Technical Report

Grigore Sandi-Gabriel
gabigrigore17@gmail.com
CEN 2.2 A
The University of Craiova

6^{th} of June 2021

1 Problem Statement

Suppose two friends live in different cities on a map, such as the Romania map shown in Figure 1. 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.

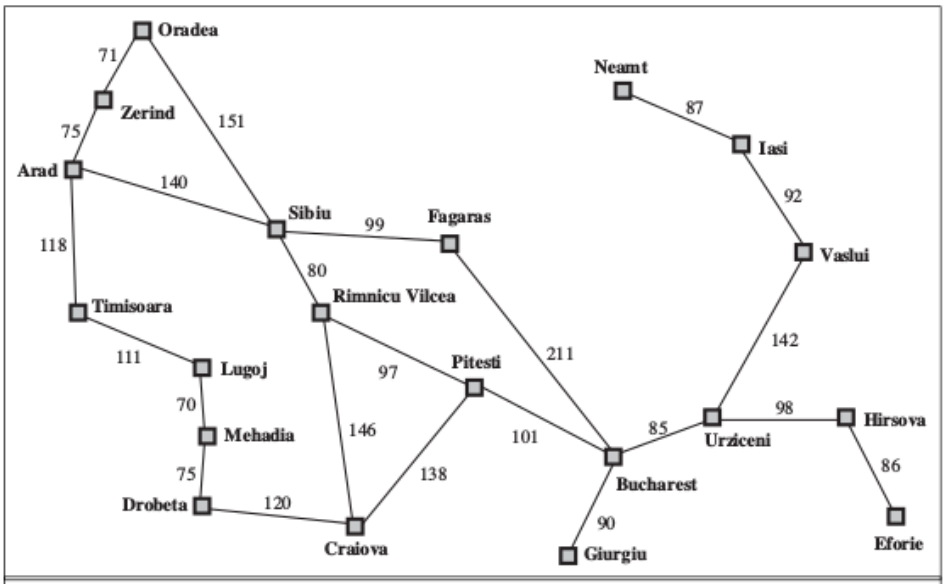

Figure 1: Romania Map

At a first glance, the statement above seems to denote a basic **"shortest path"** type of problem, but as we'll see shortly, it is more complex than it seems.

The goal is simple: **We have to place both people in the same city in the shortest amount of time possible.** One might think that the shortest path will do the trick but, even though most of the time the shortest path gives us the best time as well, not always will it be the correct answer. We have to pay attention to the fact that the two friends have to wait for each other to arrive in the next city.

Let's take the following example:

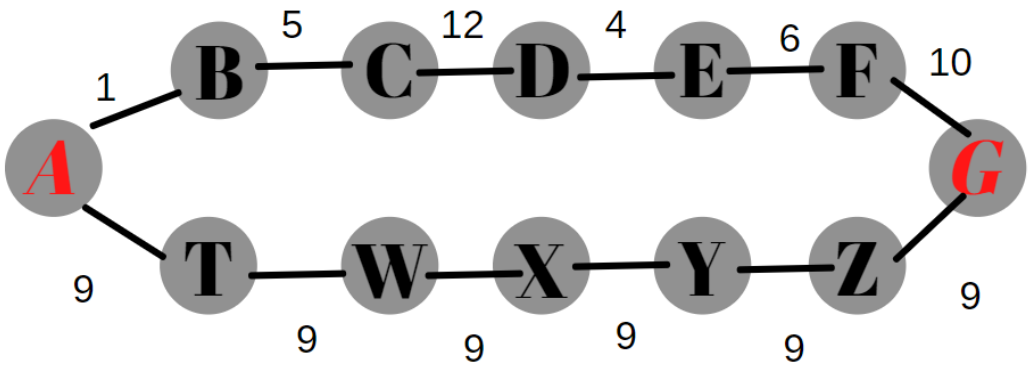

Figure 2: Route example

In this small example the route above is the shortest and yet the route below takes a smaller amount of time to traverse. This happens because the distribution of the distances is just as important as the length of the path. Every minute one of the friends has to wait for the other to arrive at their next stop is wasted.
Route above: 38km, 28 minutes
Route below: 54km, 27 minutes

To solve this problem I used a modified version of **Yen's algorithm**. Yen's algorithm computes single-source K-shortest loopless paths for a graph with non-negative edge cost.

So as to achieve minimizing the amount of paths I need to evaluate to find the one that has the optimal time, I figured I need to start from the shortest path and work my way up longer paths until a certain **condition** is met.

