Department of Computer Engineering

BLG 336E
Analysis of Algorithm II
Homework 2
Report

ID	*Name*	*Surname*
150140126	*Emre*	*Reyhanlıoğlu*

PART 1 – PROJECT ARCHITECTURE

There are many architectural design of software development area. In this project, there are many virtual objects to simulate the project. Also there are many operations to perform on these objects to simulate the travels without any intersections. For these reasons, I used “Clean Architecture” and “SOLID(Single Responsibility, Open Closed, Liskov's Substitution, Interface Segregation and Dependency Inversion) principles whenever it's possible, because I believe that good architecture makes the project more maintainable and testable.

With these architectural design, my prject has 3 layers: Data layer, service layer and presentation layer.

- Data layer has all of the data and data provider classes and necessary methods inside it.
- Service layer has all the business logic of the project and it provides data to be used for the simulation.
- Presentation layer shows the simulation results by using the provided services.

a) Data Layer

Data layer of the project has data classes and a file reader class which hold the necessary informations in the project. There are 3 classes in this layer and I will explain them shortly.

1) Path Class

In my project, there is a “Path” class which holds the path informations which are coming from the txt files. It has 3 propertiest which are from, to and distance. You can see the Path class below.

```
class Path{

    public:
        Path(int from, int to, int distance);
        int from;
        int to;
        int distance;
        void printPathInformations();
};


Path::Path(int from, int to, int distance){
    this->from = from;
    this->to = to;
    this->distance = distance;
}
```

Figure 1: “Path” Class

2) Node Class

Node class holds the informations about nodes in the paths and it has two methods inside it which are "*clone*" and "hasContain". Clone method creates a copy node from the current node. It's needed for my dijkstra algorithm. Second method "hasContain" checks whether there is a node or not in the vector that holds the full path from the reference point. This method is used for selecting a path which does not have any loop inside it. You can see the Node class below:

```
class Node{

    public:
        Node(int index, int distanceFromReferancePoint);
        int index;
        int distanceFromReferencePoint;

        bool hasContain(int nodeIndex);
        vector<Node*> pathFromReferencePoint;
        Node* clone();
        void printNode();
};



Node::Node(int index, int distanceFromReferencePoint){
    this->index = index;
    this->distanceFromReferencePoint = distanceFromReferencePoint;

}

Node* Node::clone(){
    Node* cloneNode = new Node(this->index, this->distanceFromReferencePoint);
    cloneNode->pathFromReferencePoint = this->pathFromReferencePoint;
    return cloneNode;
}

bool Node::hasContain(int nodeIndex){
    int length = this->pathFromReferencePoint.size();
    bool isFound = false;

    for(int i=0; i<length; i++){
        if(this->pathFromReferencePoint.at(i)->index == nodeIndex){
            isFound = true;
        }
    }

    return isFound;
}
```

Figure 2: Node class

3) FileReader Classes

In the project, path informations are provided by a ".txt" file. This class reads this txt file and provides necessary informations such as path list, number of nodes, number of paths, home and destination locations of Lucy and Joseph with its getter methods inside the class. You can see the FileReader class below:

```
class FileReader{
    private:
        char* filename;
        Path** pathList;
        int JH, JD, LH, LD;
        int numberOfNodes;
        int numberOfPaths;


    public:
        FileReader(const char* filename);
        ~FileReader();
        void read();

        // GETTERS
        Path** getPathList(){return this->pathList;}
        int getNumberOfPaths(){return this->numberOfPaths;}
        int getNumberOfNodes(){return this->numberOfNodes;}

        int getLucysHome(){return this->LH;}
        int getLucysDestination(){return this->LD;}
        int getJosephsHome(){return this->JH;}
        int getJosephsDestination(){return this->JD;}
};

// Constructor
FileReader::FileReader(const char* filename){
    this->filename = new char[FILENAME_LENGTH];
    strcpy(this->filename, filename);
}


// This method opens the files provided in the constructor,
// reads all of the paths and add them into the path list
void FileReader::read(){
    int from, to, distance;
    ifstream file;

    file.open(this->filename);

     if (!file) {
        cout << "File could not be opened!";
        return;
    }

    // Read the number of nodes in the city
    file >> this->numberOfNodes;

    // Read JH, JD, LH and LD
    file >> this->JH >> this->JD >> this->LH >> this->LD;

    // Memory allocation for the path array
    this->pathList = new Path*[PATH_LIST_MAX_SIZE];

    for(int i=0; !file.eof(); i++) {
        file >> from >> to >> distance;
        this->pathList[i] = new Path(from, to, distance);
        this->numberOfPaths++;
    }

    file.close();
}
```

