07.08.2019

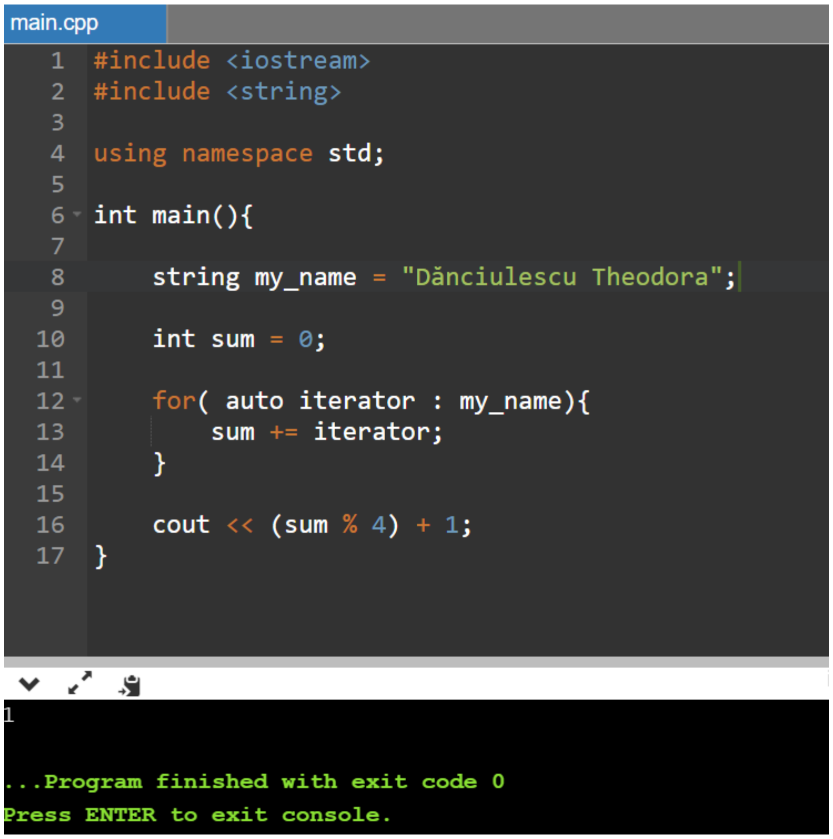
Artificial Intelligence Course Assignment

Dănciulescu Theodora-Ioana

2nd Year, Group 2.1A

CE

Hash function for choosing problem


main.cpp

```cpp
#include <iostream>
#include <string>

using namespace std;

int main(){

    string my_name = "Dănciulescu Theodora";

    int sum = 0;

    for( auto iterator : my_name){
        sum += iterator;
    }

    cout << (sum % 4) + 1;
}
```

```
1

...Program finished with exit code 0
Press ENTER to exit console.
```

In order to find which problem is assigned to me I had to compute a program that calculates the sum of the ascii codes of the letters in my family name and first name. The simple method of doing this is to simply iterate through the string that represents my name as described above, and adding the ascii code of each character to an integer, called 'sum'. After iterating through the whole string, the number of the problem will be obtained by the formula: (sum % 4) + 1. A code snippet of the is attached below:

In conclusion, the problem I had to solve was the problem with number 1.

Problem Statement

The statement is presenting the problem of two friends living in Romania in different cities.

The two friends want to see each other as soon as possible in a specific city. That's why they need a software developer to provide each of them the shortest path in distance and also in time. The distance between any two cities is also representing the time required to traverse the road between the two cities.

Given the map of Romania and the road distances between the cities, an optimum path has to be computed for each friend in order to reach the established destination. Also, beside the optimum path, they would like to know how much the first arrived friend would have to wait for the other one if they leave their home cities simultaneously.

In order to proceed with these actions there will be set:

Initial_state: { eg: Sibiu}
Goal_state: { eg.Bucharest}
    Path: ['Rimnicu', 'Pitesti', 'Bucharest']
    Cost: 278

Problem approach

1. Programming Language

I chose to implement the problem in Python because it is a popular language in the domain of Artificial Intelligence. Also, I like coding in Python for the implementation simplicity, the variety of default functions, the easy way of declaring and managing complicated data structures. Another reason for choosing Python was for adapting the laboratory framework to my problem easily.

2. Search algorithm strategy

The key idea for an optimal solution with respect to the usage of memory and also the time performance includes a heuristic search algorithm.

The A$*$ Search algorithm performs better than the Dijkstra's algorithm because of its use of **heuristics.**

The cities of the romanian map will represent the nodes in an undirected graph, and the distances between the cities will represent the cost of the edges. Using the A$*$ search strategy the a path from a given city will be computed to the goal city. The correctitude of the result will be checked performing Dikstra's algorithm on the same graph.

Modules

The application contains 7 modules that contributes to a good organisation of the application's functionality. In order to solve the problem using heuristic search strategy there will be used a set of self-defined classes for defining the problem, these classes were also used at the laboratory sessions for solving other AI problems. These classes will be defined in a separate module. The specific class for our problem is a class inherited from the base class 'Problem'. Several function will be overwritten in order to obtain the right functionality.

Using the classes used in the laboratory framework: 'Problem', 'Node', 'PriorityQueue' and other functions such like 'best_first_graph_search', 'memoize', 'astar_search' the solution of the given problem will be computed.

Inheriting from the base general class 'Problem', I created the 'FriendsMeetingProblem' (this class is created based on the skeleton of the 'GraphProblem' class from the laboratory framework) class that will generate an initial_state, goal_state(destination) also it contains functions for generating the possible_actions. In our case the possible_actions of a certain state represents the adjacent nodes of the current node. Also, the function that generates the result of a given state will return nothing else than the neighbour of that node.

The purpose of this application is to compute an optimal path regarding the cost of the path for each of the two friends.

Input/Output example

Input

The input is represented by 3 cities with the following meanings:

City1 - the initial state of friend A

City2 - the initial state of friend B

City3 - the meeting point of the 2 friend

The input is randomly generated by one of the program modules.

Take the following input:

City1 <- Fagaras

City2 <- Oradea

City <- Eforie

Output

The output is designated to:

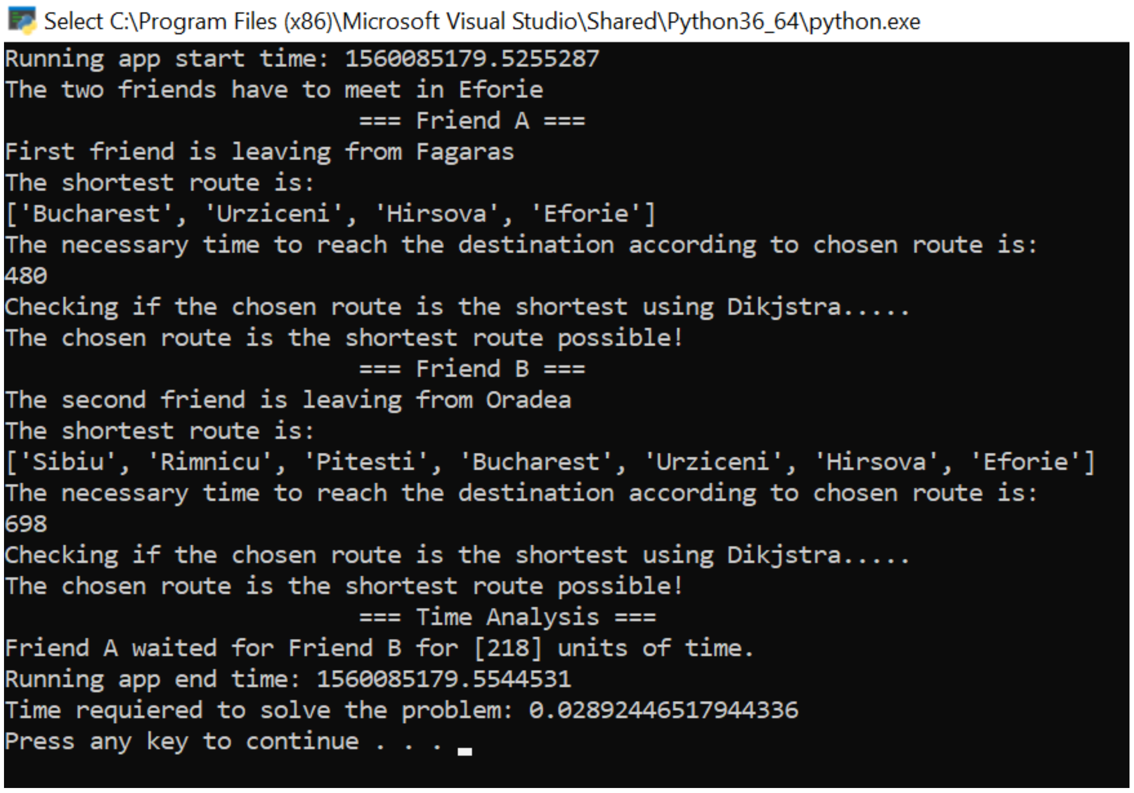
- display the solution path of every friend

