ASSIGNMENT REPORT

Draghici Robert Cristian

June 7, 2021

Faculty: Automatics, Computers and Electrotehnics

GROUP: 2.2A

1 Problem statement:

Suppose two friends live in different cities on a map, such as the Romania map shown in Figure 2. On every turn, we can simultaneously move each friend to a neighboring city on the map. The amount of time needed to move from city i to neighbor j is equal to the road distance d(i, j) between the cities, but on each turn the friend that arrives first must wait until the other one arrives (and calls the first on his/her cell phone) before the next turn can begin. We want the two friends to meet as quickly as possible.
a. Write a detailed formulation for this search problem.
b. Identify a search algorithm for this task and explain your choice.

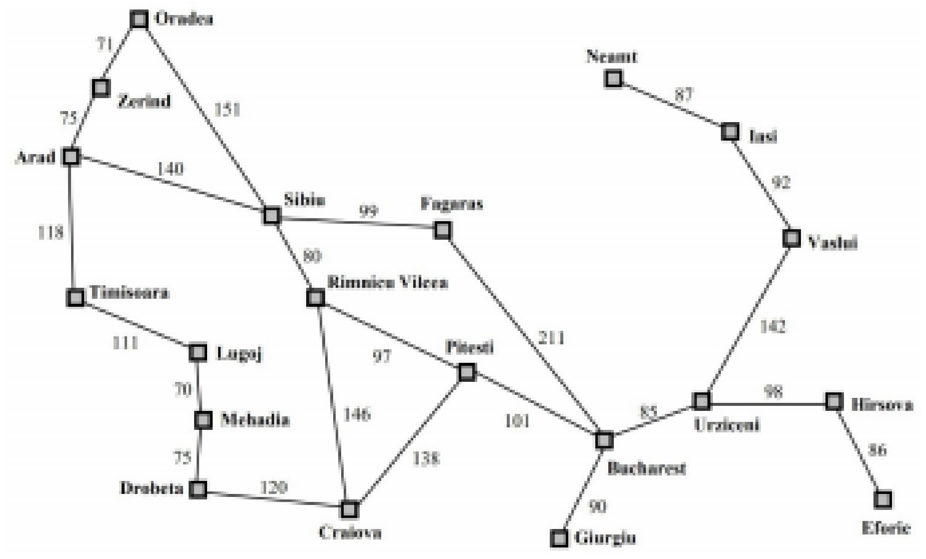


Figure 1: Romania Map

2 PSEUDOCODE for Yen's algorithm:

```
function YenKSP(Graph, source, goal, k):
  A[0] = dijkstra(Graph, source, goal);
  B = [];
  for k from 1 to K:
  // The spur node ranges from the first node to the next to last
previous k-shortest path.
     for i from 0 to size(A[k - 1]) - 2:
       // Spur node is retrieved from the previous k_shortest path
       spurNode = A[k-1].node(i);
       // The sequence of nodes from the source to the spur node o
       rootPath = A[k-1].nodes(0, i);
       for each path p in A:
           if rootPath == p.nodes(0, i):
            // Remove the links that are part of the previous short
            remove p.edge(i,i + 1) from Graph;
       for each node rootPathNode in rootPath except spurNode:
           remove rootPathNode from Graph;

       spurPath = Dijkstra(Graph, spurNode, sink);

      // Entire path is made up of the root path and spur path.
      totalPath = rootPath + spurPath;
      // Add the potential k-shortest path to the heap.
      if (totalPath not in B):
         B.append(totalPath);

      // Add back the edges and nodes that were removed from the g
      restore edges to Graph;
      restore nodes in rootPath to Graph;

     if B is empty:
       break;
    B.sort();
    // Add the lowest cost path becomes the k-shortest path.
    A[k] = B[0];
    B.pop();
  return A;
```

The Yen's algorithm is a searching algorithm that computes the first k-shortest single-source loopless paths for a weighted graph with non-negative edge cost.
I have decided to use this algorithm because in this problem's case, even though the goal is for the two friends to meet as fast as possible, finding the shortest path doesn't necessarily mean finding the best solution. This comes from the fact that after each turn the two friends have to wait for each other, so a shorter route can mean more waiting time for one of them, or both. So we need to find the optimal path starting from the shortest one, until a certain condition is fulfilled.