In order to easily understand this condition we have to lay out the following:

- The best case scenario for a path is the one in which neither of the friends has to wait for the other one at any given point. In that scenario the **path time** will be equal to half of the **distance of the path**
- There is no point in evaluating a path when its best possible time (obtained in the best case scenario) is longer than our current best time
- We only evaluate paths in ascending order in terms of distance

Considering the above statements, we can now understand why our **condition** for evaluating the minimum amount of paths possible is to **stop evaluating paths once the best possible time of the path that is currently being evaluated surpasses our current best time.**

In the example above, given than the shortest path takes 28 minutes, there cannot be any better path longer than 56km because even if none of the friends have to wait at all, it will still take longer than 28 minutes for them to meet.

Language of choice:
For this task I came to **Python** as my language of choice as I find it the most appropriate for working with large data sets and it also lets me focus more on how to solve the problem instead of how to implement the solution. It being a very high level language makes it easier to devise a concise and logical solution that is easily comprehensible by anyone who tries to read it.

2 Pseudocode of Algorithms

The following pseudocode represents the heart of the whole project, Yen's algorithm for single-source K-shortest loopless paths for a graphwith non-negative edge cost. The only modifications I made are the following: I needed an extra array to store the total distance of every path and an extra array to store the total time of every path.

```
function YenKSP(Graph, source, sink, K):
    // Determine the shortest path from the source to the sink.
    A[0] = Dijkstra(Graph, source, sink);
    // Initialize the set to store the potential kth shortest path.
    A_len = sum(edges)
    A_time = sum(max(first_edge, last_edge))

    B = [];

    for k from 1 to K:
        // The spur node ranges from the first node to
        //the next to last node in the previous k-shortest path.
        for i from 0 to size(A[k     1])     2:

            // Spur node is retrieved from the previous
            //k-shortest path, k     1.
            spurNode = A[k-1].node(i);
            // The sequence of nodes from the source to
            //the spur node of the previous k-shortest path.
            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 shortest paths which
                    //share the same root path.

                    //remove p.edge(i,i + 1) from Graph;
```

```
        for each node rootPathNode in rootPath except spurNode:
            remove rootPathNode from Graph;

        // Calculate the spur path from the spur
        //node to the sink.
        // Consider also checking if any spurPath found
        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 graph.
        restore edges to Graph;
        restore nodes in rootPath to Graph;

    if B is empty:
        // This handles the case of there being no
        //spur paths, or no spur paths left.
        // This could happen if the spur paths have
        //already been exhausted (added to A),
        // or there are no spur paths at all - such
        //as when both the source and sink vertices
        // lie along a "dead end".
        break;
    // Sort the potential k-shortest paths by cost.
    B_len = sum(edges) for every path in B
    B.sort();
    for path in B
        B_time.append(sum(max(first_edge, last_edge))
    // Add the lowest cost path becomes the k-shortest path.
    A[k] = B[0];
    A_time.append(B_time)
    A_len.append(B_len)
```

```
        // In fact we should rather use shift since we
        //are removing the first element
        B.pop();

    return A, A_len, A_time;
```

3 Application Outline

3.1 The high level architectural overview of the application

My application intends on abstracting the logic behind the scenes by providing an easy-to-use interface that is used in the **main.py** module. This interface only demands a data source in the form of a **json** file and the cities whose optimal path you need to find.

On the other hand, the real magic happens behind the scenes. Over there, the data source is read and its contents are interpreted in order to build a graph that represents our map. After that we find the shortest path (by distance) via Dijkstra's algorithm. Following this we inspect the smallest amount possible of the next shortest paths via my modified version of Yen's algorithm in order to make sure we find the most optimal (time-wise) path in the map. When all of this is over, one of the functions interprets the resulting data and outputs the most relevant information to the console.