Figure 3: File Reader class

b) Service Layer

Service layer of the project has all the business logic of the project and it provides data to be used for the simulation after processing it. This layer is the brain of the project. There are 4 service classes in the project: Adjacent node finder, graph creater, shortest path finder and travel planner classes. I will explain how these service classes work shortly.

1) CreateGraph Class

This class takes node list which is read from the file and it creates adjacency matrix based on the constructor variables. You can see the code below:

```
class CreateGraph{
    private:
        Path** nodes;
        int numberOfNodes;
        int numberOfPaths;
        int** matrix;

        void createGraph();

    public:
        CreateGraph(Path** nodes, int numberOfNodes, int numberOfPaths);
        int** getAdjacencyMatrix();
        void printGraph();

};



CreateGraph::CreateGraph(Path** nodes, int numberOfNodes, int numberOfPaths){
    this->nodes = nodes;
    this->numberOfNodes = numberOfNodes;
    this->numberOfPaths = numberOfPaths;

    // Allocating memory for a 2 dimensional array
    this->matrix = new int*[numberOfNodes];
    for(int i=0; i<numberOfNodes; i++){
        this->matrix[i] = new int[numberOfNodes];
        for(int j=0; j<numberOfNodes; j++){
            matrix[i][j] = 0;
        }
    }

    createGraph();
}

void CreateGraph::createGraph(){
    for(int i=0; i<this->numberOfPaths; i++){
        Path* currentNode = this->nodes[i];

        int from = currentNode->from;
        int to = currentNode->to;
        int distance = currentNode->distance;

        this->matrix[from][to] = distance;
    }

}

int** CreateGraph::getAdjacencyMatrix(){
    return this->matrix;
}

void CreateGraph::printGraph(){
    for(int i=0; i<numberOfNodes; i++){
        for(int j=0; j<numberOfNodes; j++){
            printf("%d ", this->matrix[i][j]);
        }
        printf("\n");
    }
}
```

Figure 4: “CreateGraph” Class

2) AdjacentNodeFinder Class

This class is used to find adjacent nodes for a given node index. It uses the adjacency matrix provided by the first service class(CreateGraph class). "*getAdjacentNodes"* method returns the adjacent nodes' indexes as a list. You can look at this class below:

```
class AdjacentNodeFinder{
    private:
        int** matrix;
        int numberOfNodes;
        int numberOfAdjacentNodes;



    public:
        AdjacentNodeFinder(int** matrix, int numberOfNodes);
        int* getAdjacentNodes(int from);
        int getNumberOfAdjacentNodes();

};



AdjacentNodeFinder::AdjacentNodeFinder(int** matrix, int numberOfNodes){
    this->matrix = matrix;
    this->numberOfNodes = numberOfNodes;
    this->numberOfAdjacentNodes = 0;


}

int* AdjacentNodeFinder::getAdjacentNodes(int from){
    int* adjacentList = new int[numberOfNodes];
    numberOfAdjacentNodes = 0;

    for(int i=0; i<numberOfNodes; i++){
        if(matrix[from][i] != 0){
            adjacentList[numberOfAdjacentNodes++] = i;
        }
    }

    return adjacentList;
}

int AdjacentNodeFinder::getNumberOfAdjacentNodes(){
    return this->numberOfAdjacentNodes;
}
```

