectiveness analysis, and profit maximization.

To improve the maintainability and scalability of the code, I followed 7 Common Programming Principles, example:

1. **`Separation of Concerns`:** This principle suggests that each module or component of the code should have a single, well-defined responsibility or concern. In the provided code, we can see a clear separation of concerns. Each route gets its class, like **`RetailerSite`, `non-Optimized`, `Optimized` and `Greedy`**. Separating the concerns makes the code easier to understand, modify, and maintain.

2. **`Encapsulation` [1]:** This principle suggests that the internal details of a module or component should be hidden from the outside world, and only a well-defined interface should be exposed.

3. **`Inheritance` [2]:** New classes are produced through the process of inheritance from the current classes. The current class is referred to as the "base class" or

"parent class," and the newly generated class is referred to as the "derived class" or "child class." The term "inherited from the base class" now refers to the derived class. For example, main.cpp only need to inherit `Generate.h` and `Comparison.h` only, since `Comparison.h` already inherits all routes.

4. **`Modularization`:** According to the notion of modularization, the code should be divided into manageable, separate modules or pieces that can be used in many situations. All codes are well-structured according to their functionality and separated into several modules/files.

My completion of this homework has been a worthwhile learning experience because it called for me to use my theoretical understanding to address a practical issue. I got the chance to hone my programming abilities and get experience with various tools, including debugging tools, version control systems, and code editors.

Working on this coursework allowed me to better appreciate the significance of adhering to fundamental programming concepts like separation of concerns, encapsulation, and modularization. It also emphasised the value of developing clear and organised code to enhance the project's long-term maintainability and scalability.

Additionally, this coursework has allowed me to gain a comprehensive understanding of a real-world problem, which has provided me with valuable insights that I can apply in my future studies. By working on this project, I have been able to gain practical experience in solving complex problems using programming, which has improved my critical thinking and problem-solving skills. Furthermore, this project has allowed me to explore new programming concepts and techniques, which will help me to become a more well-rounded and versatile programmer. Overall, completing this coursework has been a rewarding experience that has helped me to grow both personally and professionally.

IV. DIFFICULTIES

One of the main challenges of implementing an optimized or greedy supply chain distribution and transportation system is determining the best solution for the problem at hand. This requires a deep understanding of the company's processes, customer demands, supplier capabilities, and transportation constraints. Additionally, developing an algorithm that can handle large datasets and changing parameters can be difficult.

The process of gathering and processing the data necessary to appropriately display delivery information is another challenge. The system must be able to collect information from a variety of sources, such as vendors, carriers, and inventory control software. After that, the data must be prepared for analysis in order to yield valuable information that will aid the wholesaler in making wise judgements.

Another challenge is making the algorithm scalable and adaptable to the wholesaler's changing needs. Supply chain management systems often require regular updates and modifications to accommodate new products, suppliers, or transportation modes. Therefore, it's crucial to design the system with flexibility and scalability in mind.

To make the code more manageable, it's common to divide it into smaller subfiles. However, this can lead to difficulties in dependency management, header file inclusion, and code organization. It's important to ensure that each subfile has access to the dependencies it requires, includes the correct header files, and is organized in a logical and consistent manner to make it maintainable and easy to understand.

V. CONCLUSION & DISCUSSION

The coursework provided a complex task that required the development of an optimized or greedy supply chain distribution and transportation system. It required a deep understanding of the company's business processes, supplier capabilities, and transportation constraints, as well as the ability to develop algorithms that can handle large datasets and changing parameters while being scalable and adaptable to the company's changing needs. The program was written in C++ and utilized various libraries and algorithms, including non-optimised, optimized, greedy algorithms, cost-effectiveness analysis, and profit maximization, to calculate the best delivery route for a set of retailers based on various criteria.

The program was well-organized and modular, with clear comments and the use of object-oriented programming concepts. Each step of the process was separated into its own function, making the code more readable and easier to maintain. However, a full evaluation of the effectiveness of the algorithms used and the accuracy of the results produced would require further information about the project's requirements. Working on this project also entailed collecting and processing a huge quantity of data in order to accurately present delivery information, which might be a difficult undertaking because the data must be obtained from many sources before being processed and analysed to provide useful insights. Furthermore, separating the code into smaller subfiles may cause issues with dependency management, header file inclusion, and code organisation.

Completing this training gave me practical information and enabled the development of programming skills by applying theoretical concepts to real-world problems. The curriculum gave both simple and difficult assignments that forced me to employ critical thinking and problem-solving skills, as well as a heavy emphasis on precision and efficiency. Despite the obstacles and challenges, finishing this training enhanced my problem-solving abilities and prepared me for future programming assignments.

Overall, the coursework improved my understanding of logistics management and laid the groundwork for additional research into logistics and supply chain management systems. It also gave me experience with a variety of tools, such as code editors, version control systems, and debugging tools, which will be useful in future courses and employment. However, breaking up the code into smaller subfiles may cause issues with dependency management, header file inclusion, and code organisation.

VI. BENEFITS

Overall, completing this coursework has been a fantastic learning experience that has provided me with the knowledge needed to address real-world supply chain management difficulties. I learned how to create and implement optimised or greedy algorithms to efficiently distribute commodities and handle enormous datasets. This study has also increased my understanding of programming fundamentals such as data structures, functions, and loops, which I may apply to future programming projects.

Furthermore, the potential for real-world application of the abilities I've learned in this course is enormous. Companies in a variety of industries rely on efficient supply chain management to be competitive, making my talents relevant and in demand in the employment market. To summarize, completing this training not only broadened my programming knowledge and skills, but also provided me with practical experience in tackling real-world challenges, making me a more well-rounded and adaptable professional.

VII. REFERENCE

1. *Encapsulation in C++* (2023) *GeeksforGeeks*. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/encapsulation-in-cpp/ (Accessed: April 13, 2023).
2. *Inheritance in C++* (2023) *GeeksforGeeks*. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/inheritance-in-c/ (Accessed: April 13, 2023).