- display the initial data of the problem

- display if the found path is the correct one (verIfied with Dikstra's algorithm)

- display the amount of time the first arrived friend will wait for the other one

- display an execution time analysis of the program


Select C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe

```
Running app start time: 1560085179.5255287
The two friends have to meet in Eforie
                         === Friend A ===
First friend is leaving from Fagaras
The shortest route is:
['Bucharest', 'Urziceni', 'Hirsova', 'Eforie']
The necessary time to reach the destination according to chosen route is:
480
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Friend B ===
The second friend is leaving from Oradea
The shortest route is:
['Sibiu', 'Rimnicu', 'Pitesti', 'Bucharest', 'Urziceni', 'Hirsova', 'Eforie']
The necessary time to reach the destination according to chosen route is:
698
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Time Analysis ===
Friend A waited for Friend B for [218] units of time.
Running app end time: 1560085179.5544531
Time requiered to solve the problem: 0.02892446517944336
Press any key to continue . . .
```

A brief description of every module and the pseudocode for the new functions added by me are presented below:

1. RomaniaMapBuild.py

This module contains the data structures containing data about the Romania map. These data structures are used globally in the application as the Romania map data is constant.

More precisely, contains the following:

-a list with unique occurrences of all the cities

-a dictionary that represents the adjacency list of the graph represented by the links between the cities from Romania with the correspondent costs

-a dictionary with the plane coordinates of every city.

-Node instances of the cities are created for further implementation simplicity.

Data structures used

- **romania_map[start_city][destination_city]=road_cost** : adjacency dictionary of the roads between the cities(nodes) represents keys and the cost between the two cities as the value.
- **romania_map_coordinates[city] = (x, y)** : dictionary with the XOY coordinates of every city in the scaled map
- **city_list** : list with unique Node instances of the cities.

2. input_generator.py

This module contains a function that randomly chooses the initial_state of every friend and the meeting point of the two friends. This is done shuffling the list of the Node instances of the cities and then choosing the three cities with their specific meaning.

Pseudocode

```
function input_generator() returns tuple of nodes

          persistent: copy_city_list, copy of list of nodes
                    in map
                    start1, Node instance
                    start2, Node instance
                    destination,Node instance
          copy_city_list <- city_list
          shuffle(copy_city_list)
          returns (start1, start2, destination)
```

3. test_module.py

This module contains the function that verifies the result computed by the A∗ search with the result computed by the Dijkstra's algorithm. The functions used:

- dijkstra(start_node, end_node) -Dijkstra's shortest path algorithm isan algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. This algorithm guarantees to find the shortest path between any 2 given nodes in { \displaystyleO(| E| +| V| \ log | V|)}

  is the number of edges) as it is implemented using a min PriorityQueue.

  - Functionality : This function returns the minimum path cost computed with Dijkstra's from start_node to end_node.
  - Parameters:

    -start_node : source Node instance, the function will find all the minimum paths from this node to all the other nodes.

    -end_node : the Node instance that we are interested in finding the shortest path to reach it

  - Return value: the return value is an integer representing the minimum

    path cost from the start_node to the end_node.

- test_function(astar_cost, start_node, end_node) - this function return a bool that verifies if the cost computed by the A∗ search is identical with the cost computed by Dikjstra's algorithm.

- Functionality: compares the cost generated by A∗ search with the cost generated by Dijkstra's algorithm.
- Parameters:

  - astar_cost: cost resulted from the A∗ search

  -start_node : source Node instance, the function will find all the minimum paths from this node to all the other nodes.

  -end_node : the Node instance that we are interested in finding the shortest path to reach it

- Return value: the returned value is a boolean that points out if the

  result computed using heuristic search strategy is correct or not.

Pseudocode

```
function dijkstra(start_node, end_node) returns of integer
    persistent: distances, dictionary of distances from start_node
to all the other cities
        q, default min priority queue of pairs of
        (cost, destination)
    for i in city_list do
        distances[i] = MAX_VALUE
    distances[start_node] = 0
    q.insert(pair(0, start_node))
    while q is not empty do
        current <- q.top()
        current_city <- current[1]
        for neighbour in romania_map[current_city] do
            if distances[neighbour] > distances[current_city] +
                    romania_map[current_city][neighbour] then
                distances[neighbour] = distances[current_city] +
                        romania_map[current_city][neighbour]
                q.insert(pair(distances[neighbour], neighbour))

        returns destination[end_node]
```

```
function test_function(astar_cost, start, destination) returns bool value
    persistent: astar_ cost, integer representing the cost generated by the astar
        solution
        start, Node instance
        destination, Node instance
    returns (astar = dijkstra(start, destination)
```

4. OutputFormat.py

This module's target is to group the output functionalities into functions that ensures an understandable output for any user.

- run_app(start_friend1, start_friend2, meeting_point)

1) Functionality: This functions aims to create all the necessary instances for running the application. There will be created two 'FriendsMeetingProblem' instances, in order to compute two optimal paths for each friend. Having these 2 instances, the astar_search function defined in the utils.py module, the same function used in the solving of the NPuzzleProblem, is called on the two created instances. As a result of the A∗ search algorithm the paths of the 2 friends are created, and using more output function calls, the result will be transmitted in the Output Console in a very understandable way of a result. This function is also containing time execution analysis that will be displayed to the user, too.
2) Parameters:
    a) Start_friend1 -Node instance, initial state node of Friend A
    b) Start_friend2 - Node instance, initial state node of Friend B
    c) Meeting_point - Node instance, meeting node of the 2 friends

- output_route_analytics(path, cost)

1) Functionality: To print in a nice way a path and its cost
2) Parameters:
    a) Path : list with strings representing the path of a friend
    b) Cost: the cost of the path

- output_correctitude_check(cost_astar, start, destination)

a) Functionality: To print in a nice way if the result computed by A∗ search is correct using Dijkstra's to verify the resulted cost

b) Parameters:

c) cost_astar: integer representing the cost computed by A∗ search
d) start: the initial source node from where to start the path
e) Destination: the target node where the path ends

- output_time_waited(first_to_destination, second_to_destination, time1, time2)

a) Functionality: To print the time the first arrived friend has to wait until the second friend arrives.

b) Parameters:

- First_to_destination : is string representing either “Friend A” or “Friend B” , depending on the generated result
- Second_to_destination : is string representing either “Friend A” or “Friend B” , depending on the generated result
- Time1: represents the time required for the first arrived friend to traverse the path (the cost of the road)
- Time2: represents the time required for the second arrived friend to traverse the path (the cost of the road)

Pseudocode

```
function run_app(start_friend1, start_friend2, meeting_point)
    persistent: start_friend1, Node instance
                start_friend2, Node instance
                meeting_point, Node instance

    print("The two friends have to meet in " + meeting_point)
    friend1 <- FriendsMeetingProblem(start_friend1, meeting_point)
    path1 <- astar_search(friend1).solution()
    journey_time1 <- 0
    past <- start_friend1
    for i in path1 do
        journey_time1 <- journey_time1 + romania_map[past][i]
        past <- i
    print(" === Friend A ===")
    print("First friend is leaving from " + start_friend1)
    output_route_analytics(path1, journey_time1)
    output_correctitude_check(journey_time1, start_friend1, meeting_point)
    friend2 <- FriendsMeetingProblem(start_friend2, meeting_point)
    path2 <- astar_search(friend2).solution()
```

```
  journey_time2 <- 0
  past <- start_friend2.state
  for i in path2 do
      journey_time2 <- journey_time2 + romania_map[past][i]
      past <- i
  print(" === Friend B ===")
  print("The second friend is leaving from " + start_friend2)
  output_route_analytics(path2, journey_time2)
  output_correctitude_check(journey_time2, start_friend2, meeting_point)
  if journey_time1 < journey_time2 then
      first_friend <- "Friend A"
      second_friend <- "Friend B"
  else
      first_friend <- "Friend B"
      second_friend <- "Friend A"
  output_time_waited(first_friend, second_friend, journey_time1, journey_time2)


function output_route_analytics(path, cost)
    persistent: path, list with the cities in the path
                cost, integer as the cost of the path


    print("The shortest route is: ")
    print(path)
    print("The necessary time to reach the destination according to chosen route is: ")
    print(cost)


function output_correctitude_check(cost_astar, start, destination)
    persistent: astar_ cost, integer representing the cost generated by the astar
```

```
                    solution
                    start, Node instance
                    destination,Node instance


print("Checking if the chosen route is the shortest using Dikjstra.....")
if test_function(cost_astar, start, destination) = 1 then
    print("The chosen route is the shortest route possible!")
else:
    print("The chosen route is NOT the shortest route possible!")


function output_time_waited(first_to_destination, second_to_destination, time1, time2)
    persistent:first_ to_ destination, string with the friend that arrived first
               second_ to_ destination, string with the friend that arrived second
time1, integer with the time needed of the first friend to arrive
               time2, integer with the time needed of the second friend to arrive
    print(" === Time Analysis ===")
    difference <- | time1 - time2|
    if difference = 0 then
            print("The two friends arrived simultaneously to the destination!
Great!")
    else
            print(first_to_destination + " waited for " + second_to_destination + "for
    [" + difference + "] units of time.")
```

5. problem1.py

This module contains the class that represents the definition of problem 1 from the course assignment. The class is called “FriendsMeetingProblem” and is inherited from the base class “Problem” from the laboratory framework. The “FriendsMeetingProblem” is implementing functions that serves for solving a graph search AI problem. In other words, the “FriendsMeetingProblem” is the “GraphProblem” class from the laboratory framework but some of its functions implementation are slightly modified in order to match my problem requirements.

