C++ code (Sample code for Guidance II on Optimized and Non-Optimized Route Algorithms)

// Below is the **main.cpp** file
// This file defines wasteLocations Class where each instance represents a waste collection pick-
// up point in a fictitious city named City Dragon

```
#include <iostream>
#include <string>
#include <vector>
#include <stdio.h>
#include <stdlib.h>
#include <ctime>
#include <limits>
#include <list>
#include <algorithm>
#include "optimizedRoute.h"
#include "nonOptimizedRoute.h"
#include "printToConsole.h"
#include "writeToFile.h"
using namespace std;

#define INF (std::numeric_limits<float>::max())

int main()
{
   srand((unsigned)time(0));
   vector<wasteLocation> wasteLocations = wasteLocation::initialize_wasteLocation_vector();
   for (int i = 0; i < 8; i++)
   {
      for (int j = 0; j < 8; j++)
      {
         optimizedRoute::shortest_route_matrix[i][j] = j;
      }
   }
   // Floyd-Warshall algorithm to find VALUE DISTANCE  of shortest path between all pairs
// and shortest_route_matrix
   memcpy(&optimizedRoute::floyd_warshall_matrix,  &wasteLocation::map_distance_matrix,
sizeof(wasteLocation::map_distance_matrix));
   for (int a = 0; a < 8; a++)
   {
      for (int i = 0; i < 8; i++)
      {
         for (int j = 0; j < 8; j++)
         {
            // if current shortest path from i to j > path from i to j via intermediate node a; update
//Floyd_Warshall_matrix & optimizedRoute::shortest_route_matrix; else continue
            if                    (optimizedRoute::floyd_warshall_matrix[i][j]                    >
optimizedRoute::floyd_warshall_matrix[i][a] + optimizedRoute::floyd_warshall_matrix[a][j])
            {
               optimizedRoute::floyd_warshall_matrix[i][j]                                        =
optimizedRoute::floyd_warshall_matrix[i][a] + optimizedRoute::floyd_warshall_matrix[a][j];
               // Find path involved in shortest path between all pairs
               optimizedRoute::shortest_route_matrix[i][j]                                        =
optimizedRoute::shortest_route_matrix[i][a];
```

```
            }
        }
    }
}
optimizedRoute route_40(40, wasteLocations);
optimizedRoute route_60(60, wasteLocations);
nonOptimizedRoute route_default;
int program_choice;
int exit_program = 0; // exit or continue program and int 1 == exit, int 0 == continue
printMap();
while (exit_program == 0)
{
 cout << "\nWhat would you like to do? \n Enter integer to choose program \n"<< endl;
cout << " 1 - Display Map of City Dragon \n 2 - Display Cost of Operation of Non-Optimized Default Route (Collect From All Locations Despite Waste Level) \n 3 - Display Cost of Operation of Optimized Route (Collect @ 40% Waste Level) \n 4 - Display Cost of Operation of Optimized Route (Collect @ 60% Waste Level) \n 5 - Export full report as 'wasteCollection_costOfOperation_*.txt' file\n 6 - For developer use \n\n \n";
    cin >> program_choice;

    while (cin.fail() || program_choice < 1 || program_choice > 6)
    {
        // check if input type is compatible AND within range
        {
            cout << "Invalid input. Please enter an integer from 1-6.\n";
        }
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
        cin >> program_choice;
    }
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
    switch (program_choice)
    {
    case 1:
    {
        cout <<" 1 - Display Map of City Dragon" << endl << endl;
        printMap();
        break;
    }
    case 2:
    {
   cout << "2 - Display Cost of Operation of Non-Optimized Default Route" << endl << endl;
        print_nonOptimizedRoute(route_default);
        break;
    }
    case 3:
    {
  cout << "3. Display Cost of Operation of Optimized Route (40% Waste Level)" << endl<< endl;
        print_optimizedRoute(wasteLocations, route_40);
        break;
    }
    case 4:
    {
cout << "4 - Display Cost of Operation of Optimized Route (60% Waste Level)" << endl<< endl;
```

```
            print_optimizedRoute(wasteLocations, route_60);
            break;
        }
        case 5:
        {
 cout << "5 - Export full report as 'wasteCollection_costOfOperation_*.txt' file" << endl<<endl;
            save_to_report(route_40, route_60, route_default, wasteLocations);
 cout << "File is successfully exported. Pls check your current working directory" << endl<<endl;
            break;
        }
        case 6:
        {
            cout << "FOR DEVELOPER USE: " << endl;
            printMap();
            print_map_distance_matrix();
            print_shortest_route_matrix();
            print_floyd_warshall_matrix();
        }
        }
            cout << endl<< endl
                << "Do you want to continue? \n Enter '0' for YES, '1' for NO. \n ";
            cin >> exit_program;
            while (cin.fail() || exit_program < 0 || exit_program > 1)
            {
                // check if input type is compatible AND within range
                {
                    cout << "Invalid input. Please enter 0 to Continue Program, 1 to exit program.\n";
                }
                cin.clear();              // always clear flag to return to 'normal' operation mode after input
                cin.ignore(numeric_limits<streamsize>::max(), '\n'); // always remove all input