The best case scenario would be that none of them have to wait at none of their turns, leading to the path time being equal to half of the distance of the path, so by finding the most optimal path we can say where they meet.
So, considering this, we keep finding and comparing the first k paths until the best possible time of the path that is evaluated is better than our current best time, which cannot be better than half of the distance.
The algorithm can be broken down into two parts, determining the first k-shortest path, A^k, and then determining all other k-shortest paths. It is assumed that the container A will hold the k-shortest path, whereas the container B, will hold the potential k-shortest paths.

The time complexity of Yen's algorithm is dependent on the shortest path algorithm used in the computation of the spur paths, so the Dijkstra algorithm is assumed. Dijkstra's algorithm has a worse case time complexity of O(N^2), but using a Fibonacci heap it becomes O(M+N log N), where M is the amount of edges in the graph.
Since Yen's algorithm makes K*l calls to the Dijkstra in computing the spur paths, where l is the length of spur paths. In a condensed graph, the expected value of l is O(log N), while the worst case is N, the time complexity becomes O(KN(M + Nlog N)).

3 Experimental Data

In this section, I describe the algorithm used to create the random data generator, what input it generates and the importance of this data for the testing of my program.

PSEUDOCODE:

```
1. import random
2. city_list = [...] //names of romanian cities
3. random_city = random.choice(city_list)
4. C1 = random_city
5. random_city = random.choice(city_list)
6. C2 = random_city
```

First, I created a list with all the names of the cities on the Romanian map taken as example for this problem.
Then I took two variables C1 and C2 which represent the cities from where each of the two friend starts. This cities are randomly chosen from the list of cities using a very simple random function upon that list.
Here it is worth mentioning that I chose the Python language because I find it the most user-friendly and easiest when it comes to working with data structures such as graphs.

4 Design of the experimental application

The application consists of: main.py and algorithm.py
The input of the application is generated randomly and consists of:

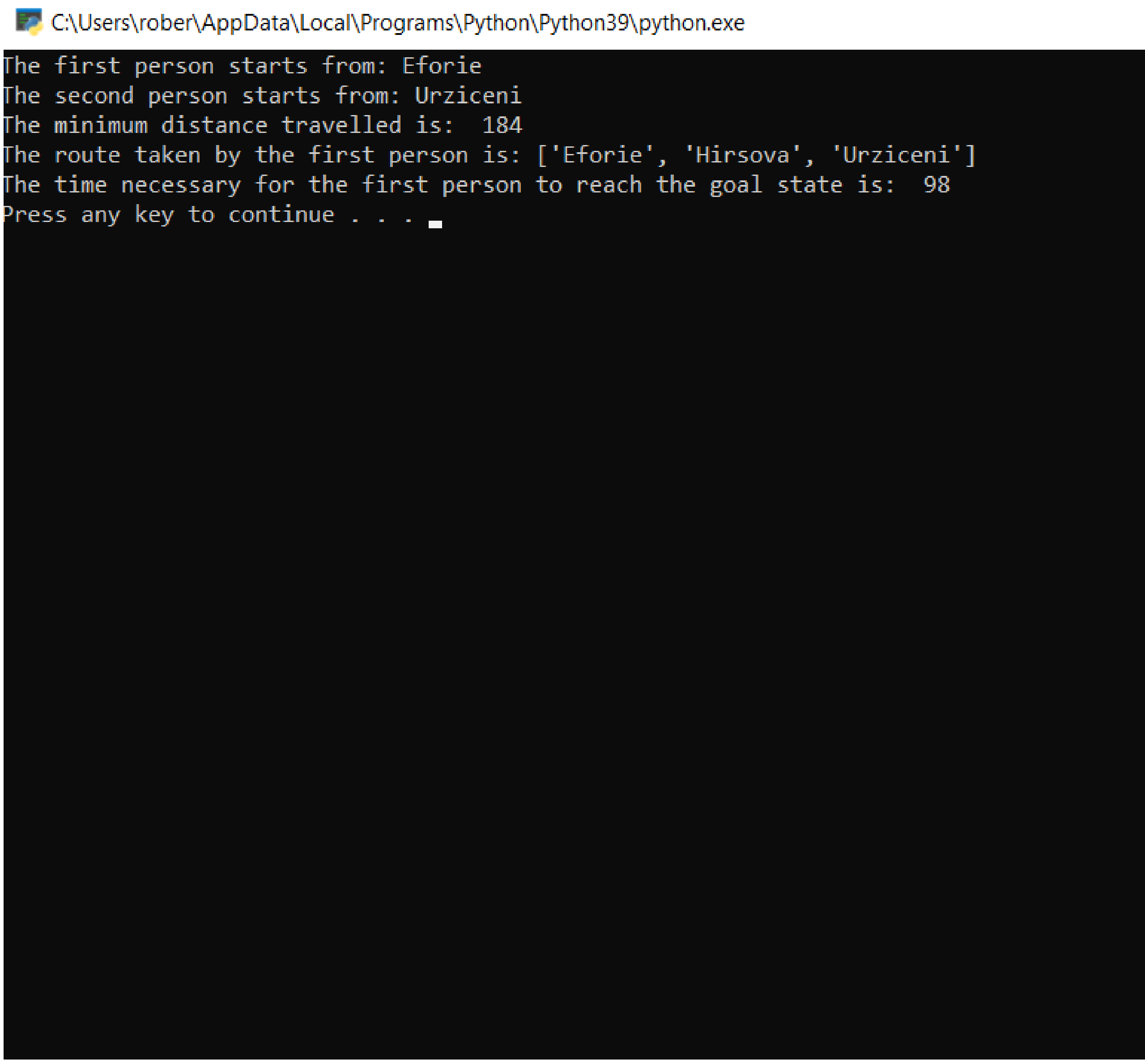
1. C1 - the city on the map from where the first person starts
2. C2 - the city on the map from where the second person starts
3. G - list of non-negative weighed edges that represent the map
4. citylist - list of names of cities from where C1 and C2 are taken

```
function Simple-Problem-Solving-Agent(percept) returns an action
 persistent: seq, an action sequence, initially empty
             state, some description of the current world state
             goal, a goal, initially null
```

```
              problem, a problem formulation
 state <- Update-State(state,percept)
 if seq is empty then
    goal <- Formulate-Goal(state)
    problem <- Formulate-Problem(state,goal)
    seq <- Search(problem)
    if seq = failure then return a null action
 action <- First(seq)
 seq <- Rest(seq)
 return action
```

The output consist of:
For example:
Eforie and Urziceni are the cities from where the 2 people start, so Eforie will be considered the starting point in the algorithm and Urziceni is the goal.

OUTPUT IS:
They will meet after 98 units, as this is the bigger distance between the two that the friends traverse.
The total distance that the first person would travel to the second is 184 so we approximate that by dividing by 2.
The route is Eforie − > Hirsova − > Urziceni so they will meet in Hirsova.



```
C:\Users\rober\AppData\Local\Programs\Python\Python39\python.exe
The first person starts from: Eforie
The second person starts from: Urziceni
The minimum distance travelled is:  184
The route taken by the first person is: ['Eforie', 'Hirsova', 'Urziceni']
The time necessary for the first person to reach the goal state is:  98
Press any key to continue . . .
```

Figure 2: Output example

5 Results and Conclusions

From running the program on at least 10 random inputs, which are presented in the folder "testdata" and where are also described the outputs, I reached the conclusion that in almost all the examples the friends meet at the half of the distance, so in the city at the middle of the route.
This project helped me understand search algorithms as well as search problems better, by using them in a more practical way and with palpable results.

6 References:

https://www.python-course.eu/networkx.php
https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/
https://en.wikipedia.org/wiki/Yen