Functions of the class and the modification done at each function:

1) **The class constructor** - has two parameters representing the intial_state and the goal_state, meaning the city to leave from, respectively the destination city. The parameters are 'Node' instances whose states are the string representing the city name. This constructor simply creates a 'Problem' instance with the given initial_state and goal_state.

2) **actions(A)** - this function creates a list with the possible action from a given state. A given state represents the current city and the possible actions are the neighbour cities of the current city. From a graph point of view, this means that the possible action of a given Node 'A' are the nodes from his adjacency list. The return value of this function is a list with all the Nodes from the adjacency list of the current node 'A'.

3) **result(state, action)** - the result of going to a neighbor is just that neighbor. This means that state represents the current node, and action represents one of its neighbours/adjacent node. The result of such an action is the neighbour Node itself, so the returned value is the 'action' Node itself.

4) **find_min_edge(current)** - find the minimum value of edges connected to the 'current' node. 'Current' is a Node instance, and the returned value of this function is an integer representing the lowest cost of all the edges connected to 'current' Node.

5) **h(node)** - this function represents our problem heuristic, in other words the function the approximates our searching (the function that makes A∗ search better than Dijkstra's). 'node' represents a Node , and the returned value is the plane distance between the city 'node' and the goal Node. This distance is calculated using the information from the 'romania_map_coordinates' and the used formula is:

   Let A(x1, y1), B (x2, y2)

   D = sqrt((x1∗ x1 - x2∗ x2) + (y1∗ y1 + y2∗ y2))

6. utils.py

This module contains the classes and function extracted from the laboratory framework and that were not at all modified. I used the classes 'Problem', 'Node', 'PriorityQueue' and the functions 'best_first_graph_search', 'memoize', 'astar_search'.

The reason I needed this classes/functions:

- **'Problem'** - is necessary for inheriting from it. Base class for class 'FriendsMeetingProblem'
- **'Node'** - class necessary to create instances that represents nodes in a search tree. Contains a pointer to the parent (the node that this is a successor of) and to the actual state for this node. Note that if a state is arrived at by two paths, then there are two nodes with the same state. Also includes the action that got us to this state, and the total path_cost (also known as g) to reach the node. This class is also used for memorizing the right path in the search tree.

- **‘PriorityQueue’** - is a class that is necessary for implementing the ‘best_first_search_algorthm’. It is implementing the operation of a min/max priority queue, or in other words, it implements a min/max heap.
- **‘best_first_graph_search’** - this graph search algorithm is similar to Dijkstra’s algorithm but better, because it takes as an argument a function ‘f’ that represents an estimation from an initial_state to a goal_state. This also means that there will be computed a greedy best_first_search, This algorithm searches the nodes with the lowest ‘f’ scores (Dijkstra was searching the nodes with lowest edge cost). Just like Dijkstra’s, it is used a priority queue to store the reached nodes. Here, calling the ‘memoize’ function, in the nodes will also be stored their computed ‘f’ score.
- **‘memoize’** -make it remember the computed value for any argument list
- **‘astar_search’** - the key function of the program. It takes as an argument a problem class instance. Then, using ‘memoize’ call for the heuristic function given as a parameter or as defined function in the ‘Problem’ subclass. It calls the ‘best_first_graph_search’ on the given instance of the ‘Problem’ subclass and also using a heuristic that represents the path cost accumulated until a given point plus the estimation defined in the heuristic.

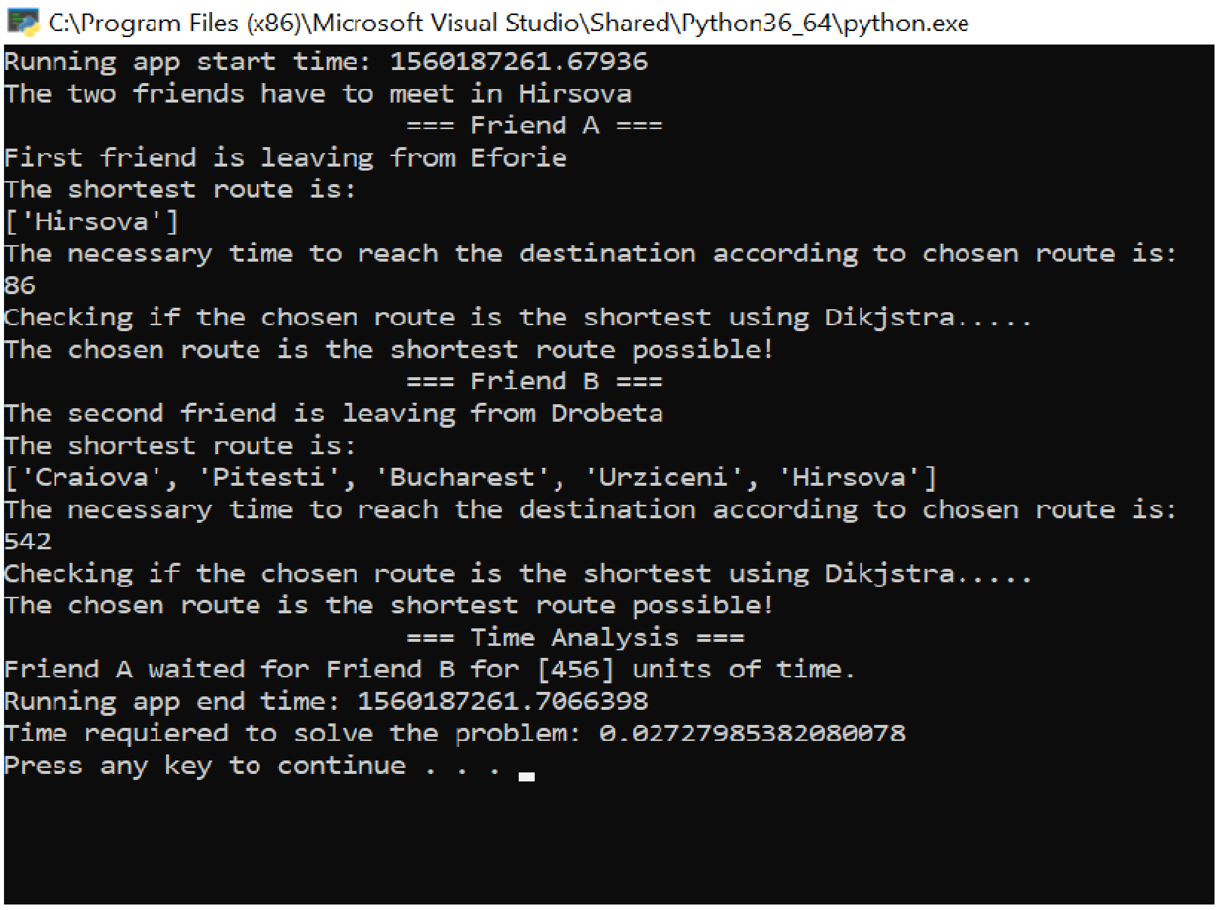
7. main.py

The main module purpose was to start the application functionality. I tried to have as few function calls as possible and also little extra functionalities defined here.

