İskender Akyüz
150150150

BLG336E – Homework II
Report

- The program can be compiled by using this command line:
  $ g++ -std=c++11 main.cpp -o main
- In order to run the program, you could use this command line:
  $ ./main test1.txt

1) Code Explanation

1.a) Part 1: Constructing the Graph

```
struct Edge {
    int src, dest, weight;
};

struct Node {
    int vertex, weight;
};

typedef pair<int, int> Pair;
```

First of all, I have created some useful structs in order to use them in my graph class. The Edge struct stores the information about the edges and the Node struct stores the nodes in the graph.

```
class Graph {
private:
    int numberOfNodes;
    vector<vector<Pair>> adjList;
    int getCost(int, int) const;
public:
    Graph(vector<Edge> const&, int);        // Constructor
    Graph(const Graph &, Pair);             // Copy Constructor
    vector<int> DijkstraSP(int);
    void getPath(vector<int> &, int, int, vector<Pair> &);
};
```

The Graph class I have written is above. It has two variables: *numberOfNodes* and *adjList*. *adjList* holds the information about the connections in the graph. *adjList* is a vector of vector<Pair>s. Basically, each index which is node number holds the connections to other nodes.

First variable of the Pair is the connected node number and the second variable is the weight of the connection.

```
Graph::Graph(vector<Edge> const& edges, int n) {        // Constructor
    numberOfNodes = n;
    adjList.resize(numberOfNodes);

    for (auto& edge : edges)
        adjList[edge.src].push_back(make_pair(edge.dest, edge.weight));
}
```

The constructor for the class is shown above. It gets the edges as a parameter and push them into adjList.

```
Graph::Graph(const Graph& graph, Pair edge) {   // copying the class without the
edge that causeS the intersection
    numberOfNodes = graph.numberOfNodes;
    adjList.resize(numberOfNodes);
    for (int i = 0; i < numberOfNodes; i++) {
        for (auto p : graph.adjList[i]) {
            if (i == edge.first && p.first == edge.second)  continue;
            adjList[i].push_back(p);
        }
    }
}
```

My copy constructor function is above. This function does not just copy a graph. It takes an edge and copy the graph by removing this edge. It is required when there is an intersection between paths. We'll see it in detail in part 2.

1.b) Part 2: Implementing the algorithm

```
vector<int> Graph::DijkstraSP(int source) {
    priority_queue<Node, vector<Node>, comp> minHeap;
    minHeap.push({source, 0});

    vector<int> dist(numberOfNodes, INT_MAX);
    dist[source] = 0;

    vector<bool> done(numberOfNodes, false);
    done[source] = true;

    vector<int> prev(numberOfNodes, -1);

    while(!minHeap.empty()) {
        // get the best vertex
        Node node = minHeap.top();
        minHeap.pop();

        int u = node.vertex;
```

```
        for (auto i : adjList[u]) {
            int v = i.first;
            int weight = i.second;
            if (!done[v] && (dist[u] + weight) < dist[v]) {
                dist[v] = dist[u] + weight;
                prev[v] = u;
                minHeap.push({v, dist[v]});
            }
        }
        done[u] = true;
    }
    return prev;
}
```

The Dijkstra shortest path algorithm is shown above. I used a min heap in order to check the node with the lowest distance first. Also, I have created a *dist* vector in order to keep track of the distances. At first, all nodes have infinite distances. Also, I have created a bool vector *done* in order to prevent loops.

After the creation of these necessary variables, the algorithm starts popping the node with the lowest distance and checks its connections. If the new connection's distance is lower than the previous, then it updates it. This loop continues until the min heap is empty. Also, in each update, *prev* vector is also updated. *prev* vector stores the previous nodes for each node that has the lowest distance to it. Thus, by using this vector, not only we get the shortest distances to each node but also, we will get the paths.

```
void Graph::getPath(vector<int>& prev, int i, int startTime, vector<Pair>& pathAn
dCost) {
    if (i < 0)  return;
    getPath(prev, prev[i], startTime, pathAndCost);
    int timeCost = startTime;
    if (!pathAndCost.empty())
        timeCost = pathAndCost.back().second + getCost(prev[i], i);
    pathAndCost.push_back(make_pair(i, timeCost));
}
```

The *getPath()* function is above. It is recursive function. By starting from the destination node, it goes to the start node. It updates the given vector *pathAndCost* which stores the visited nodes and their visit times.

