Istanbul Technical University

Analysis of Algorithms II
Homework 2 Report

Ece Çınar

150150138

BLG 336E

20.04.2020

Classes & Methods

1. CustomCompare

This class is a custom comparison class. It compares two pairs according to their second elements. Since priority queue of C++ doesn't have a pair comparison function, this class was created.

2. Graph

Methods of this class create the graph, calculate shortest paths on it, find alternative paths, solve conflicts and print the final results. Adjacency matrix representation was used in order to implement the graph.

a. Constructor

Takes a parameter representing node number of the graph and resizes the adjacency matrix according to node number. Each cell of the matrix is initially zero.

b. print

Has one argument which represents path of Joseph or Lucy. It prints out content of the path in the required format.

c. set_edges

Has two arguments. The first argument is a node pair representing an edge of the graph and the second is weight of that edge. The cell representing this edge is updated with the weight value.

d. set_locations

Sets source (hotel) and destination nodes of Joseph and Lucy.

e. dijkstra

Takes three parameters which are start node, end node and initial time of the algorithm. After calculating shortest path from the start node to the end node, function returns that path. In order to represent infinity, limits library is used. Largest possible integer value is chosen to represent infinity.

Priority queue data type is used to find the node with minimum shortest path length from the start node. This queue consists node and shortest path length pairs. *CustomCompare* class is used to find the element with shortest path length and keep it at the top of the queue.

Map implementation is used to keep track of the visited nodes. A vector is used to keep the previous node of each node. This way it is possible to find the shortest path by backtracking.

At the beginning, each node is inserted into the priority queue with infinite path length except the start node. Also, previous node of each node is set to itself initially.

At each iteration of dijkstra, top node of the priority queue is processed. This node, I will call it *main node* in order to prevent confusion, is inserted into the visited map and removed from the priority queue. Then, remaining nodes of priority queue are removed from the queue and inserted into a temporary stack since all of these nodes are potential candidates to be updated.

Nodes which are connected to the *main node* are neighbors of it. Shortest path length of a neighboring node is updated if newly calculated path length is shorter than previous path length of that node. If shortest path length of a node is updated, previous node of that node is also changed to be the *main node*. After the calculations, each node is removed from the temporary stack and inserted back into the priority queue.

The algorithm continues with the next top node of priority queue. If all of the nodes are inserted into visited map or top element of the priority queue has infinite shortest path length, which means remaining nodes are not reachable from the start node, Dijkstra ends.

Since previous node is known for each node, it is easy to backtrack the shortest path. Starting from the end point, we continue with the previous node of each node and insert each node we encounter to our path. If we cannot reach start node, it means that there is no path between source and destination. In such a case, path variable is cleared. At the end, function returns the path.

f. *solve_conflicts*

Has three arguments which are overlap node and two paths with conflict. Aim of this function is to solve the overlapping problem by coming up with an alternative path for Lucy or Joseph. At the end, function returns true if the conflict is solved and false otherwise.

First of all, the edge causing overlap is temporarily removed from the adjacency matrix for Lucy. This edge is the edge connecting overlap node to the previous node of Lucy's path. Then, *dijkstra* method is called to calculate a second shortest path for Lucy. The same process is repeated for Joseph by removing the problematic edge and calculating a second shortest path for him.

If there are no alternative paths for Joseph and Lucy, function returns false immediately.

Two variables are used to describe validity of Lucy's and Joseph's alternative paths. If Lucy doesn't have an alternative path or the alternative path she has still intersects with Joseph's original path, her alternative path is considered to be invalid. The same rule applies for Joseph. If both alternative paths are invalid, function returns false.

If Joseph's alternative path is invalid but Lucy's is valid, then Lucy's original path is changed with her alternative path. Also, if both paths are valid but Lucy's alternative path in addition to Joseph's original path takes shorter time, again Lucy's alternative path is chosen. Otherwise, Joseph's alternative path is chosen. Function returns true after finding a valid combination of paths.

g. *find_shortest_paths*

This method calls other methods of Graph class in order to find four different shortest paths for Lucy's and Joseph's two-way tours.