3.2 The specification of the input data format

The input data takes the form of a **json file** and its format is the following (based on the example used above):

```
"small_sample": {
  "1": {
    "source": "A","target": "B","weight": 1
  },
  "2": {
    "source": "B","target": "C","weight": 5
  },
```

```
    "3": {
      "source": "C","target": "D","weight": 12
    },
    "4": {
      "source": "D","target": "E","weight": 4
    },
    "5": {
      "source": "E","target": "F","weight": 6
    },
    "6": {
      "source": "F","target": "G","weight": 10
    },
    "7": {
      "source": "A","target": "T","weight": 9
    },
    "8": {
      "source": "T","target": "W","weight": 9
    },
    "9": {
      "source": "W","target": "X","weight": 9
    },
    "10": {
      "source": "X","target": "Y","weight": 9
    },
    "11": {
      "source": "Y","target": "Z","weight": 9
    },
    "12": {
      "source": "Z","target": "G","weight": 9
    }
  },
```

3.3 The specification of the output data format

The output data is presented in the console and has the following format (based on the example used above):

```
The optimal route between A and G will
take 27 minutes and spans over 54 kilometers
The friends will meet in X
The path is the following:
A T W X Y Z G
```

3.4 The list of all the modules in the application and their description

The application is composed of the following modules:

- **route_generator.py**
  This is the brain of the application as it contains the main algorithm. It contains only one class that can create map-like objects (based on graphs) that can be interacted with.

- **main.py**
  This is the entry point into the application. This is where the actual maps are created and interacted with. This is the module that is supposed to be executed and played with in order to test different scenarios. Here is where I left examples ranging from small (10 cities) to very large (5000 cities).

- **data_generator.py**
  This is the module that auto-generates test data sets based on multiple adjustable parameters. Over 90% of the data sets used for experimentation in **main.py** were produced using this generator.

Other important files:

- **samples.json**
  This is the file that contains the **DEMO** data sets that can be used to quickly evaluate the efficiency of the algorithm. They have been thoroughly tested.

- **requirements.txt**
  This is the file that states which python package must be downloaded for the application to work (networkx).

- **cities.txt**
  This is a file that contains 10000 random cities extracted from an online data set. It is used for generating the data sets we need.

- **generated_data_.... .json**
  These are the data sets generated by my generator.

3.5 The list of all the functions in the application, grouped by modules

route_generator.py:

- **k_shortest_paths(self, source, target, k=1)**
  This function represents the main algorithm (my modified version of **Yen's Algorithm**. It expects a the starting node, the target node and the amount of short paths you need. It returns three ordered lists containing the paths, their lengths and the time it takes to traverse them.

- **load_data(self, json_file, data_set)**
  This function loads the data you provide into the map-like object. It expects the name of the file and the name of the section within it that you want to load in. This function does not return anything

- **optimal_route(self, source, target, k=3)**
  This function analyzes the data provided by the first function in order to find the optimal path. In order to evaluate as few paths as possible, it starts by only evaluating 3 and if those are not enough it recursively calls a version of itself that evaluates one more path than before. It expects the parameters needed by the first function and returns the parameters of the optimal route (the time, the distance and the route).

- **print_optimal_route(self, source, target)**
  This function is the one we interact with as it only demands the two cities and it provides the result of the search in a readable, easy to understand way. It does not return anything.

data_generator.py:

- **__init__(self, number_of_nodes=25, min_weight=50, max_weight=300, min_edges=25, max_edges=300)**
  This is the constructor of the generator. It lets you customize every parameter of the auto-generated data set. None of the parameters are mandatory as they have default values. It also performs check for invalid parameters.

- **json_dump_random_data(self, json_file)**
  This function generates the random data and it dumps it inside the json file that you provide. It doesn't return anything.

main.py has no functions

4 Experimental Data:

4.1 Introduction:

In order to test my application I had to build a data generator that could provide me with an infinite amount of data sets ranging from small to very large.

How it works:
The generator has its default values for the dimensions of the data set but everything can be modified by the user: the number of cities (we'll call this parameter **n**); the range in which the distance between two cities can be generated; the range in which the generator will choose a random number of roads to be built between cities (we'll call this parameter **e**).

When the parameters are set (by the user or to default), the algorithm starts the generation. From the file **cities.txt**, the algorithm randomly chooses **n** cities from the total of 10000. After that we randomly choose a pair of two cities, we check if this pair has ever been previously selected, and if not, we assign it an edge with a random weight within the desired interval. We repeat this process until we have an **e** amount of unique edges.

When this process is over, we dump the resulting **python dictionary** into a **json** file with the name specified by the user.