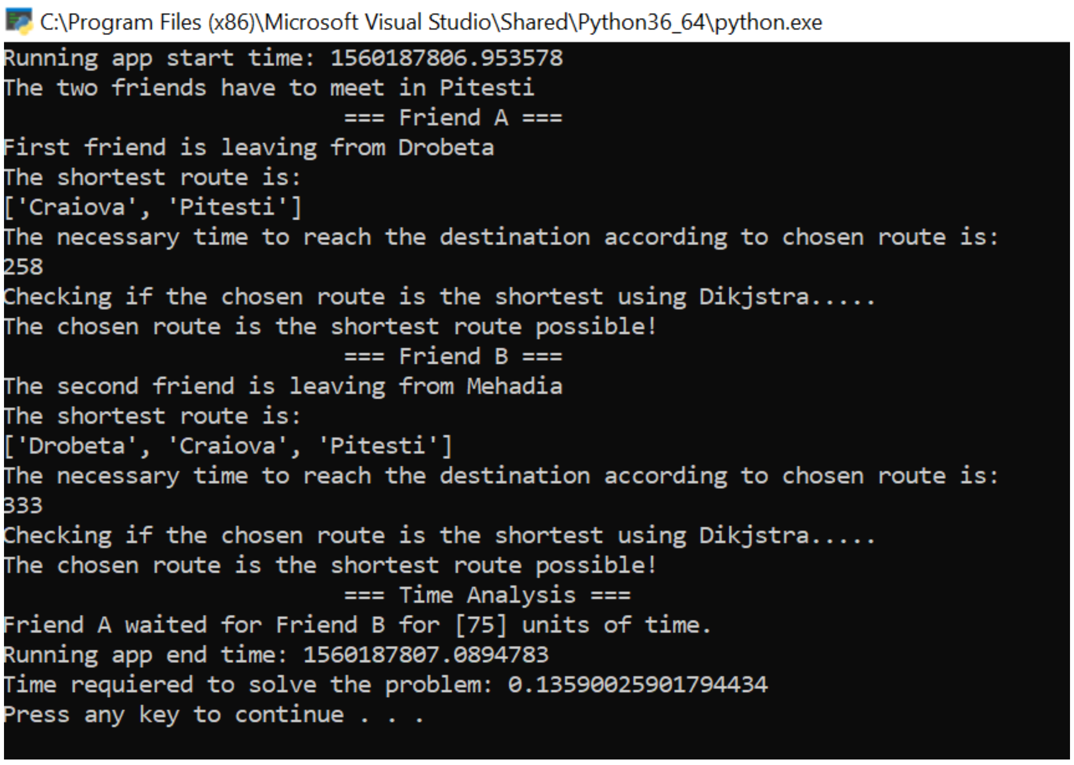
The main module is simply calling the ‘input_generator’ function in order to get the input necessary to run the application. After that, the function that basically does “all the hard job” is the ‘run_app’ function. The extra functionality added here is that I took time variables that are also displayed to see the time performance of the execution.

Experiments and results

As the output contains all the necessary analysis factors of an experiment, in this section I will add some Output examples with different inputs. The analysis of every execution is done in the Output Console itself.

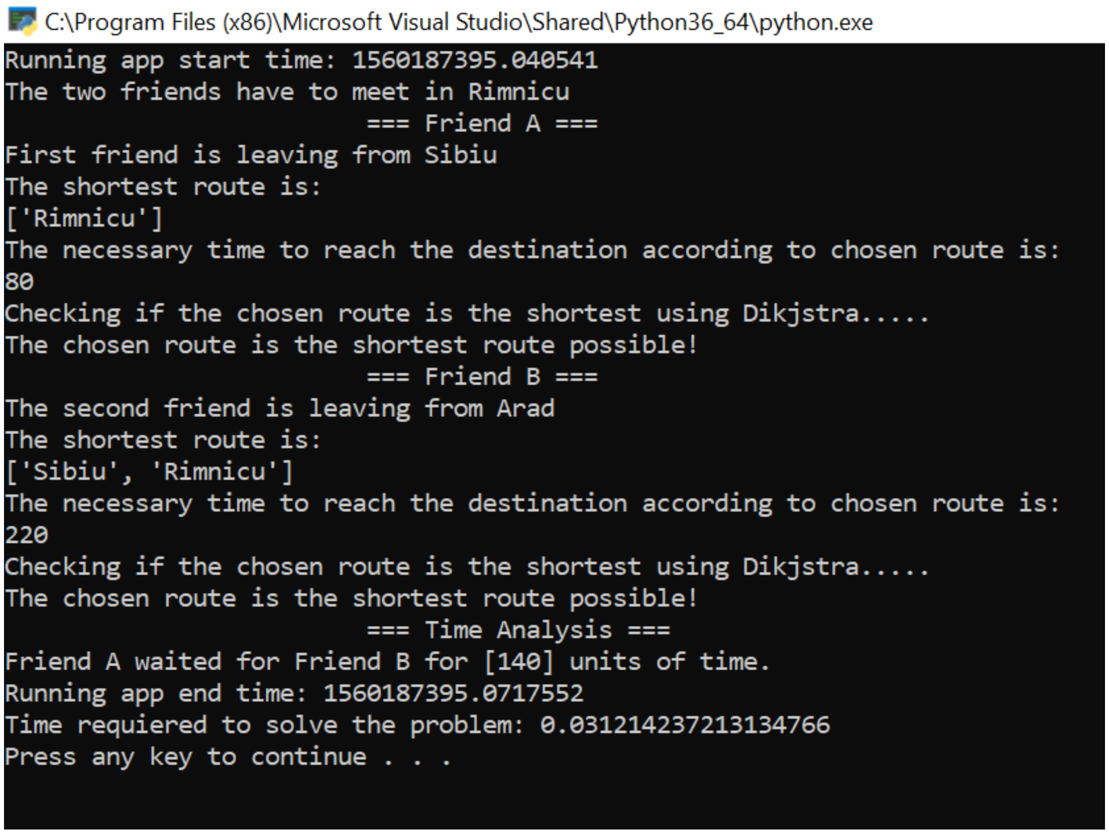


```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe
Running app start time: 1560187261.67936
The two friends have to meet in Hirsova
                          === Friend A ===
First friend is leaving from Eforie
The shortest route is:
['Hirsova']
The necessary time to reach the destination according to chosen route is:
86
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                          === Friend B ===
The second friend is leaving from Drobeta
The shortest route is:
['Craiova', 'Pitesti', 'Bucharest', 'Urziceni', 'Hirsova']
The necessary time to reach the destination according to chosen route is:
542
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                          === Time Analysis ===
Friend A waited for Friend B for [456] units of time.
Running app end time: 1560187261.7066398
Time requiered to solve the problem: 0.02727985382080078
Press any key to continue . . .
```



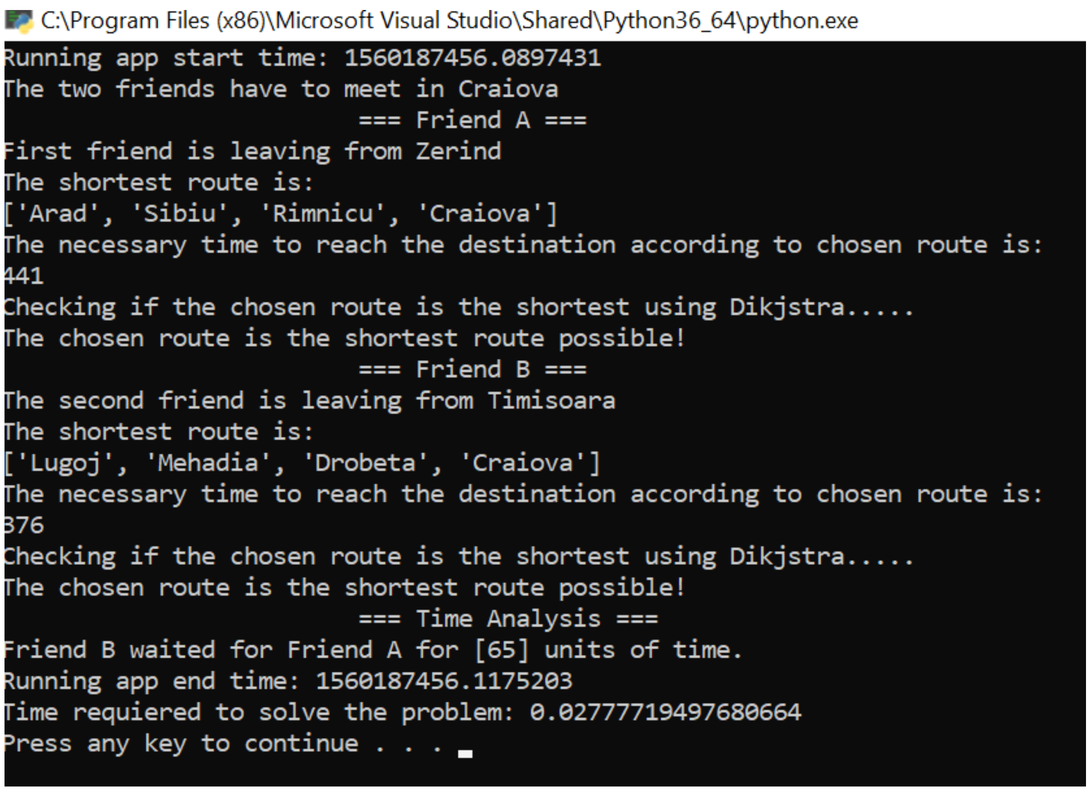
```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe
Running app start time: 1560187806.953578
The two friends have to meet in Pitesti
                         === Friend A ===
First friend is leaving from Drobeta
The shortest route is:
['Craiova', 'Pitesti']
The necessary time to reach the destination according to chosen route is:
258
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Friend B ===
The second friend is leaving from Mehadia
The shortest route is:
['Drobeta', 'Craiova', 'Pitesti']
The necessary time to reach the destination according to chosen route is:
333
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Time Analysis ===
Friend A waited for Friend B for [75] units of time.
Running app end time: 1560187807.0894783
Time requiered to solve the problem: 0.13590025901794434
Press any key to continue . . .
```