                cin >> exit_program;
            }
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            // check input
            // ignore input
            // repeat if wrong input
        }
}
```

// Below is the **nonOptimizedRoute.h** file

```
#ifndef NONOPTIMIZEDROUTE_H_INCLUDED
#define NONOPTIMIZEDROUTE_H_INCLUDED
// This file defines unOptimizedRoute Class where all pick-up locations are visited on a
//default circular route (drive straight back & fro from waste disposal station)
#include <iostream>
#include <string>
#include <vector>
#include <stdio.h>
#include <stdlib.h>
#include <ctime>
#include <cmath>
#include <map>
```

```
#include <limits>
#include <algorithm>
#include <tuple>
#include "wasteLocation.h"
using namespace std;

#ifndef NONOPTIMIZED_ROUTE_H
#define NONOPTIMIZED_ROUTE_H
#define INF (std::numeric_limits<float>::max())
/* This class represents a nonOptimizedRoute which travels across all wasteLocation points */
class nonOptimizedRoute{
    // Private members in optimizedRoute class
    vector<int> final_route = {0,1,3,2,4,3,6,5,6,7,0};
    vector<float> individual_route_distance;
    // Holds distance between each travelled wasteLocations
    float sum_dist = 0;
    float time_taken = 0;         // Total time taken to complete route (min), 1.5 min per km
   float fuel_consumption = 0;
 // Total duel used in waste collection route (RM), RM 1.5 per km
    float wage = 0;                                  // Wage if truck driver (RM), RM5.77 per hour
    float total_cost;                                                  // Fuel + Wage (RM)
public:
nonOptimizedRoute(){
    // calculate distance between each wasteLocation visited
        for (auto it = final_route.begin(); next(it, 1) != final_route.end(); ++it)
        {
individual_route_distance.push_back(wasteLocation::map_distance_matrix[*it][*(next(it, 1))]);
        }
    for (auto it = individual_route_distance.begin(); it != individual_route_distance.end(); ++it)
        {
            sum_dist += *it;
        }
        time_taken = ceil(1.5 * sum_dist * 100.0) / 100.0;
        fuel_consumption = ceil(1.5 * sum_dist * 100.0) / 100.0;
        wage = ceil(1.5 * (time_taken / 60) * 100.0) / 100.0;
        total_cost = fuel_consumption + wage;
    }
    /* Defining public getter & setter function for PRIVATE attributes*/
    vector<int> getFinal_route() {
            return this->final_route;
    }
    vector<float> getIndividual_route_distance() {
            return this->individual_route_distance;
    }
    float getSum_dist() {
            return this->sum_dist;
    }
    float getTime_taken() {
```

```
        return this->time_taken;
    }
    float getFuel_consumption() {
        return this->fuel_consumption;
    }
    float getWage() {
        return this->wage;
    }
    float getTotal_cost()
    {
       return this->total_cost;
    }
};
#endif
#endif // NONOPTIMIZEDROUTE_H_INCLUDED
```

// Below is the **optimizedRoute.h** file

```
#ifndef OPTIMIZEDROUTE_H_INCLUDED
#define OPTIMIZEDROUTE_H_INCLUDED

// This file defines the optimizedRoute Class where pick-up locations are determined
//wasteLevel and the pick-up route is minimized using Floyd-Warshall Algorithm
#include <iostream>
#include <string>
#include <vector>
#include <stdio.h>
#include <stdlib.h>
#include <ctime>
#include <cmath>
#include <map>
#include <limits>
#include <algorithm>
#include <tuple>
#include "wasteLocation.h"
using namespace std;

#ifndef OPTIMIZED_ROUTE_H
#define OPTIMIZED_ROUTE_H
#define INF (std::numeric_limits<float>::max())

/* This class represents contains information of the max wasteLevel before waste should
be collected, a vector of all destinations that >= wasteLevel.
   Both information are used to create an optimized waste collection route using Floyd-
Warshall + Greedy Algorithm.
   NOTE: Acceptable waste level must be given upon instantiation*/
class optimizedRoute
{
   float max_wasteLevel_for_collection;                // wasteLocations with wasteLevel
//above 'max_wasteLevel_for_collection' will be listed as a pick-up location
```

```
    vector<int> filtered_dest = {0};                // Holds unique code of each wasteLocation
//that is listed as pick-up location; (code = index in wasteLocation::map_distance_matrix)
    vector<int> final_route = {0};              // Holds unique code of all wasteLocations that
//shortest-path collection route will pass through in order
    vector<float> individual_route_distance = {0};
    float sum_dist = 0;                             // Sum of 'individual_route_distance' (km)
    float time_taken = 0;          // Total time taken to complete route (min), 1.5 min per km
float fuel_consumption = 0;      // Total duel used in waste collection route (RM), RM 1.5
//per km
    float wage = 0;                 // Wage if truck driver (RM), RM5.77 per hour
    float total_cost;                               // Fuel + Wage (RM)
    bool pick_up_required = true
public:
    // Shortest-routes matrix R to store intermediate INT ID of nodes between shortest path
//to each wasteLocation pair
    static int shortest_route_matrix[8][8];
    // Floyd-Warshall matrix to store a FLOAT value representing distance of shortest path
between each wasteLocation pair.