Now, the Dijkstra's shortest path algorithm is ready. Let's analyze the implementation of arranging the travel plans.

```
void ArrangeTravelPlans(Graph graph, TravelPlan* plans) {
    vector<Pair> JosephHomeToDestPath, LucyHomeToDestPath;
```

```
    if (!OneWayTravel(graph, JosephHomeToDestPath, LucyHomeToDestPath, plans, { 0
, 0 }, true)) {        // travelling from home to dest
        cout << "No solution!" << endl;
        return;
    }

    vector<Pair> JosephDestToHomePath, LucyDestToHomePath;
    if (!OneWayTravel(graph, JosephDestToHomePath, LucyDestToHomePath, plans, { J
osephHomeToDestPath.back().second + 30, LucyHomeToDestPath.back().second + 30 },
false)) {     // travelling from dest to home
        cout << "No solution!" << endl;
        return;
    }

    PrintPaths(JosephHomeToDestPath, LucyHomeToDestPath, JosephDestToHomePath, Lu
cyDestToHomePath);
}
```

In *ArrangeTravelPlans()*, I have separated the journeys from home to destination and destination to home. After running OneWayTravel() function for both of them, I have printed the result. It is a quite simple function and does what it is supposed to do.

```
bool OneWayTravel(Graph graph, vector<Pair>& JosephPathAndCost, vector<Pair>& Luc
yPathAndCost, TravelPlan* plans, Pair startTimes, bool goingToDest) {
    // Joseph's path
    vector<int> JosephPath = graph.DijkstraSP(goingToDest ? plans[Joseph].homeNod
e : plans[Joseph].destNode);
    graph.getPath(JosephPath, goingToDest ? plans[Joseph].destNode : plans[Joseph
].homeNode, startTimes.first, JosephPathAndCost);
    if (goingToDest && JosephPathAndCost[0].first != (goingToDest ? plans[Joseph]
.homeNode : plans[Joseph].destNode))       // could not find a path
        return false;

    // Lucy's path
    vector<int> LucyPath = graph.DijkstraSP(goingToDest ? plans[Lucy].homeNode :
plans[Lucy].destNode);
    graph.getPath(LucyPath, goingToDest ? plans[Lucy].destNode : plans[Lucy].home
Node, startTimes.second, LucyPathAndCost);
    if (goingToDest && LucyPathAndCost[0].first != (goingToDest ? plans[Lucy].hom
eNode : plans[Lucy].destNode))         // could not find a path
        return false;

    int intersectionNode = CheckIntersection(JosephPathAndCost, LucyPathAndCost);
    if (intersectionNode != -1) {
        Pair startingNodes = { JosephPath[intersectionNode], LucyPath[intersectio
nNode] };
```

```
        FindAlternativePath(graph, JosephPathAndCost, LucyPathAndCost, plans, sta
rtTimes, startingNodes, intersectionNode, goingToDest);
        if (JosephPathAndCost.empty() || LucyPathAndCost.empty())
            return false;
    }
    return true;
}
```

In *OneWayTravel()*, I have run the Dijkstra's algorithm and found the shortest path for both Joseph and July. By just checking the first element in the paths, I have checked if the paths are valid and go from home to their destinations. If they are not, the function returns false indicating that valid paths could not be found.

If there is no problem with finding paths, then the function checks if there are any intersection between Joseph's and Lucy's paths. For this purpose, *CheckIntersection()* is run. This function returns the node number that the intersection happens if it finds any; otherwise, it returns -1. Also, in the case of intersection, *FindAlternativePath()* is run. If this function could not find any alternatives then, OneWayTravel returns false again.

Firstly, we will see how *CheckIntersection()* works.

```
int CheckIntersection(vector<Pair> JosephPathAndCost, vector<Pair> LucyPathAndCos
t) {
    for (vector<Pair>::iterator it = JosephPathAndCost.begin(); it != JosephPathA
ndCost.end(); ++it)
        for (vector<Pair>::iterator it2 = LucyPathAndCost.begin(); it2 != LucyPat
hAndCost.end(); ++it2)
            if (it->first == it2->first && it->second == it2->second)
                return it->first;

    if (JosephPathAndCost.front().second == 0) {        // checking Joseph's wait
ing when he goes to his destination
        for (vector<Pair>::iterator it2 = LucyPathAndCost.begin(); it2 != LucyPat
hAndCost.end(); ++it2)
            if (JosephPathAndCost.back().first == it2->first && (it2-
>second >= JosephPathAndCost.back().second && it2-
>second <= JosephPathAndCost.back().second + 30))
                return it2->first;
    }
    else {          // checking Joseph's waiting when he goes back to his home
        for (vector<Pair>::iterator it2 = LucyPathAndCost.begin(); it2 != LucyPat
hAndCost.end(); ++it2)
            if (JosephPathAndCost.front().first == it2->first && (it2-
>second >= JosephPathAndCost.front().second - 30 && it2-
>second <= JosephPathAndCost.front().second))
                return it2->first;
```

```
    }
    if (LucyPathAndCost.front().second == 0) {      // checking Lucy's waiting wh
en he goes to his destination
        for (vector<Pair>::iterator it = JosephPathAndCost.begin(); it != JosephP
athAndCost.end(); ++it)
            if (LucyPathAndCost.back().first == it->first && (it-
>second >= LucyPathAndCost.back().second && it-
>second <= LucyPathAndCost.back().second + 30))
                return it->first;
    }
    else {          // checking Lucy's waiting when he goes back to his home
        for (vector<Pair>::iterator it = JosephPathAndCost.begin(); it != JosephP
athAndCost.end(); ++it)
            if (LucyPathAndCost.front().first == it->first && (it-
>second >= LucyPathAndCost.front().second - 30 && it-
>second <= LucyPathAndCost.front().second))
                return it->first;
    }
    return -1;      // could not find any intersection
}
```