-



C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe

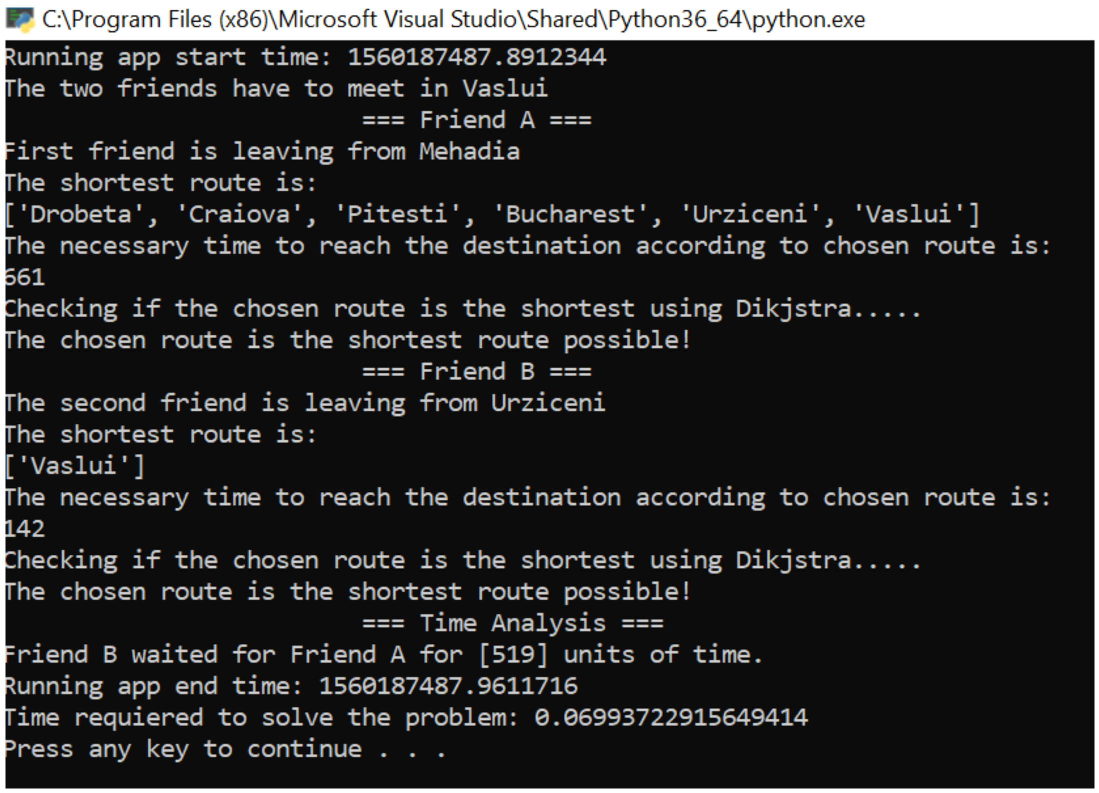
```
Running app start time: 1560187395.040541
The two friends have to meet in Rimnicu
                         === Friend A ===
First friend is leaving from Sibiu
The shortest route is:
['Rimnicu']
The necessary time to reach the destination according to chosen route is:
80
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Friend B ===
The second friend is leaving from Arad
The shortest route is:
['Sibiu', 'Rimnicu']
The necessary time to reach the destination according to chosen route is:
220
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Time Analysis ===
Friend A waited for Friend B for [140] units of time.
Running app end time: 1560187395.0717552
Time requiered to solve the problem: 0.031214237213134766
Press any key to continue . . .
```



C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe

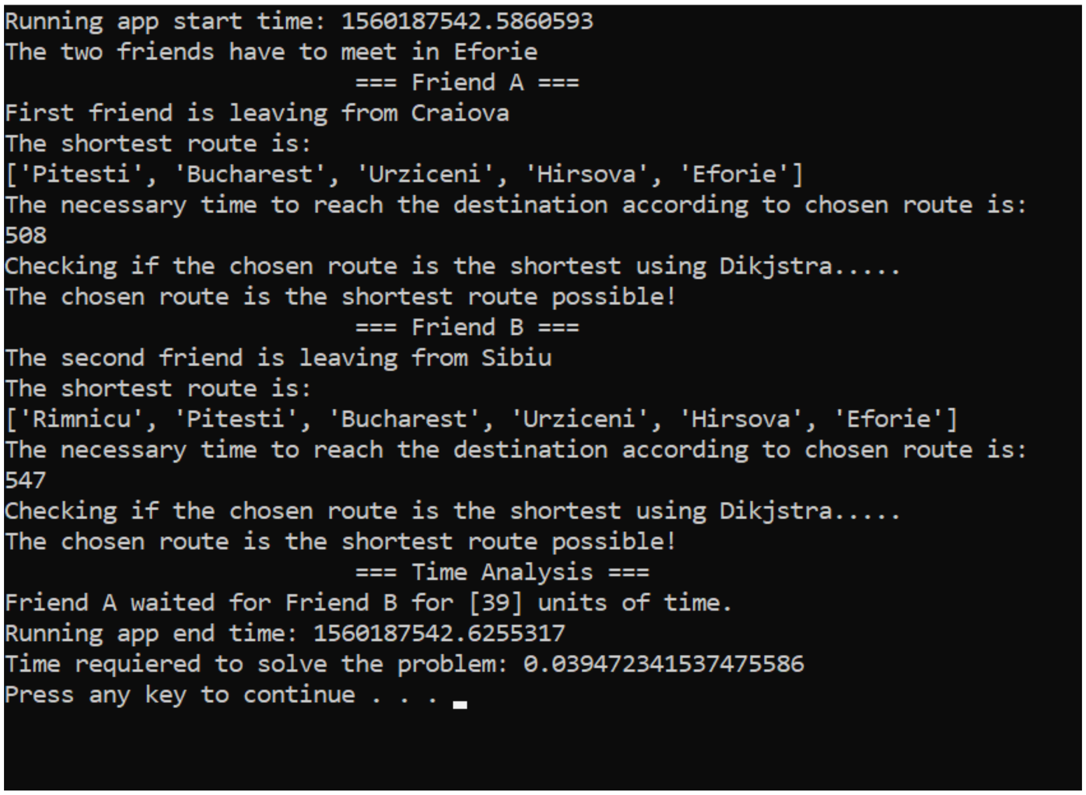
```
Running app start time: 1560187456.0897431
The two friends have to meet in Craiova
                         === Friend A ===
First friend is leaving from Zerind
The shortest route is:
['Arad', 'Sibiu', 'Rimnicu', 'Craiova']
The necessary time to reach the destination according to chosen route is:
441
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Friend B ===
The second friend is leaving from Timisoara
The shortest route is:
['Lugoj', 'Mehadia', 'Drobeta', 'Craiova']
The necessary time to reach the destination according to chosen route is:
376
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Time Analysis ===
Friend B waited for Friend A for [65] units of time.
Running app end time: 1560187456.1175203
Time requiered to solve the problem: 0.02777194976806664
Press any key to continue . . .
```