Figure 5: AdjacentNodeFinder Class

3) ShortestPathFinder Class

This class finds shortest path and second shortest path between two nodes. It uses the previous service AdjacentNodeFinder class inside it. Initial shortest distances are set as 999999 (represents infinity value). My shortest path finder class changes these infinity values if there is any shortest or second shortest paths. It does this operation recursively by using a helper method. You can see the signiaure of the class below:

```
class ShortestPathFinder{
    private:
        int from;
        int to;
        int** matrix;
        int numberOfNodes;
        AdjacentNodeFinder* adjacentFinder;

        vector<Node*> shortestPath;
        vector<Node*> secondShortestPath; // If there is any

        int shortestDistance;
        int secondShortestDistance; // If there is any

        void helperMethod(Node* node);

    public:
        ShortestPathFinder(int** adjacencyMatrix, int from, int to, int numberOfNodes);
        ShortestPathFinder* find();
        int getShortestPathLength(){return this->shortestDistance;};
        int getSecondShortestPathLength(){return this->secondShortestDistance;};

        vector<Node*> getShortestPath(){return this->shortestPath;}
        vector<Node*> getSecondShortestPath(){return this->secondShortestPath;}


};



ShortestPathFinder::ShortestPathFinder(int** adjacencyMatrix, int from, int to, int numberOfNodes){
    this->from = from;
    this->to = to;
    this->matrix = adjacencyMatrix;
    this->numberOfNodes = numberOfNodes;
    this->adjacentFinder = new AdjacentNodeFinder(adjacencyMatrix, numberOfNodes);

    this->shortestDistance = DISTANCE_MAX;
    this->secondShortestDistance = DISTANCE_MAX;
}
```

Figure 6: ShortestPathFinder Class Signature

find() method of the class initializes the starting node and calls the recursive helper method. It calls itself until it arrives the destination node or it does not have any more node to go. My comments in the code explains everything, please check the comments in the code below for these methods.

```
ShortestPathFinder* ShortestPathFinder::find(){
    // First starting node (Reference Node)
    Node* startingNode = new Node(from, 0);

    helperMethod(startingNode);

    return this;
}

void ShortestPathFinder::helperMethod(Node* node){

    // If destination is arrived
    if(node->index == this->to){
        node->pathFromReferencePoint.push_back(node);

        // If its shortest distance, then update variables
        if(node->distanceFromReferencePoint < shortestDistance){
            secondShortestDistance = shortestDistance;
            shortestDistance = node->distanceFromReferencePoint;

            secondShortestPath = shortestPath;
            shortestPath = node->pathFromReferencePoint;
        }
        // Else if its the second shortest distance, then update variables
        else if(node->distanceFromReferencePoint < secondShortestDistance){
            secondShortestDistance = node->distanceFromReferencePoint;
            secondShortestPath = node->pathFromReferencePoint;
        }
    }

    // Else, keep going with adjacent nodes
    else {
        int* adjacentNodeIndexes = adjacentFinder->getAdjacentNodes(node->index);
        int numberOfAdjacentNodes = adjacentFinder->getNumberOfAdjacentNodes();

        for(int i=0; i<numberOfAdjacentNodes; i++){

            // If new node does not exist in the path history, then create new node from current node

            if( !node->hasContain(adjacentNodeIndexes[i]) ){
                // Clone the current node
                Node* clonedNode = node->clone();

                // Set the node index
                clonedNode->index = adjacentNodeIndexes[i];

                // Add current node into path history of the node
                clonedNode->pathFromReferencePoint.push_back(node);

                // Update the distance
                int from = node->index;
                int to = adjacentNodeIndexes[i];
                clonedNode->distanceFromReferencePoint += matrix[from][to]; // += distance between from and to

                // Keep going with new path RECURSIVELY
                helperMethod(clonedNode);

            }
        }
    }
}
```

Figure 7: ShortestPathFinder Class Methods

4) TravelPlanner Class

TravelPlanner class does all of the hard work. It takes Lucy's and Joseph's home&destination locations and finds a forward and backward path which they don't see each other's face.