    // Since all optimizedRoute will be planned on the same city map, all optimizedRoute
instance can share the same 'floyd_warshall_matrix'
    static float floyd_warshall_matrix[8][8];
    /* Public Constructor: Waste level must be given upon instantiation*/
public:
    optimizedRoute(float i, vector<wasteLocation> wasteLocations)
    {
        max_wasteLevel_for_collection = round(i);
        filtered_dest = filter_dest_by_wasteLevel(wasteLocations);
        if (filtered_dest.empty()){
            pick_up_required = false;
        }
        else{
        calculate_all_cost();
        }
    }
    /* Defining public getter & setter function for PRIVATE attributes*/
public:
    float getMax_wasteLevel_for_collection()
    {
        return this->max_wasteLevel_for_collection;
    }
    void setMax_wasteLevel_for_collection(float max_wasteLevel_for_collection)
    {
        this->max_wasteLevel_for_collection = max_wasteLevel_for_collection;
    }
    vector<int> getFiltered_dest()
    {
        return this->filtered_dest;
    }
    vector<int> getFinal_route()
    {
```

```
        return this->final_route;
    }
    vector<float> getIndividual_route_distance()
    {
        return this->individual_route_distance;
    }
    float getSum_dist()
    {
        return this->sum_dist;
    }
    float getTime_taken()
    {
        return this->time_taken;
    }
    float getFuel_consumption()
    {
        return this->fuel_consumption;
    }

    float getWage()
    {
        return this->wage;
    }
    float getTotal_cost()
    {
        return this->total_cost;
    }
    bool getPick_up_required()
    {
        return this->pick_up_required;
    }
    // Return an INT vector representing ID of wasteLocation which has wasteLevel >=
//max_wasteLevel_for_collection (i.e. wasteLocation requires waste collection service)
    vector<int> filter_dest_by_wasteLevel(vector<wasteLocation> wasteLocations)
    {
        vector<int> filtered_dest;
        for (auto it = wasteLocations.begin(); it != wasteLocations.end(); ++it)
        {
            if (it->getWasteLevel() >= max_wasteLevel_for_collection)

{       filtered_dest.push_back(wasteLocation::dict_Name_toId[it->getWasteLocation_n
ame()]);
            };
        }
        return filtered_dest;
    }
    vector<int> path_reconstruction(int start, int end, int matrix[8][8])
    {
        vector<int> path;
        while (start != end)
```

```
        {
            path.push_back(start);
            start = matrix[start][end];
        }
        path.push_back(end);
        return path;
    }
    /* Generate full route for all destinations
    Takes in 3 parameter: an INT vector of filtered_dest_id, floyd_warshall_matrix and shortest_route_matrix */
    vector<int> generateFullRoute(vector<int> filtered_dest_id, float floyd_warshall_matrix[8][8], int shortest_route_matrix[8][8])
    {
        // Vector which holds all intermediate stops to reach all destinations in a full trip
        vector<int> final_route;
        // 1 - Select first node to travel to based on lowest; Greedy algorithm - node choosen //based on lowest floyd-warshall route value
        // Representing ID of destination from filtered_dest_id
        int id_choosen_dest = filtered_dest_id.at(0);
        for (int i = 0; i < filtered_dest_id.size(); i++)
        {
            if (floyd_warshall_matrix[0][i] < floyd_warshall_matrix[0][id_choosen_dest])
            {
                // find INDEX of next destination in filtered_dest_id
                id_choosen_dest = i;
            }
        }
        id_choosen_dest = filtered_dest_id[id_choosen_dest];
// access INDEX of id_choosen_dest array to find actual destination
        // 2 - Find intermediate path from source of origin "Station" to 1st destination/stop
        vector<int>path1 = path_reconstruction(0, id_choosen_dest, shortest_route_matrix);
        // 3 - Remove from travelled node from "Destination" list
        // Use erase() if vector has element > 1, erase() fucntion returns ;  else clear
        if (filtered_dest_id.size() > 1)
        {
 filtered_dest_id.erase(find(filtered_dest_id.begin(),filtered_dest_id.end(), id_choosen_dest));
        }
        else
        {
            filtered_dest_id.clear();
        }
        // 4 - Check if found path include other points in "Destinations"; If yes, remove location node from filtered_dest_id
        for (auto it = path1.begin(); it != path1.end(); ++it)
        {
            // find() function is not used since find() returns pointer to last element if not found; //might clash with situation where destination node is the last element in filtered_dest_id
            if (count(filtered_dest_id.begin(), filtered_dest_id.end(), *it) > 0)
            {
            filtered_dest_id.erase(find(filtered_dest_id.begin(), filtered_dest_id.end(), *it));
```

```
        }
    }
    // 5 - Concat found route to vector of final route at the end of final_route
    final_route.insert(std::end(final_route), std::begin(path1), std::end(path1));
    // 6 - Repeat while filtered_dest_id vector is not empty