-



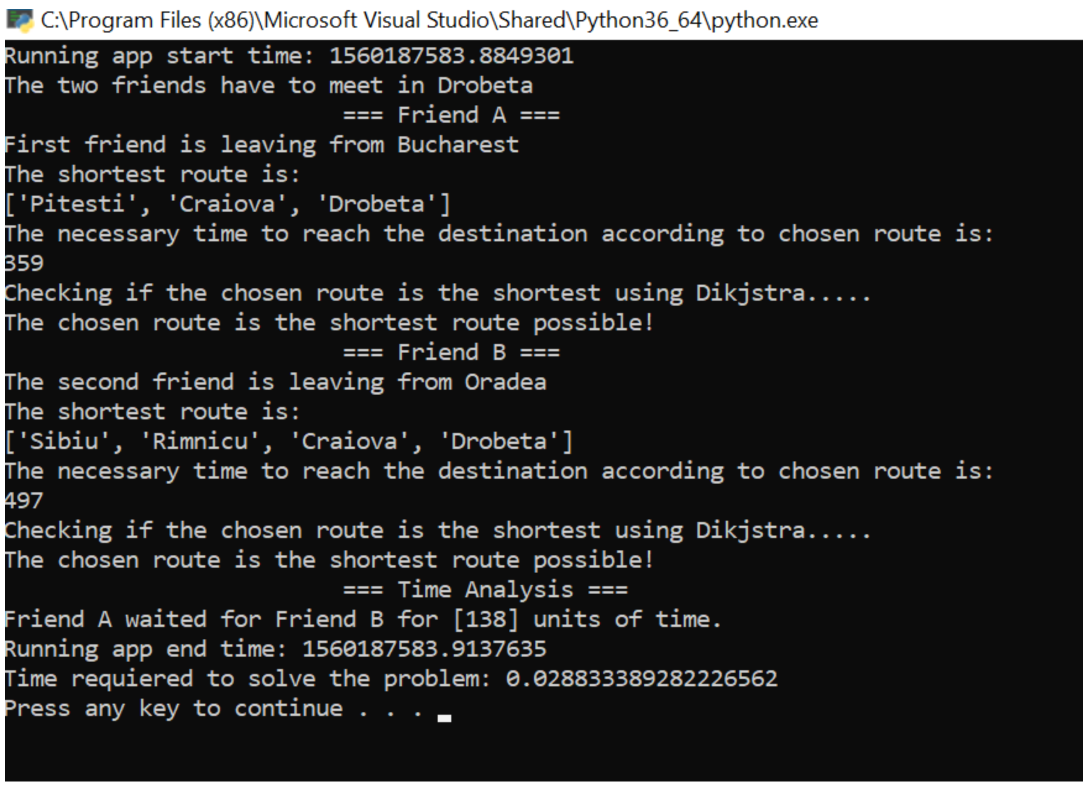
```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe
Running app start time: 1560187487.8912344
The two friends have to meet in Vaslui
                         === Friend A ===
First friend is leaving from Mehadia
The shortest route is:
['Drobeta', 'Craiova', 'Pitesti', 'Bucharest', 'Urziceni', 'Vaslui']
The necessary time to reach the destination according to chosen route is:
661
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Friend B ===
The second friend is leaving from Urziceni
The shortest route is:
['Vaslui']
The necessary time to reach the destination according to chosen route is:
142
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Time Analysis ===
Friend B waited for Friend A for [519] units of time.
Running app end time: 1560187487.9611716
Time requiered to solve the problem: 0.06993722915649414
Press any key to continue . . .
```

-
-



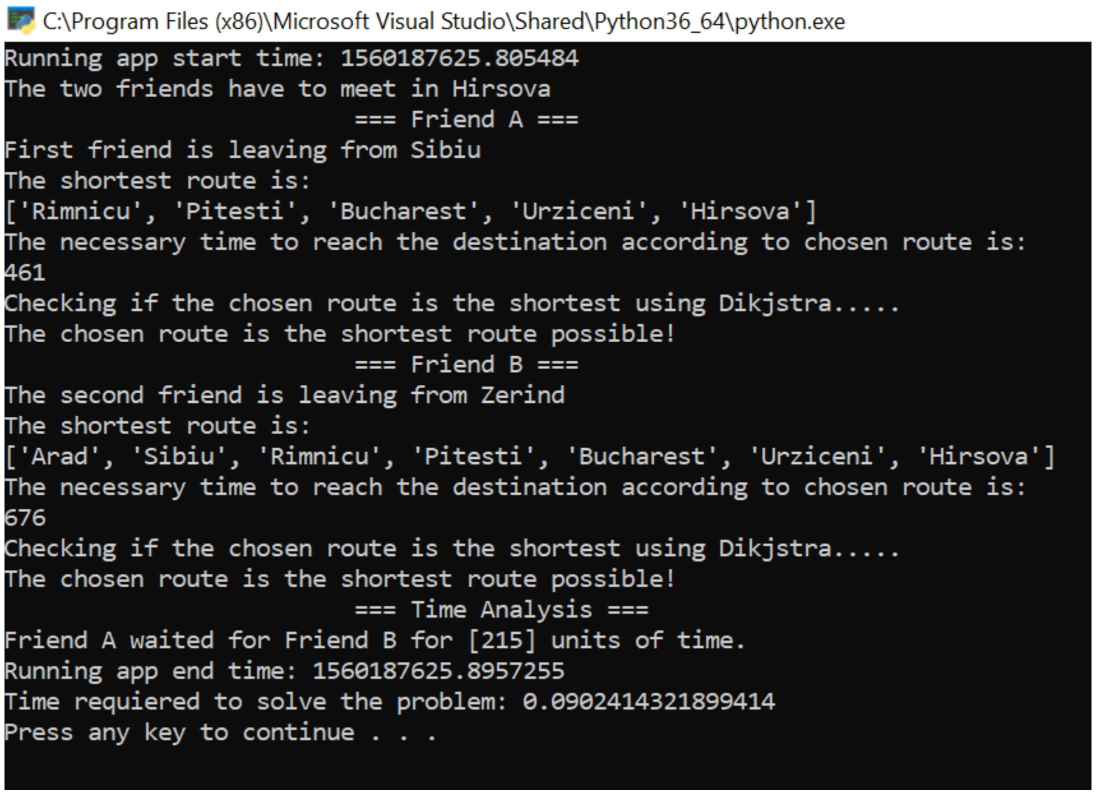
```
Running app start time: 1560187542.5860593
The two friends have to meet in Eforie
                         === Friend A ===
First friend is leaving from Craiova
The shortest route is:
['Pitesti', 'Bucharest', 'Urziceni', 'Hirsova', 'Eforie']
The necessary time to reach the destination according to chosen route is:
508
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Friend B ===
The second friend is leaving from Sibiu
The shortest route is:
['Rimnicu', 'Pitesti', 'Bucharest', 'Urziceni', 'Hirsova', 'Eforie']
The necessary time to reach the destination according to chosen route is:
547
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Time Analysis ===
Friend A waited for Friend B for [39] units of time.
Running app end time: 1560187542.6255317
Time requiered to solve the problem: 0.039472341537475586
Press any key to continue . . .
```

-



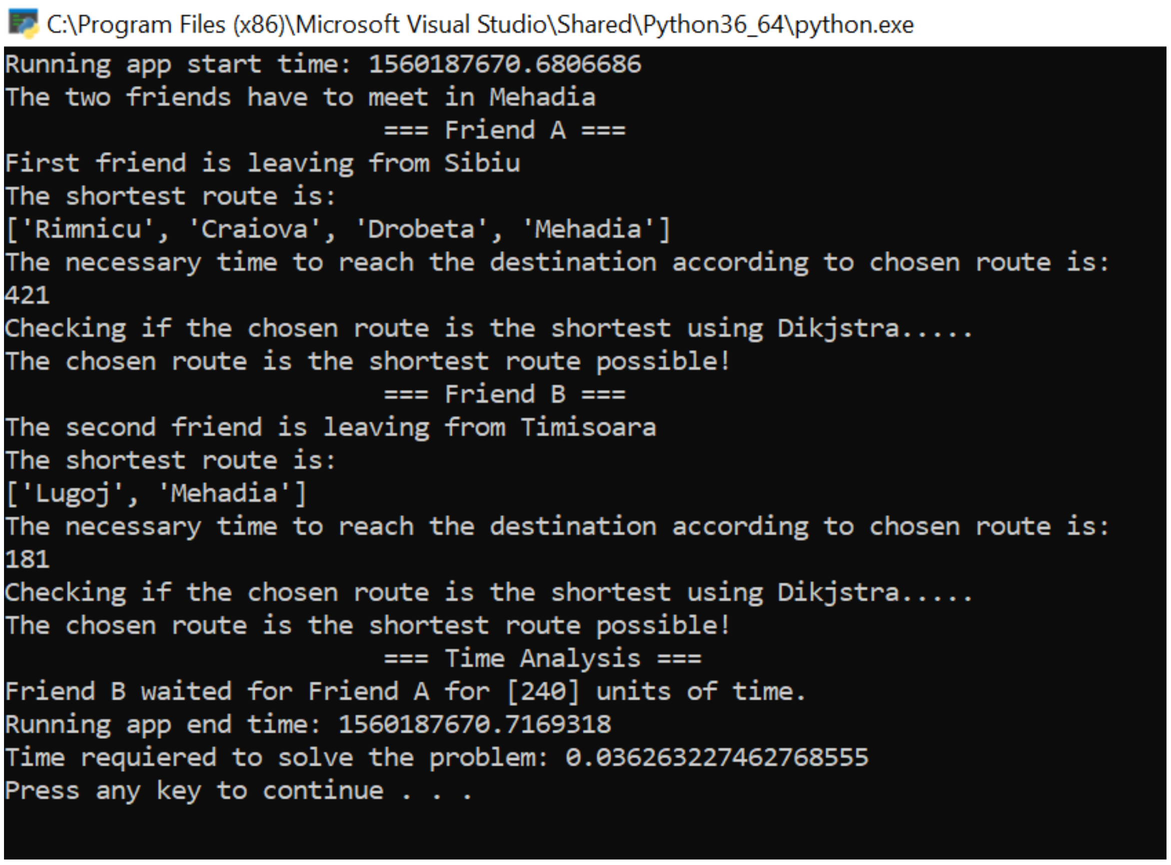
```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe
Running app start time: 1560187583.8849301
The two friends have to meet in Drobeta
                         === Friend A ===
First friend is leaving from Bucharest
The shortest route is:
['Pitesti', 'Craiova', 'Drobeta']
The necessary time to reach the destination according to chosen route is:
359
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Friend B ===
The second friend is leaving from Oradea
The shortest route is:
['Sibiu', 'Rimnicu', 'Craiova', 'Drobeta']
The necessary time to reach the destination according to chosen route is:
497
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Time Analysis ===
Friend A waited for Friend B for [138] units of time.
Running app end time: 1560187583.9137635
Time requiered to solve the problem: 0.028833389282226562
Press any key to continue . . .
```