First of all, *dijkstra* method is called for Lucy and Joseph to calculate initial paths for them. Time is set to zero for both and the paths are calculated between their hotels and destinations.

There are three different overlapping scenarios:

- Joseph and Lucy are at the same node at the same time.
- While Joseph is waiting at his destination node, Lucy visits his node and then continues to her destination point, and vice versa.

- While Joseph is waiting at his destination node, Lucy has already started going back to her hotel and she visits Joseph's destination node during her travel, and vice versa.

To check the first scenario, each node of Joseph's and Lucy's path are compared.

For the second scenario, it is checked if Lucy visits Joseph's destination node in the time interval [t, t+30] where t represents Joseph's arrival time to the destination node, and vice versa.

For the third scenario, it is checked if Lucy visits Joseph's destination node in the time interval [t-30, t] where t represents Joseph's departure time from the destination node, and vice versa.

All three scenarios are checked. If one of them is true, overlap variable is set and overlap node is determined.

If there are multiple overlaps, only the first one is taken into consideration. After finding an overlap node, overlap search process ends.

If overlap occurs, *solve_conflicts* method is called and this method returns a boolean which shows if the conflict was solved or not. If the conflict could not be solved, "No solution!" message is displayed and program terminates.

After valid paths are found for Joseph and Lucy, destination and source nodes are swapped both for Lucy and Joseph. Also, initial time for *dijkstra* is updated by adding 30 minutes to Joseph's and Lucy's path durations.

After necessary changes have been made, the same process is repeated for the opposite direction.

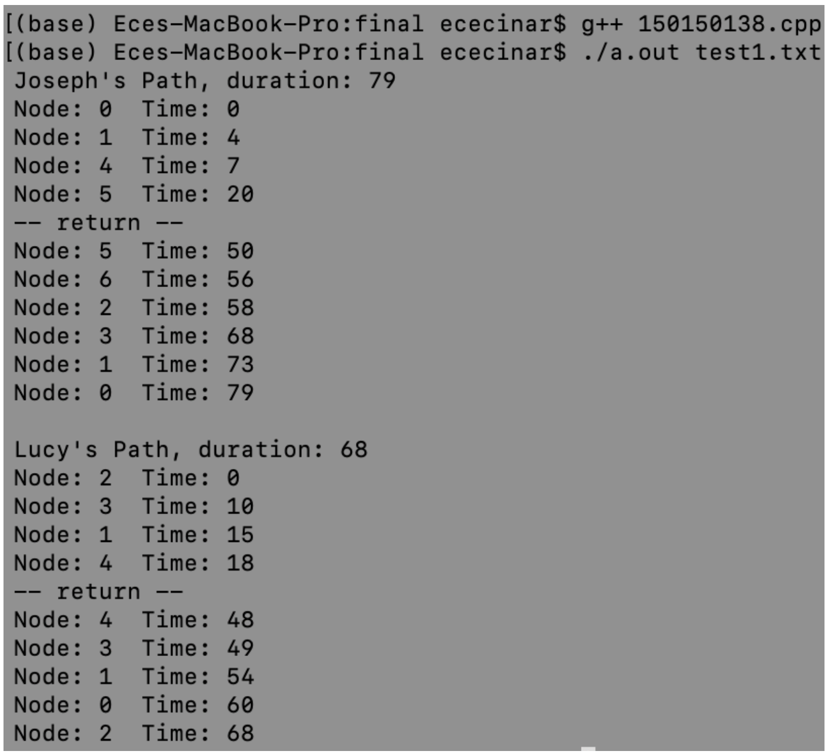
If four valid paths were successfully calculated, these paths are printed out by calling *print* function for each path.