    while (!filtered_dest_id.empty())
    {
        // 1 - Select first node to travel to based on lowest; Greedy algorithm - node chosen based on lowest Floyd-Warshall route value
        int prev_dest_id = id_choosen_dest;
        id_choosen_dest = filtered_dest_id.at(0);
        for (int i = 0; i < filtered_dest_id.size(); i++)
        {
            // find INDEX of next destination in filtered_dest_id
            if (floyd_warshall_matrix[prev_dest_id][i] < floyd_warshall_matrix[prev_dest_id][id_choosen_dest])
            {
                id_choosen_dest = i;
            }
        }
        id_choosen_dest = filtered_dest_id[id_choosen_dest];
// access INDEX of id_choosen_dest array to find actual destination
        // 2 - Find intermediate path from previous destination/stop as new source of origin
//to next destination/stop
 vector<int> path1 = path_reconstruction(prev_dest_id, id_choosen_dest, shortest_route_matrix);
        path1.erase(path1.begin());
 // remove first , first node was included from previous route
        // 3 - Remove from travelled node from "Destination" list
        if (filtered_dest_id.size() > 1)
        {
            filtered_dest_id.erase(find(filtered_dest_id.begin(), filtered_dest_id.end(), id_choosen_dest));
        }
        else
        {
            filtered_dest_id.clear();
        }
        // 4 - Check if found path include other points in "Destinations"; If yes, remove
//location node from filtered_dest_id
        for (auto it = path1.begin(); it != path1.end(); ++it)
        {
       // find() function is not used since find() returns pointer to last element if not found;
            // might clash with situation where destination node is the last element in
//filtered_dest_id
            if (count(filtered_dest_id.begin(), filtered_dest_id.end(), *it) > 0)
            {
         filtered_dest_id.erase(find(filtered_dest_id.begin(), filtered_dest_id.end(), *it));
            }
        }
```

```
            // 5 - Concat found route to vector of final route at the end of final_route
            final_route.insert(std::end(final_route), std::begin(path1), std::end(path1));
        }
        // At final stop, return to station
        // 2 - Find intermediate path from source of origin "Station" to 1st destination / stop
 vector<int>return_path=path_reconstruction(id_choosen_dest,0, shortest_route_matrix);
        return_path.erase(return_path.begin());
// remove first , first node was included from previous route
        // 5 - Concat found route to vector of final route at the end of final_route
final_route.insert(std::end(final_route), std::begin(return_path), std::end(return_path));
 return final_route;
    }
    /* Find distance of each individual path
        Takes in 2 parameter: an INT vector representing the full route of trip & a
floyd_warshall_matrix*/
    vector<float> all_individual_path_distance(vector<int> final_route, float
floyd_warshall_matrix[8][8])
    {
        vector<float> individual_route_distance;
        for (auto it = final_route.begin(); next(it, 1) != final_route.end(); ++it)
        {
            individual_route_distance.push_back(floyd_warshall_matrix[*it][*(next(it, 1))]);
        }
        return individual_route_distance;
    }
    /* Sum up all distance of individual path
        Takes in 1 parameter:  */
    float sum_distance(vector<float> individual_route_distance)
    {
        float sum_dist;
        for (auto it = individual_route_distance.begin(); it !=
individual_route_distance.end(); ++it)
        {
            sum_dist += *it;
        }
        return sum_dist;
    }
    // This function runs all atomic operation functions in order to calculate all cost factors
    void calculate_all_cost()
    {
      final_route = generateFullRoute(filtered_dest,
optimizedRoute::floyd_warshall_matrix, optimizedRoute::shortest_route_matrix);
        individual_route_distance = all_individual_path_distance(final_route,
optimizedRoute::floyd_warshall_matrix);
        sum_dist = sum_distance(individual_route_distance);
        time_taken = ceil(1.5 * sum_dist * 100.0) / 100.0;
        fuel_consumption = ceil(1.5 * sum_dist * 100.0) / 100.0;
        wage = ceil(1.5 * (time_taken / 60) * 100.0) / 100.0;
        total_cost = fuel_consumption + wage;
    }
```

```
};
int optimizedRoute::shortest_route_matrix[8][8];
float optimizedRoute::floyd_warshall_matrix[8][8];
#endif
// vector<int> final_route = generateFullRoute({1, 2, 6}, optimizedRoute::floyd_warshall_matrix, optimizedRoute::shortest_route_matrix);

// for (auto it = final_route.begin(); it != final_route.end(); ++it)
// {
//     std::cout << *it << "\n";
// }

// cout << endl
//      << endl
//      << "Route distance: " << endl;
// // convert route to distance
// vector<float> individual_route_distance = all_individual_path_distance(final_route, optimizedRoute::floyd_warshall_matrix);
// for (auto it = individual_route_distance.begin(); it != individual_route_distance.end(); ++it)
// {
//     std::cout << *it << "\n";
// }
// // find total distance
// float sum_dist = sum_distance(individual_route_distance);