This function first checks every element in both paths and compare them with each other. If the node number and visit time is the same then, there is an intersection and the function returns the intersection node number. If there is no problem, then it checks the waiting times for both of them.

While one of them is waiting in their destination node, the other one cannot visit that node. The following 4 if statements are responsible for this check. After checking the waiting condition, the function returns -1 if there is no intersection.

Now, let's see how am I finding an alternative path.

```
void FindAlternativePath(Graph graph, vector<Pair>& JosephPathAndCost, vector<Pai
r>& LucyPathAndCost, TravelPlan* plans, Pair startTimes, Pair startingNodes, int
intersectionNode, bool goingToDest) {
    int JosephTravelTime = JosephPathAndCost.back().second - JosephPathAndCost.fr
ont().second,
        LucyTravelTime = LucyPathAndCost.back().second - LucyPathAndCost.front().
second,
        JosephAltTravelTime = -1, LucyAltTravelTime = -1;
    bool altPathFound = true;

    // Joseph - Alternative path
    Graph copyGraphJoseph = Graph(graph, { startingNodes.first, intersectionNode
});
```

```
    vector<int> JosephAltHomeToDestPath = copyGraphJoseph.DijkstraSP(goingToDest
? plans[Joseph].homeNode : plans[Joseph].destNode);
    vector<Pair> JosephAltPathAndCost;
    copyGraphJoseph.getPath(JosephAltHomeToDestPath, goingToDest ? plans[Joseph].
destNode : plans[Joseph].homeNode, startTimes.first, JosephAltPathAndCost);
    if (JosephAltPathAndCost[0].first == (goingToDest ? plans[Joseph].homeNode :
plans[Joseph].destNode))        // found an alternative path
        JosephAltTravelTime = JosephAltPathAndCost.back().second - JosephAltPathA
ndCost.front().second;

    // Lucy - Alternative path
    Graph copyGraphLucy = Graph(graph, { startingNodes.second, intersectionNode }
);
    vector<int> LucyAltHomeToDestPath = copyGraphLucy.DijkstraSP(goingToDest ? pl
ans[Lucy].homeNode : plans[Lucy].destNode);
    vector<Pair> LucyAltPathAndCost;
    copyGraphLucy.getPath(LucyAltHomeToDestPath, goingToDest ? plans[Lucy].destNo
de : plans[Lucy].homeNode, startTimes.second, LucyAltPathAndCost);
    if (LucyAltPathAndCost[0].first == (goingToDest ? plans[Lucy].homeNode : plan
s[Lucy].destNode))          // found an alternative path
        LucyAltTravelTime = LucyAltPathAndCost.back().second - LucyAltPathAndCost
.front().second;

    if (JosephAltTravelTime != -1 && LucyAltTravelTime != -
1) {     // alternative paths exist for both of them
        bool JosephAltPathIntersect = CheckIntersection(JosephAltPathAndCost, Luc
yPathAndCost) != -1;        // checking if alt path causes intersection
        bool LucyAltPathIntersect = CheckIntersection(JosephPathAndCost, LucyAltP
athAndCost) != -1;          // checking if alt path causes intersection
        if (!JosephAltPathIntersect && !LucyAltPathIntersect) {
            if ((JosephAltTravelTime + LucyTravelTime) <= (JosephTravelTime + Luc
yAltTravelTime))            // Joseph's alt path takes less time
                JosephPathAndCost = JosephAltPathAndCost;
            else if ((JosephTravelTime + LucyAltTravelTime) <= (JosephAltTravelTi
me + LucyTravelTime))       // Lucy's alt path takes less time
                LucyPathAndCost = LucyAltPathAndCost;
        }
        else if (!JosephAltPathIntersect)
            JosephPathAndCost = JosephAltPathAndCost;
        else if (!LucyAltPathIntersect)
            LucyPathAndCost = LucyAltPathAndCost;
        else        // both alternative paths cause intersection, no solution
            altPathFound = false;
    }
    else if (JosephAltTravelTime != -1) { // only Joseph may take an alt path
```

```
        if (CheckIntersection(JosephAltPathAndCost, LucyPathAndCost) == -
1) // Joseph's alt path does not cause intersection
            JosephPathAndCost = JosephAltPathAndCost;
        else                        // the alt path is not valid
            altPathFound = false;
    }
    else if (LucyAltTravelTime != -1) { // only Lucy may take an alt path
        if (CheckIntersection(JosephPathAndCost, LucyAltPathAndCost) == -1)
// Lucy's alt path does not cause intersection
            LucyPathAndCost = LucyAltPathAndCost;
        else                        // the alt path is not valid
            altPathFound = false;
    }
    else        // no alterantive paths for any of them
        altPathFound = false;

    if (!altPathFound) {        // could not find any alternative paths, so clear
 the previous paths
        JosephPathAndCost.clear();
        LucyPathAndCost.clear();
    }
}
```