4.2 Experiments and results:

In this subsection I am going to present the results generated by experimenting with 10 randomly generated data sets raging in size from small to very large:

SMALL:
The results from testing generated_data_1_small.json:
Nodes: 10
Edges: 14
Execution time: 0.006s

```
The optimal route between Dearborn and Greenbriar will take 202 minutes and spans over 261 kilometers
The friends will meet in El Paso
The path is the following:
Dearborn El Paso Greenbriar

0.006349599999999955

Process finished with exit code 0
```

The results from testing generated_data_2_small.json:
Nodes: 15
Edges: 23
Execution time: 0.007s

```
The optimal route between Bargoed and Kurdamir will take 290 minutes and spans over 469 kilometers
The friends will meet in Mangalore
The path is the following:
Bargoed Madinat ash Shamal Mangalore Kedougou Kurdamir

0.007377000000000022

Process finished with exit code 0
```

The results from testing generated_data_3_small.json:
Nodes: 20
Edges: 31
Execution time: 0.006s

```
The optimal route between Balqash and Haora will take 103 minutes and spans over 204 kilometers
The friends will meet in Highwood
The path is the following:
Balqash Highwood Haora

0.006393800000000005

Process finished with exit code 0
```

MEDIUM:
The results from testing generated_data_4_medium.json:
Nodes: 40
Edges: 108
Execution time: 0.011s

```
The optimal route between Frimley and Malabon will take 245 minutes and spans over 351 kilometers
The friends will meet in Cairo
The path is the following:
Frimley Cairo Malabon

0.011151499999999981

Process finished with exit code 0
```

The results from testing generated_data_5_medium.json:
Nodes: 60
Edges: 140
Execution time: 0.008s

```
The optimal route between Dinnington and Nalut will take 170 minutes and spans over 285 kilometers
The friends will meet in Seaham
The path is the following:
Dinnington Seaham Nalut

0.008623500000000006

Process finished with exit code 0
```

The results from testing generated_data_6_medium.json:
Nodes: 80
Edges: 173
Execution time: 0.006s

```
The optimal route between Toledo and Everett will take 643 minutes and spans over 946 kilometers
The friends will meet in Malmesbury
The path is the following:
Toledo The Crossings Malmesbury Nyon Everett

0.006502700000000028
```

LARGE:
The results from testing generated_data_7_large.json:
Nodes: 150
Edges: 316
Execution time: 0.008s

```
The optimal route between Hechi and Stopsley will take 439.5 minutes and spans over 730 kilometers
The friends will meet halfway between Varzea Grande and Kamalia
The path is the following:
Hechi Varzea Grande Kamalia Stopsley

0.008495599999999992
```

The results from testing generated_data_8_large.json:
Nodes: 200
Edges: 485
Execution time: 0.007s

```
The optimal route between Charsadda and Durgapur will take 394 minutes and spans over 638 kilometers
The friends will meet in Kitamoto
The path is the following:
Charsadda Kitamoto Durgapur

0.007211100000000026

Process finished with exit code 0
```

The results from testing generated_data_9_large.json:
Nodes: 250
Edges: 427
Execution time: 0.006s

```
The optimal route between Goranboy and Uruguaiana will take 611 minutes and spans over 991 kilometers
The friends will meet in Koknese
The path is the following:
Goranboy Sabirabad San Juan Koknese Nakasongola Kovin Uruguaiana

0.006349700000000014

Process finished with exit code 0
```

The results from testing generated_data_10_very_large.json:
Nodes: 5000
Edges: 10177
Execution time: 0.006s

```
The optimal route between Monclova and Pushkino will take 581.0 minutes and spans over 1060 kilometers
The friends will meet halfway between Kremenchuk and Glodeni
The path is the following:
Monclova As Sib Kremenchuk Glodeni Cotabato Pushkino

0.006348800000000043
```

All of these tests are available in main.py and can be evaluated by the reader. Execution time may vary.

5 Conclusions:

As to conclude this assignment I have to mention that I really enjoyed the process of building this project from the ground up. The hardest part by far was to translate my ideas into python code because even though I had no problem understanding how the problem worked, making the computer understand how the problem works is never as easy.
My favorite part of the whole project was the research. During the research I got to better my understanding of graph theory as well as discovering a lot of different search algorithms by reading dissertations, articles and online forum posts. I also got to interact with people over the internet who've had similar problems in the past. That is how I stumbled upon Yen's Algorithm, the heart of my application. I don't think I will extend this assignment further but I will for sure use the knowledge I needed to aquire to solve it in the future.

6 References:

1. Yen's Algorithm **https://en.wikipedia.org/wiki/Yen%27s_algorithm**

2. Dijkstra's algorithm with path **https://www.geeksforgeeks.org/printing-paths-dijkstras-shortest-path-algorithm/**

3. Python interpretation of Yen **https://github.com/beegeesquare/k-shortest-path**

4. Networkx **https://networkx.org/**