// int main()
// {
//                                     vector<wasteLocation> wasteLocations = wasteLocation::initialize_wasteLocation_vector();
//     for (auto it = wasteLocations.begin(); it != wasteLocations.end(); ++it)
//     {
//         cout << "WasteLocation: " << it->getWasteLocation_name() << " || "
//              << "Waste Level: " << it->getWasteLevel() << endl;
//     }
//     optimizedRoute route(40, wasteLocations);
//     vector<int> paths = route.getFiltered_dest();
//     for (auto it = paths.begin(); it != paths.end(); ++it)
//     {
//         cout << "WasteLocation: " << *it << endl;
//     }
// }
#endif // OPTIMIZEDROUTE_H_INCLUDED
```

// Below is the **wasteLocation.h** file

```
#ifndef WASTELOCATION_H_INCLUDED
#define WASTELOCATION_H_INCLUDED

// This file defines a wasteLocation Class which represents each waste collection pick-up point within the city=
```

```
#include <iostream>
#include <string>
#include <vector>
#include <stdio.h>
#include <stdlib.h>
#include <ctime>
#include <cmath>
#include <map>
#include <limits>
#include <algorithm>
using namespace std;

#ifndef WASTE_LOCATIONS_H
#define WASTE_LOCATIONS_H

#define INF (std::numeric_limits<float>::max())
/* This class represents the waste pick-up locations in the city */
class wasteLocation
{
   /* Private members in wasteLocation class */
   const float wasteLevel = (std::round(rand() % 100)); // Waste level is assigned randomly every time system runs; Set as constant - Reading is given by sensor, manual tampering is not allowed
   string wasteLocation_name;
 // A string to represent pick-up location's name
   /* Public Constructor: Waste level must be given upon instantiation*/
public:
   wasteLocation(string name)
   {
      wasteLocation_name = name;
   }
   /* Static public variables that is shared across all instances of wasteLocation class*/
public:
   // 'map_distance_matrix' as static information to be shared across all instance of wasteLocation class;
   // Size = 8x8 since there are total of 7 wasteLocation + 1 waste disposal station;
   //
map_distance_matrix[wasteLocationID_from_origin][wasteLocationID_to_destination] = distance between origin to destination
   inline const static float map_distance_matrix[8][8] = {0, 3, INF, INF, INF, INF, INF, 4, 3, 0, INF, 6, INF, INF, INF, INF, INF, INF, 0, 5, 4, INF, INF, INF, INF, 6, 5, 0, 2, INF, 2, INF, INF, INF, 4, 2, 0, INF, INF, INF, INF, INF, INF, INF, INF, 0, 7, INF, INF, INF, INF, 2, INF, 7, 0, 3, 4, INF, INF, INF, INF, INF, 3, 0};
   // Dictionary to store mapping of index of wasteLocation in vector
   // index in vector represents unique numerical code of wasteLocation within city
   // Key = wasteLocation_name string， Value = wasteLocation index in vector
   inline static map<string, int>
dict_Name_toId ={{"Station",0},{"A",1},{"B", 2},{"C", 3},{"D", 4},{"E", 5},{"F",6},{"G", 7}};
```

```
    // Dictionary to store mapping of wasteLocation_name to index of wasteLocation in vector
    //  Key = wasteLocation index in vector, Value = WasteLocation_name string
    inline static map<int, string> dict_Id_to_Name = {{0, "Station"},
                                                      {1, "A"},
                                                      {2, "B"},
                                                      {3, "C"},
                                                      {4, "D"},
                                                      {5, "E"},
                                                      {6, "F"},
                                                      {7, "G"}};
    /* Defining public getter & setter function for PRIVATE attributes*/
    // NOTE: wasteLevel is automatically assigned upon initialisation, no setter method
public:
    float getWasteLevel()
    {
        return this->wasteLevel;
    }
    string getWasteLocation_name()
    {
        return this->wasteLocation_name;
    }
    void setWasteLocation_name(string wasteLocation_name)
    {
        this->wasteLocation_name = wasteLocation_name;
    }
    // This function is used to initialise a vector of all wasteLocations within the city at the
//beginning of program
    static vector<wasteLocation> initialize_wasteLocation_vector()
    {
        vector<wasteLocation> wasteLocations;
        wasteLocations.push_back(wasteLocation("A"));
        wasteLocations.push_back(wasteLocation("B"));
        wasteLocations.push_back(wasteLocation("C"));
        wasteLocations.push_back(wasteLocation("D"));
        wasteLocations.push_back(wasteLocation("E"));
        wasteLocations.push_back(wasteLocation("F"));
        wasteLocations.push_back(wasteLocation("G"));
        return wasteLocations;
    }
};
#endif
// int main()
// {
//     vector<wasteLocation> wasteLocations= initialize_wasteLocation_vector();
//     for (auto it = wasteLocations.begin(); it != wasteLocations.end(); ++it)
//     {
//         cout << wasteLocation::dict_Name_toId[it->getWasteLocation_name()] << endl;
//     }
// }
```

```
#endif // WASTELOCATION_H_INCLUDED
```

// Below is the **printToConsole.h** file

```