•



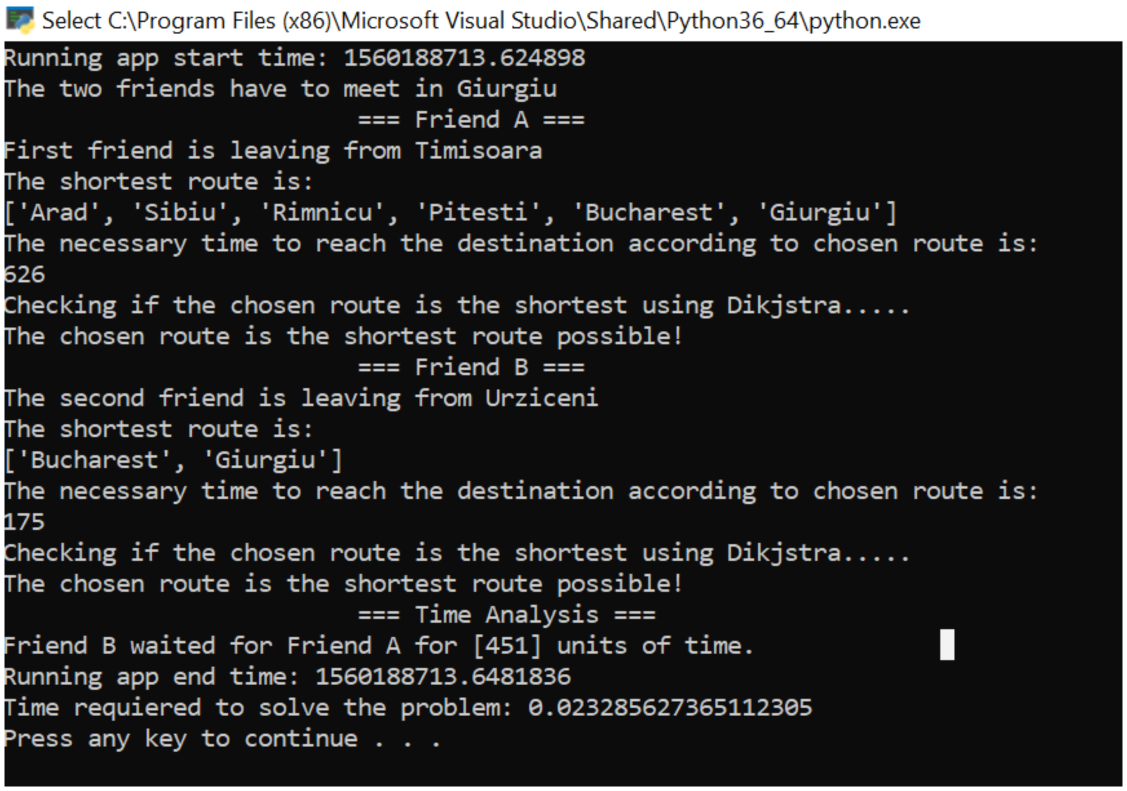
```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe
Running app start time: 1560187625.805484
The two friends have to meet in Hirsova
                         === Friend A ===
First friend is leaving from Sibiu
The shortest route is:
['Rimnicu', 'Pitesti', 'Bucharest', 'Urziceni', 'Hirsova']
The necessary time to reach the destination according to chosen route is:
461
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Friend B ===
The second friend is leaving from Zerind
The shortest route is:
['Arad', 'Sibiu', 'Rimnicu', 'Pitesti', 'Bucharest', 'Urziceni', 'Hirsova']
The necessary time to reach the destination according to chosen route is:
676
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Time Analysis ===
Friend A waited for Friend B for [215] units of time.
Running app end time: 1560187625.8957255
Time requiered to solve the problem: 0.0902414321899414
Press any key to continue . . .
```

•

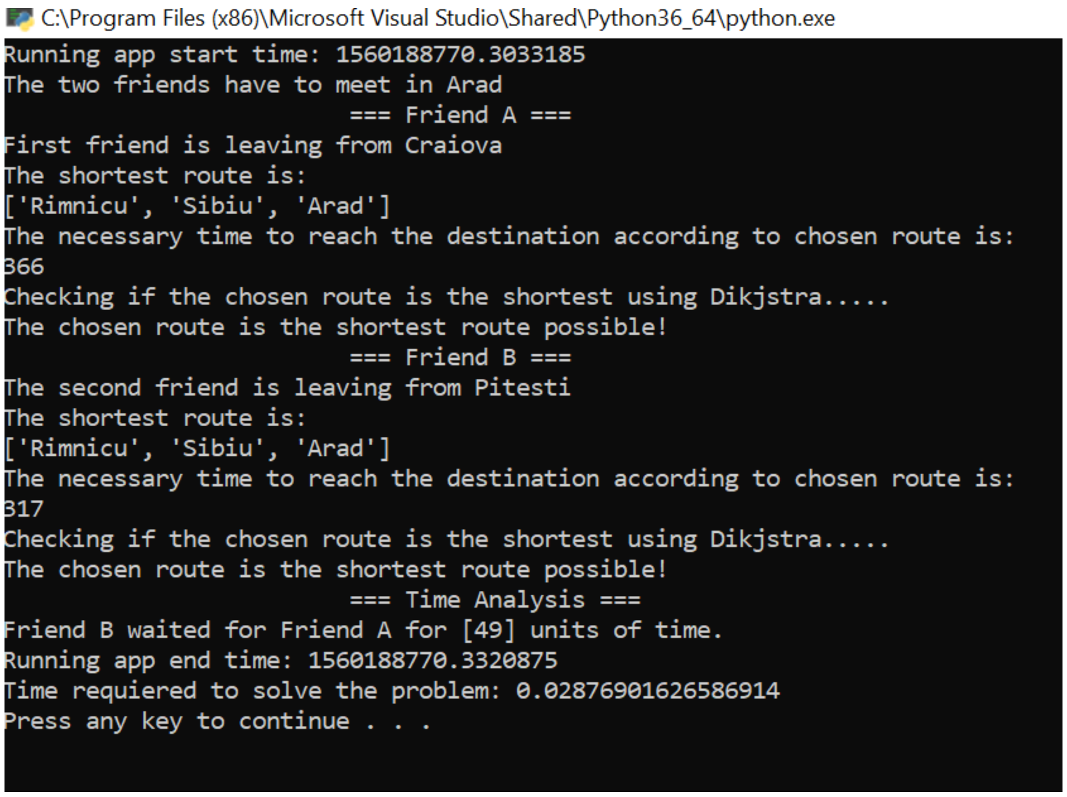


```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe
Running app start time: 1560187670.6806686
The two friends have to meet in Mehadia
                         === Friend A ===
First friend is leaving from Sibiu
The shortest route is:
['Rimnicu', 'Craiova', 'Drobeta', 'Mehadia']
The necessary time to reach the destination according to chosen route is:
421
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Friend B ===
The second friend is leaving from Timisoara
The shortest route is:
['Lugoj', 'Mehadia']
The necessary time to reach the destination according to chosen route is:
181
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Time Analysis ===
Friend B waited for Friend A for [240] units of time.
Running app end time: 1560187670.7169318
Time requiered to solve the problem: 0.036263227462768555
Press any key to continue . . .
```