3. File

File class performs necessary file operations. Constructor of File opens the file and destructor closes it. Read function of File initializes a Graph object with the number of nodes. Also, edges of the graph are read from the file and *set_edges* function is called for each edge to adjust the adjacency matrix.

Test Cases & Outputs

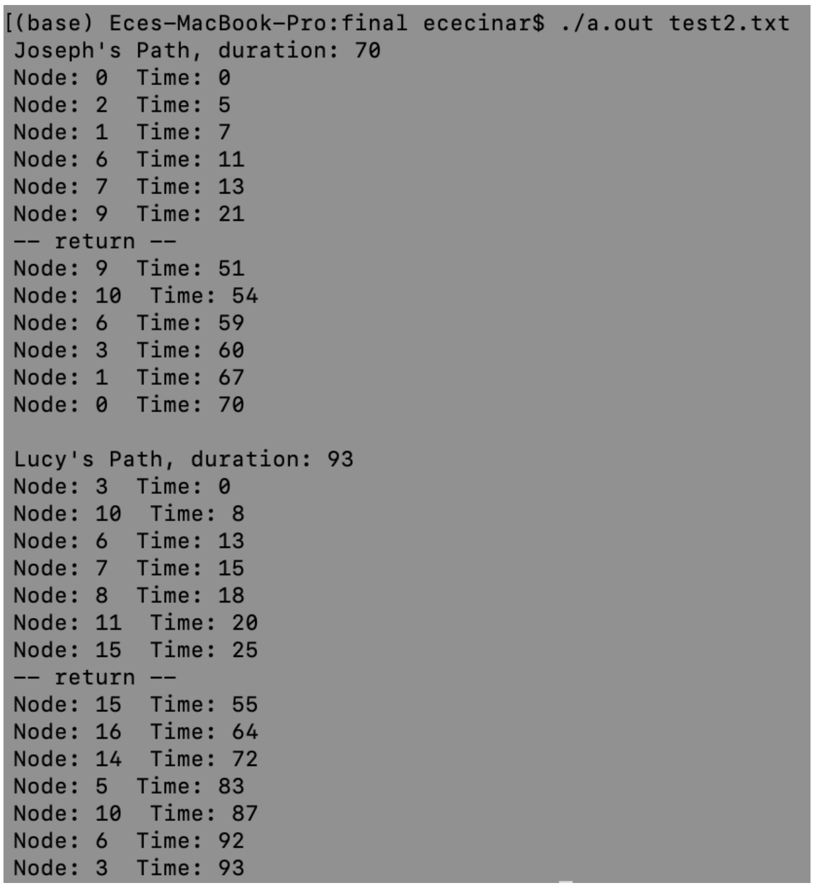
1. Test#1



```
[(base) Eces-MacBook-Pro:final ececinar$ g++ 150150138.cpp
[(base) Eces-MacBook-Pro:final ececinar$ ./a.out 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
```