#ifndef PRINTTOCONSOLE_H_INCLUDED
#define PRINTTOCONSOLE_H_INCLUDED
// This file contains all general method used for printing information onto the console
#include "optimizedRoute.h"
#include "nonOptimizedRoute.h"
#ifndef PRINT_TO_CONSOLE_H
#define PRINT_TO_CONSOLE_H

void print_optimizedRoute(vector<wasteLocation> wasteLocations, optimizedRoute route)
{
   // Print all wasteLocation & their waste level
   cout << "Waste Level at each wasteLocation: " << endl;
   for (auto it = wasteLocations.begin(); it != wasteLocations.end(); ++it)
   {
      cout << "WasteLocation: " << it->getWasteLocation_name() << " || "
          << "Waste Level: " << it->getWasteLevel() << endl;
   }
   std::cout << endl
           << endl;
   if (route.getPick_up_required())
   {
      // Print filtered wasteLocation
      cout << "WasteLocation that requires pick up service (>=" << route.getMax_wasteLevel_for_collection() << "%)" << endl;
      vector<int> filtered_wasteLocations = route.getFiltered_dest();
      for (auto it = filtered_wasteLocations.begin(); it != filtered_wasteLocations.end(); ++it)
      {
         cout << "WasteLocation: " << wasteLocations[*it - 1].getWasteLocation_name() << " || " << "Waste Level: " << wasteLocations[*it - 1].getWasteLevel() << endl;
      }
      std::cout << endl << endl;

      // Print chosen path
      cout << "Pick-up route choosen: " << endl;
      vector<int> finalRoute = route.getFinal_route();
      for (auto it = finalRoute.begin(); it != finalRoute.end(); ++it)
      {
         cout << wasteLocation::dict_Id_to_Name[*it];
         if (next(it, 1) != finalRoute.end())
         {
            cout << " -> ";
         }
      }
      std::cout << endl << endl;
      // Print cost of each path
```

```
        vector<float> individual_dist = route.getIndividual_route_distance();
        cout << "Distance between each wasteLocation choosen: " << endl;
        for (auto it = individual_dist.begin(); it != individual_dist.end(); ++it)
        {
           cout << *it;
           if (next(it, 1) != individual_dist.end())
           {
              cout << " -> ";
           }
        }
        std::cout<< endl << endl;
        // Print all cost
        cout << "Total Distance (km): " << route.getSum_dist() << endl;
    cout << "Time taken (Driving speed @ 1.5 min per km): " << route.getTime_taken()
<< endl;
    cout << "Total Fuel Consumption (RM 1.5 per km): " << route.getFuel_consumption() << endl;
        cout << "Wage of Driver (RM 5.77 per hr): " << route.getWage() << endl;
        cout << "Total  Cost of Operation (RM):" << route.getTotal_cost() << endl;

        std::cout
           << endl
           << endl;
        cout                                                                    <<
"_____________________________________________________________________________"
<< endl<< endl;
    }
    else
    {
        cout << "No wasteLocation with wasteLevel above " <<
route.getMax_wasteLevel_for_collection() << "% \n No pick-up service required" <<
endl;
        std::cout
           << endl
           << endl;

        cout                                                                    <<
"_____________________________________________________________________________"
<< endl<< endl;
    }
};
void print_nonOptimizedRoute(nonOptimizedRoute route)
{
    cout << "Non-optimized route covers all the wasteLocation according to the highway
road. \n All wasteLocations will be visited despite wasteLevel" << endl
         << endl;
    // Print choosen path
    cout << "Default Pick-up route : " << endl;
    vector<int> finalRoute = route.getFinal_route();
    for (auto it = finalRoute.begin(); it != finalRoute.end(); ++it)
    {
```

```
        cout << wasteLocation::dict_Id_to_Name[*it];
        if (next(it, 1) != finalRoute.end())
        {
            cout << " -> ";
        }
    }
    std::cout
        << endl
        << endl;
    // Print cost of each path
    vector<float> individual_dist = route.getIndividual_route_distance();
    cout << "Distance spent between each wasteLocation choosen: " << endl;
    for (auto it = individual_dist.begin(); it != individual_dist.end(); ++it)
    {
        cout << *it;
        if (next(it, 1) != individual_dist.end())
        {
            cout << " -> ";
        }
    }
    std::cout
        << endl
        << endl;
    // Print all cost
    cout << "Total Distance (km): " << route.getSum_dist() << endl;
    cout << "Time taken (Driving speed @ 1.5 min per km): " << route.getTime_taken()
<< endl;
    cout << "Total Fuel Consumption (RM 1.5 per km): " << route.getFuel_consumption()
<< endl;
    cout << "Wage of Driver (RM 5.77 per hr): " << route.getWage() << endl;
    cout << "Total  Cost of Operation (RM):" << route.getTotal_cost() << endl;
    std::cout
        << endl
        << endl;
    cout                                                                    <<
"                                                                                "
<< endl
         << endl;
}

void print_floyd_warshall_matrix()
{
    cout << "Floyd-Warshall Matrix: " << endl;