This class has plan() method which calls planForwardPath() and planBackwardPath() private methods inside it. Also it has showPlannedTravelDetails() method which shows the planned paths if there is a possible solution. You can see the signature of this class below:

```
enum TravelStatus{
    NOT_INITIALIZED,
    NO_SOLUTION,
    BOTH_USE_SHORTEST_PATHS,
    LUCY_USES_ALTERNATIVE_PATH,
    JOSEPH_USES_ALTERNATIVE_PATH,
    BOTH_USE_ALTERNATIVE_PATHS,
};


class TravelPlanner{
    private:
        vector<Node*> LucysPathFromHomeToDest;
        vector<Node*> LucysAlternativePathFromHomeToDest;

        vector<Node*> LucysPathFromDestToHome;
        vector<Node*> LucysAlternativePathFromDestToHome;

        vector<Node*> JosephsPathFromHomeToDest;
        vector<Node*> JosephsAlternativePathFromHomeToDest;

        vector<Node*> JosephsPathFromDestToHome;
        vector<Node*> JosephsAlternativePathFromDestToHome;

        int LucysHome, LucysDestination;
        int JosephsHome, JosephsDestination;

        int LucysLeavingTime;
        int JosephsLeavingTime;


        TravelStatus travelStatusForward;
        TravelStatus travelStatusBackward;

        vector<Node*> lucysForwardPath;
        vector<Node*> josephsForwardPath;

        vector<Node*> lucysBackwardPath;
        vector<Node*> josephsBackwardPath;


        bool isThereAnyCollisionFromHomeToDestination(vector<Node*> LucysPathFromHomeToDest, vector<Node*> JosephsPathFromHomeToDest);
        bool isThereAnyCollisionFromDestinationToHome(vector<Node*> LucysPathFromDestToHome, vector<Node*> JosephsPathFromDestToHome);

        void planForwardPath();
        void planBackwardPath();

        bool canWeChangeLucysForwardPath();
        bool canWeChangeJosephsForwardPath();
        bool canWeChangeBothForwardPaths();



    public:
        TravelPlanner(int LucysHome, int LucysDestination, int JosephsHome, int JosephsDestination, int** adjacencyMatrix, int numberOfNodes);
        void plan();
        void showPlannedTravelDetails();

};
```

Figure 8: TravelPlanner Class Signature

This class has almost 600 lines of codes and it's hard to explain everything here but these codes and comments are very readable and clean. Comments explains every step of the code. Please check this class' methods for detailed informations.

c) Presentation Layer

Presentation layer has only TravelSimulation class which has all of the necessary classes which setup the initial status of the scenario and simulate the travel with that given scenario. These simulation class uses the services and data providers that I already explained, so that it's task is very easy. It uses these classes and shows the simulation results. You can see the TravelSimulation class and main below:

```
class TravelSimulation{
    private:
        TravelSimulation();

    public:
        static void simulate(char* filename);
};

void TravelSimulation::simulate(char* filename){

    FileReader fileReader(filename);
    fileReader.read();
    Path** list = fileReader.getPathList();


    int numberOfNodes = fileReader.getNumberOfNodes();
    int numberOfPaths = fileReader.getNumberOfPaths();

    int LucysHome = fileReader.getLucysHome();
    int LucysDestination = fileReader.getLucysDestination();
    int JosephsHome = fileReader.getJosephsHome();
    int JosephsDestination = fileReader.getJosephsDestination();



    CreateGraph graph(list, numberOfNodes, numberOfPaths);
    int** adjacencyMatrix = graph.getAdjacencyMatrix();


    TravelPlanner* travelPlanner = new TravelPlanner(
                                        LucysHome, LucysDestination,
                                        JosephsHome, JosephsDestination,
                                        adjacencyMatrix, numberOfNodes
                                        );

    travelPlanner->plan();
    travelPlanner->showPlannedTravelDetails();

}
```