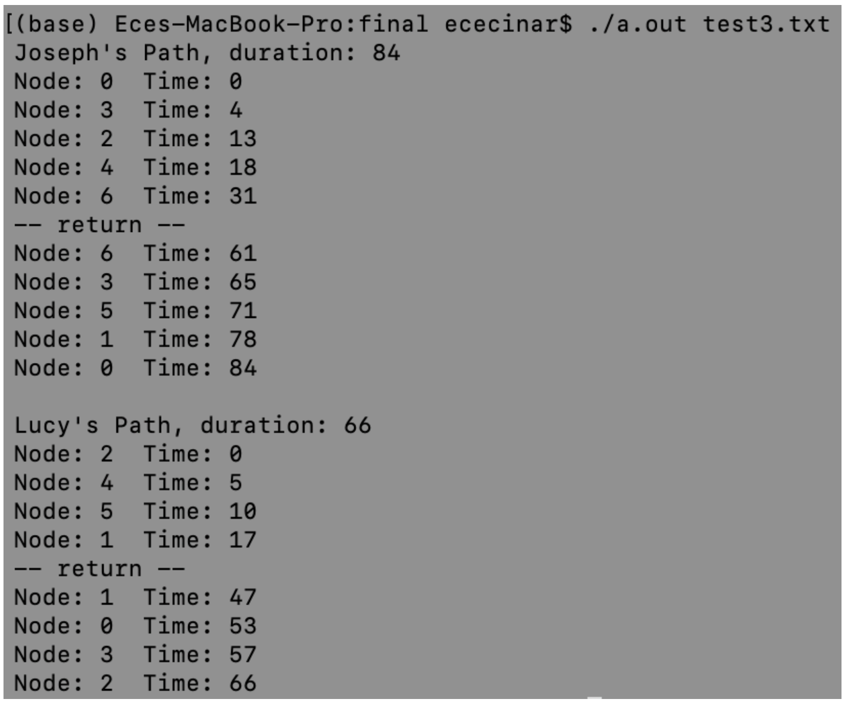
2. Test#2



```
[(base) Eces-MacBook-Pro:final ececinar$ ./a.out 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
```

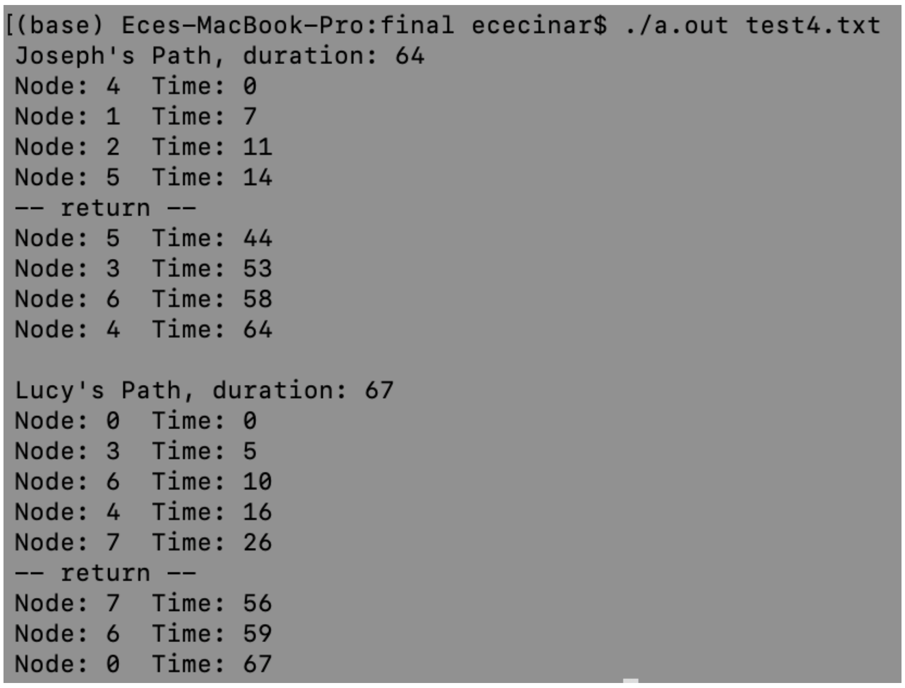
3. Test#3



```
[(base) Eces-MacBook-Pro:final ececinar$ ./a.out 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
```

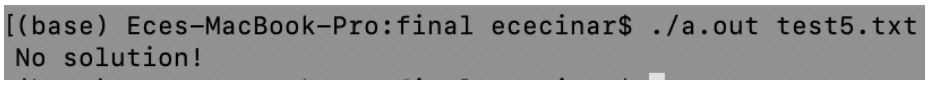
4. Test#4



```
[(base) Eces-MacBook-Pro:final ececinar$ ./a.out 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
```

5. Test#5



```
[(base) Eces-MacBook-Pro:final ececinar$ ./a.out test5.txt
 No solution!
```