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            cout << optimizedRoute::floyd_warshall_matrix[i][j] << "   ";
        }
        cout << endl;
```

```
    }
    std::cout
        << endl
        << endl;
    cout                                                                    <<
"                                                                           "
<< endl  << endl;
};
void print_map_distance_matrix(){
    cout << "Distance Matrix (between each wasteLocation pair): " << endl;
    cout << "Note: 3.40282e+38 = INF = no direct distance between a certain
wasteLocation pair" << endl;
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            cout << wasteLocation::map_distance_matrix[i][j] << "   ";
        }
        cout << endl;
    }
    std::cout
        << endl
        << endl;

    cout                                                                    <<
"                                                                           "
<< endl
        << endl;
}
void print_shortest_route_matrix()
{
    cout << "Shortest Route Matrix: " << endl;
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            cout << optimizedRoute::shortest_route_matrix[i][j] << "   ";
        }
        cout << endl;
    }
    std::cout
        << endl
        << endl;

    cout                                                                    <<
"                                                                           "
<< endl
        << endl;
};
void printMap()
```

```
{
   cout << endl
        << "Map of City X: " << endl
        << endl;
   cout << "Station ------------3-------------A" << endl;
   cout << "  |                              |" << endl;
   cout << "  |                              |" << endl;
   cout << "  |                         |" << endl;
   cout << "  4                                6" << endl;
   cout << "  |                              |" << endl;
   cout << "  |                              |" << endl;
   cout << "  |                              |" << endl;
   cout << "G ----3-----F ----2----C ----5----B" << endl;
   cout << "                |              |" << endl;
   cout << "                |              |" << endl;
   cout << "                |              |" << endl;
   cout << "                2----D ----4" << endl;

   std::cout<< endl << endl;

   cout                                                                                    <<
"                                                                                                  "
<< endl<< endl;
}
#endif // PRINTTOCONSOLE_H_INCLUDED
#endif
```

// Below is the **writeToFile.h** file

```
#ifndef WRITETOFILE_H_INCLUDED
#define WRITETOFILE_H_INCLUDED
// This file contains general method used for writing report and save as external file
#include "optimizedRoute.h"
#include "nonOptimizedRoute.h"
#include <iostream>
#include <fstream>
#include <ctime>
using namespace std;

#ifndef WRITE_FILE_H
#define WRITE_FILE_H
void write_optimizedRoute(vector<wasteLocation> wasteLocations, optimizedRoute
route, ofstream& file)
{
   // Print all wasteLocation & their waste level
   file << "Waste Level at each wasteLocation: \n";
   for (auto it = wasteLocations.begin(); it != wasteLocations.end(); ++it)
   {
      file << "WasteLocation: " << it->getWasteLocation_name() << " || "
           << "Waste Level: " << it->getWasteLevel() << "\n";
   }
```

```
    file << "\n"
         << "\n";
    if (route.getPick_up_required())
    {
        // Print filtered wasteLocation
        file << "WasteLocation that requires pick up service (>=" << route.getMax_wasteLevel_for_collection() << "%)\n";
        vector<int> filtered_wasteLocations = route.getFiltered_dest();
    for (auto it = filtered_wasteLocations.begin(); it != filtered_wasteLocations.end(); ++it)
        {
file << "WasteLocation: " << wasteLocations[*it - 1].getWasteLocation_name() << " || "
                << "Waste Level: " << wasteLocations[*it - 1].getWasteLevel() << "\n";
        }
        file << "\n"
             << "\n";
        // Print the chosen path
        file << "Pick-up route choosen: \n";
        vector<int> finalRoute = route.getFinal_route();
        for (auto it = finalRoute.begin(); it != finalRoute.end(); ++it)
        {
            file << wasteLocation::dict_Id_to_Name[*it];
            if (next(it, 1) != finalRoute.end())
            {
                file << " -> ";
            }
        }
        file
            << "\n"
            << "\n";
        // Print cost of each path
        vector<float> individual_dist = route.getIndividual_route_distance();
        file << "Distance between each wasteLocation choosen: \n";
        for (auto it = individual_dist.begin(); it != individual_dist.end(); ++it)
        {
            file << *it;
            if (next(it, 1) != individual_dist.end())
            {
                file << " -> ";
            }
        }
        file<< "\n" << "\n";
        // Print all cost
        file << "Total Distance (km): " << route.getSum_dist() << "\n";
        file << "Time taken (Driving speed @ 1.5 min per km): " << route.getTime_taken()