After finding the intersection node, firstly, the function creates a new temporary graph by copying the original one. This new copy does not include the intersection node; thus, it could find an alternative path. After that, the function runs the Dijkstra's algorithm again and tries to find new alternative paths for both Joseph and Lucy. If both have alternative paths and don't have intersect with each other, then the path with less time is chosen. If they cause intersection again, finding alternative paths fails.

The second case is that only one of them has an alternative path. If this new path does not cause intersection, then the previous one is updated; otherwise, the alternative path finding fails again.

The final case is where there is no alternative path. In this case, alternative path finding fails again.

```
int main(int argc, char* argv[]) {
    string filename(argv[1]);
    int numberOfNodes;
    vector<Edge> edges;
    TravelPlan plans[2];
    ReadFile(filename, numberOfNodes, edges, plans);
    Graph graph(edges, numberOfNodes);          // creating the graph
    ArrangeTravelPlans(graph, plans);
    return 0;
}
```

This is my main function. After reading the file, *ArrangeTravelPlans()* is run and tries to find paths for Joseph and Lucy. Now, it is time to run the program and see the outputs.

2) Running the program

- ./main test1.txt

```
Joseph's Path, duration: 79
Node: 0 Time: 0
Node: 1 Time: 4
Node: 4 Time: 7
Node: 5 Time: 20
-- return --
Node: 5 Time: 50
Node: 6 Time: 56
Node: 2 Time: 58
Node: 3 Time: 68
Node: 1 Time: 73
Node: 0 Time: 79

Lucy's Path, duration: 68
Node: 2 Time: 0
Node: 3 Time: 10
Node: 1 Time: 15
Node: 4 Time: 18
-- return --
Node: 4 Time: 48
Node: 3 Time: 49
Node: 1 Time: 54
Node: 0 Time: 60
Node: 2 Time: 68
```

./main test2.txt

```
Joseph's Path, duration: 70
Node: 0 Time: 0
Node: 2 Time: 5
Node: 1 Time: 7
Node: 6 Time: 11
Node: 7 Time: 13
Node: 9 Time: 21
-- return --
Node: 9 Time: 51
Node: 10 Time: 54
Node: 6 Time: 59
Node: 3 Time: 60
Node: 1 Time: 67
Node: 0 Time: 70

Lucy's Path, duration: 93
Node: 3 Time: 0
Node: 10 Time: 8
Node: 6 Time: 13
Node: 7 Time: 15
Node: 8 Time: 18
Node: 11 Time: 20
Node: 15 Time: 25
-- return --
Node: 15 Time: 55
Node: 16 Time: 64
Node: 14 Time: 72
Node: 5 Time: 83
Node: 10 Time: 87
Node: 6 Time: 92
Node: 3 Time: 93
```

./main test3.txt

```
Joseph's Path, duration: 84
Node: 0 Time: 0
Node: 3 Time: 4
Node: 2 Time: 13
Node: 4 Time: 18
Node: 6 Time: 31
-- return --
Node: 6 Time: 61
Node: 3 Time: 65
Node: 5 Time: 71
Node: 1 Time: 78
Node: 0 Time: 84

Lucy's Path, duration: 66
Node: 2 Time: 0
Node: 4 Time: 5
Node: 5 Time: 10
Node: 1 Time: 17
-- return --
Node: 1 Time: 47
Node: 0 Time: 53
Node: 3 Time: 57
Node: 2 Time: 66
```

- ./main test4.txt

```
Joseph's Path, duration: 64
Node: 4 Time: 0
Node: 1 Time: 7
Node: 2 Time: 11
Node: 5 Time: 14
-- return --
Node: 5 Time: 44
Node: 3 Time: 53
Node: 6 Time: 58
Node: 4 Time: 64

Lucy's Path, duration: 67
Node: 0 Time: 0
Node: 3 Time: 5
Node: 6 Time: 10
Node: 4 Time: 16
Node: 7 Time: 26
-- return --
Node: 7 Time: 56
Node: 6 Time: 59
Node: 0 Time: 67
```

./main test5.txt

```
No solution!
```