Figure 9: TravelSimulation Class

```
int main(int argc, char *argv[]){


    if(argc == 2){
        TravelSimulation::simulate(argv[1]);
    }

    else{
        printf("Check your input parameters\n\n");
    }

    return 0;
}
```

Figure 10: main method

PART 2 – SIMULATION RESULTS

In this part, I will show my code's outputs for 5 test cases in Ninova. You can see them below:

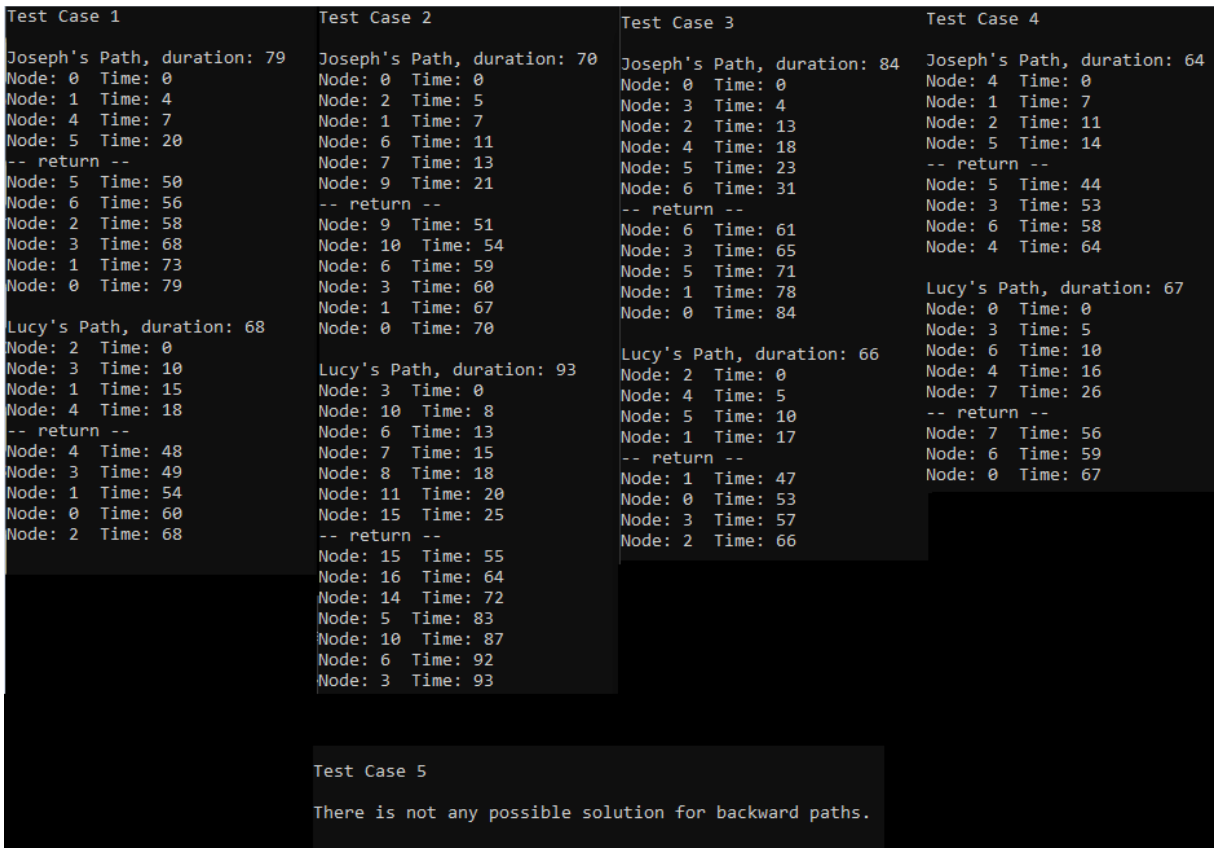

Figure 11: Simulation results for 5 test cases