<< "\n";
        file << "Total Fuel Consumption (RM 1.5 per km): " << route.getFuel_consumption()
<< "\n";
        file << "Wage of Driver (RM 5.77 per hr): " << route.getWage() << "\n";
        file << "Total  Cost of Operation (RM):" << route.getTotal_cost() << "\n";
```

```
        file
            << "\n"
            << "\n";
        file                                                                    <<
"_____\
n"
             << "\n";
    }
    else
    {
        file << "No wasteLocation with wasteLevel above " <<
route.getMax_wasteLevel_for_collection() << "% \n No pick-up service required\n";
        file
            << "\n"
            << "\n";

        file                                                                    <<
"_____\
n"
             << "\n";
    }
};
void write_nonOptimizedRoute(nonOptimizedRoute route, ofstream& file)
{
    file << "Non-optimized route covers all the wasteLocation according to the highway
road. \n All wasteLocations will be visited despite wasteLevel\n"<< "\n";
    // Print chosen path
    file << "Default Pick-up route : \n";
    vector<int> finalRoute = route.getFinal_route();
    for (auto it = finalRoute.begin(); it != finalRoute.end(); ++it)
    {
        file << wasteLocation::dict_Id_to_Name[*it];
        if (next(it, 1) != finalRoute.end())
        {
            file << " -> ";
        }
    }
    file << "\n" << "\n";
    // Print cost of each path
    vector<float> individual_dist = route.getIndividual_route_distance();
    file << "Distance spent between each wasteLocation choosen: \n";
    for (auto it = individual_dist.begin(); it != individual_dist.end(); ++it)
    {
        file << *it;
        if (next(it, 1) != individual_dist.end())
        {
            file << " -> ";
        }
    }
    file<< "\n"<< "\n";
```

```
    // Print all cost
    file << "Total Distance (km): " << route.getSum_dist() << "\n";
    file << "Time taken (Driving speed @ 1.5 min per km): " << route.getTime_taken() <<
"\n";
    file << "Total Fuel Consumption (RM 1.5 per km): " << route.getFuel_consumption()
<< "\n";
    file << "Wage of Driver (RM 5.77 per hr): " << route.getWage() << "\n";
    file << "Total  Cost of Operation (RM):" << route.getTotal_cost() << "\n";

    file
        << "\n"
        << "\n";

    file                                                                                <<
"_____\
n"
         << "\n";
}
void write_floyd_warshall_matrix(ofstream& file)
{
    file << "Floyd-Warshall Matrix: \n";
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            file << optimizedRoute::shortest_route_matrix[i][j] << "   ";
        }
        file << "\n";
    }

    file
        << "\n"
        << "\n";

    file                                                                                <<
"_____\
n"
         << "\n";
};

void writeMap(ofstream& file)
{
    file << "\n"
         << "Map of City X: \n"
         << "\n";
    file << "Station ----------3----------A\n";
    file << "  |                          |\n";
    file << "  |                          |\n";
    file << "  |                    |\n";
    file << "  4                          6\n";
```

```
    file << "  |                              |\n";
    file << "  |                              |\n";
    file << "  |                              |\n";
    file << "G ----3-----F ----2----C ----5----B\n";
    file << "                  |             |\n";
    file << "                  |             |\n";
    file << "                  |             |\n";
    file << "                  2----D ----4\n";

    file
        << "\n"
        << "\n";

    file                                                                    <<
"_____\
n"
         << "\n";
}

void save_to_report(optimizedRoute route_40, optimizedRoute route_60,
nonOptimizedRoute route_default, vector<wasteLocation> wasteLocations)
{
    ofstream file;
    char timeString[100];
    time_t curr_time;
    tm *curr_tm;
    time(&curr_time);
    curr_tm = localtime(&curr_time);

strftime(timeString,100,"wasteCollection_costOfOperation_%y%d%m%H%M%S.txt",
curr_tm);
    file.open(timeString);
    writeMap(file);
    file << "DEFAULT ROUTE" << "\n";
    write_nonOptimizedRoute(route_default, file);
    file << "COLLECT @ 40% WASTE LEVEL" << "\n";
    write_optimizedRoute(wasteLocations, route_40, file);
    file << "COLLECT @ 60% WASTE LEVEL" << "\n";
    write_optimizedRoute(wasteLocations, route_60, file);
    file.close();
}
#endif
#endif // WRITETOFILE_H_INCLUDED
```

// Below is the **output.cpp** file

```
#include <iostream>
#include <string>
#include <vector>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <ctime>
#include <cmath>
#include <map>
using namespace std;

// This function is used to print the optimisedRoute found on console within app
void print_summary_console(){

}

// This function is used to output the full summary file as a .txt file with filename
//'wasteCollectionRoute_report_{time}.txt'
void save_as_report(){

}
```

---------- End of **CW2** Sample Code Guidance II ----------