•

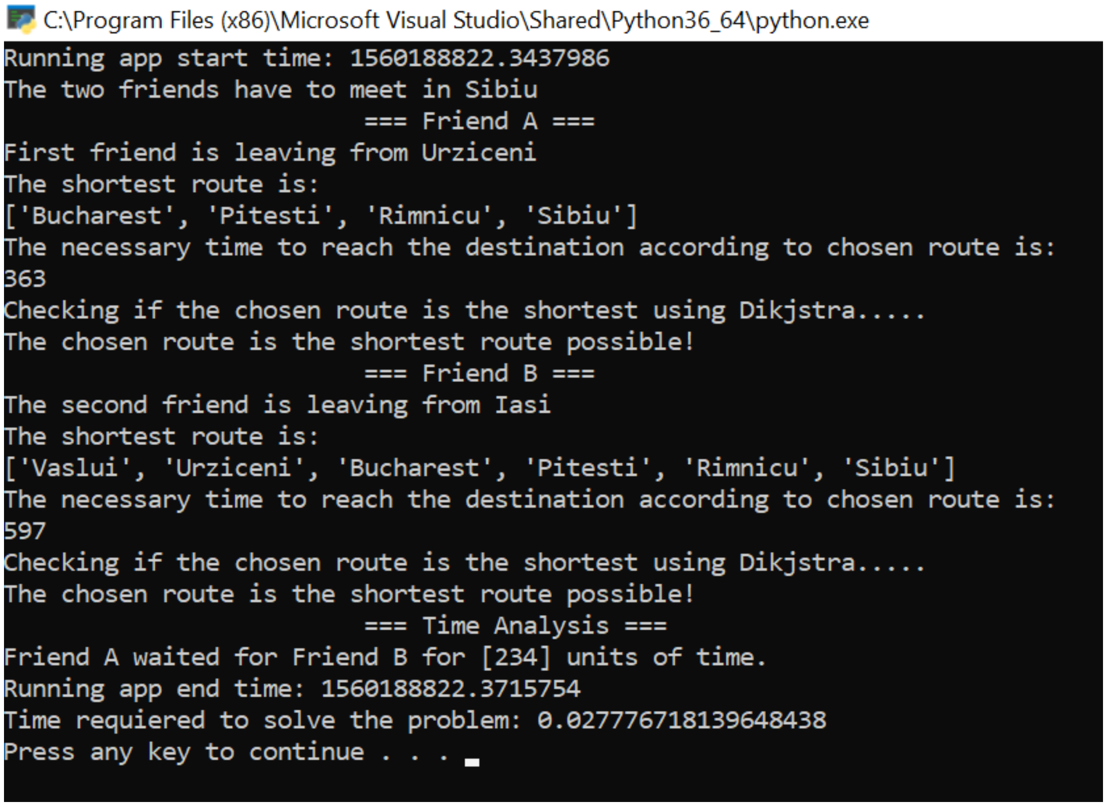


```
Select C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe
Running app start time: 1560188713.624898
The two friends have to meet in Giurgiu
                         === Friend A ===
First friend is leaving from Timisoara
The shortest route is:
['Arad', 'Sibiu', 'Rimnicu', 'Pitesti', 'Bucharest', 'Giurgiu']
The necessary time to reach the destination according to chosen route is:
626
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Friend B ===
The second friend is leaving from Urziceni
The shortest route is:
['Bucharest', 'Giurgiu']
The necessary time to reach the destination according to chosen route is:
175
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Time Analysis ===
Friend B waited for Friend A for [451] units of time.
Running app end time: 1560188713.6481836
Time requiered to solve the problem: 0.023285627365112305
Press any key to continue . . .
```



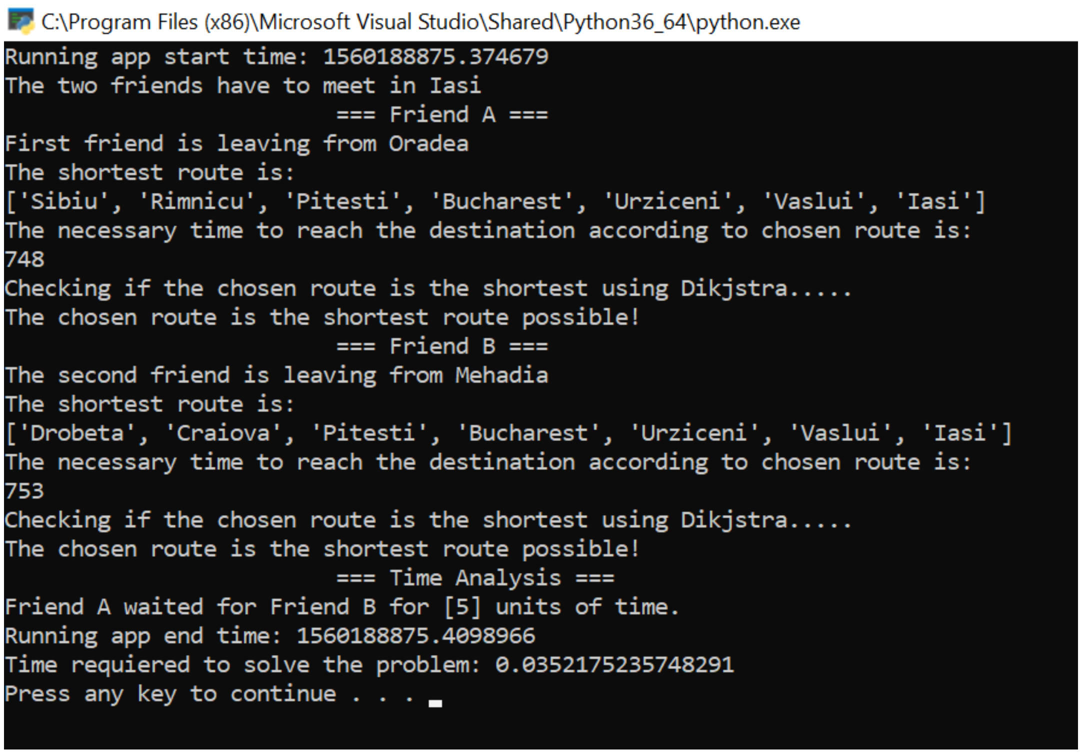
```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe
Running app start time: 1560188770.3033185
The two friends have to meet in Arad
                         === Friend A ===
First friend is leaving from Craiova
The shortest route is:
['Rimnicu', 'Sibiu', 'Arad']
The necessary time to reach the destination according to chosen route is:
366
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Friend B ===
The second friend is leaving from Pitesti
The shortest route is:
['Rimnicu', 'Sibiu', 'Arad']
The necessary time to reach the destination according to chosen route is:
317
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Time Analysis ===
Friend B waited for Friend A for [49] units of time.
Running app end time: 1560188770.3320875
Time requiered to solve the problem: 0.02876901626586914
Press any key to continue . . .
```

•



```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe
Running app start time: 1560188822.3437986
The two friends have to meet in Sibiu
                         === Friend A ===
First friend is leaving from Urziceni
The shortest route is:
['Bucharest', 'Pitesti', 'Rimnicu', 'Sibiu']
The necessary time to reach the destination according to chosen route is:
363
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Friend B ===
The second friend is leaving from Iasi
The shortest route is:
['Vaslui', 'Urziceni', 'Bucharest', 'Pitesti', 'Rimnicu', 'Sibiu']
The necessary time to reach the destination according to chosen route is:
597
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Time Analysis ===
Friend A waited for Friend B for [234] units of time.
Running app end time: 1560188822.3715754
Time requiered to solve the problem: 0.027776718139648438
Press any key to continue . . .
```

•

- 

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python36_64\python.exe
Running app start time: 1560188875.374679
The two friends have to meet in Iasi
                         === Friend A ===
First friend is leaving from Oradea
The shortest route is:
['Sibiu', 'Rimnicu', 'Pitesti', 'Bucharest', 'Urziceni', 'Vaslui', 'Iasi']
The necessary time to reach the destination according to chosen route is:
748
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Friend B ===
The second friend is leaving from Mehadia
The shortest route is:
['Drobeta', 'Craiova', 'Pitesti', 'Bucharest', 'Urziceni', 'Vaslui', 'Iasi']
The necessary time to reach the destination according to chosen route is:
753
Checking if the chosen route is the shortest using Dikjstra.....
The chosen route is the shortest route possible!
                         === Time Analysis ===
Friend A waited for Friend B for [5] units of time.
Running app end time: 1560188875.4098966
Time requiered to solve the problem: 0.0352175235748291
Press any key to continue . . .
```

-

Conclusions

The favourite part of this problem was the practical sense of the statement. Given a real map, with real costs and coordinates, I could solve using heuristic search strategy a practical and perhaps real problem.

Some improvements can be done regarding the generality of the solution. I used the map of Romania as constant data, but easily I can make this solution also available for any given map, or more precisely for any given cities and the distances between them. As a result, the 2 friends could plan meeting even if they are in different countries. They could be anywhere, as long as they would provide a similar map with cost, the solution described in this project would be available, but with little modifications.

One of my favourite achievements of this project was the organisation of the code, I wanted to create a readable application with clean code organisation. I liked that I could combine the OOP knowledge with the algorithms and strategies used in AI problems. Also, it was a challenge to choose the most friendly and practical way of output such that the result generated is understood as good as possible.