Team:

Andreea Matei

Florin Deleanu

Peking Express

Data structures & algorithms

Fontys

University of Applied sciences

Table of contents:

1. Overview ……………………………………………...2

1.1 Code Structure ………………………………….3

1.2 Strategy

1. Code Analysis

2.1 Time Complexity

2.2 Performance Test

1. Quality Assurance

3.1 Unit Tests

1. OVERVIEW

## This section provides an extremely useful insight into the overall organization of the code as well as introduces the strategy chosen for the algorithm behind.

## TOPICS:

#### 1.1 Code Structure

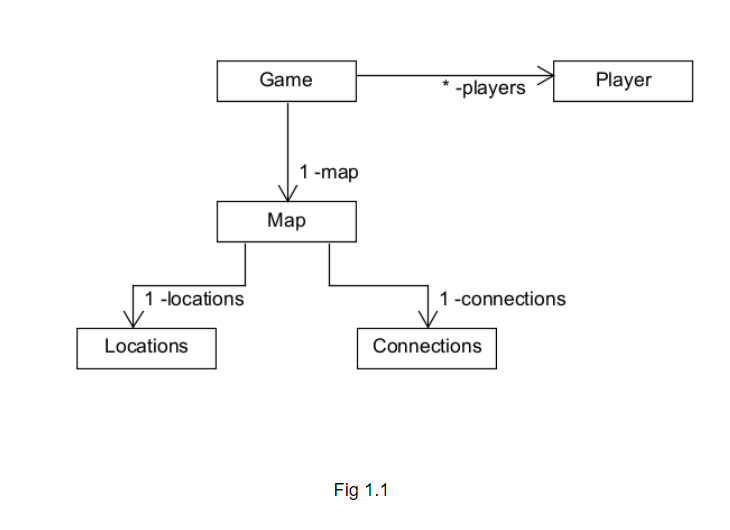
#### 1.2 Strategy

# 1.1 Code Structure:

The architecture of the whole code is based as much as possible on object-oriented principles, thus trying to reach the goal of a piece of software easy to integrate, maintain and especially to extend.

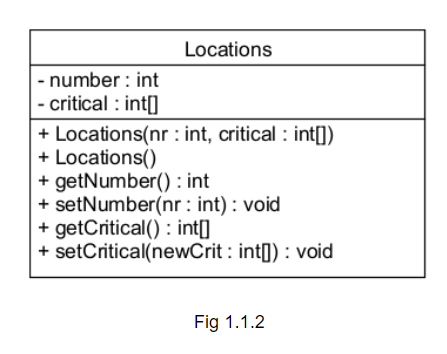
The logic behind the game is in the Game class, which manages the whole game with all the players, in other words, “pulling the strings of the show”. This class contains the “game map” as well as a list of all the players. In turn, the Map class contains other objects of the Locations and Connections classes, which will prove themselves to be of great help later when initializing the map for the game start.

A better overview of the whole structure can be seen below in Fig. 1.1.



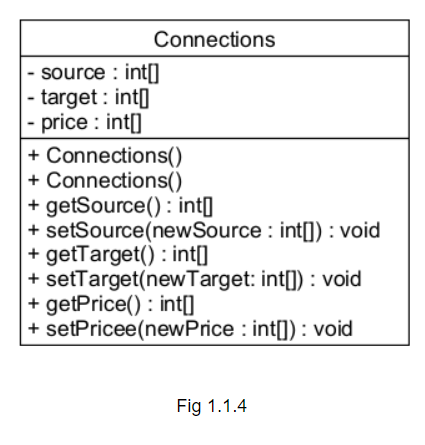
The Locations class has as attributes the total number of vertices (“locations”) of the map and a list of all the vertices that are considered to be “critical”. A critical point can only accommodate one player at a time, which means that the location, if occupied, is temporarily unavailable.

The Locations class can be better understood and visualised with the help of the figures(Fig 1.1.2 and Fig 1.1.3) from below:





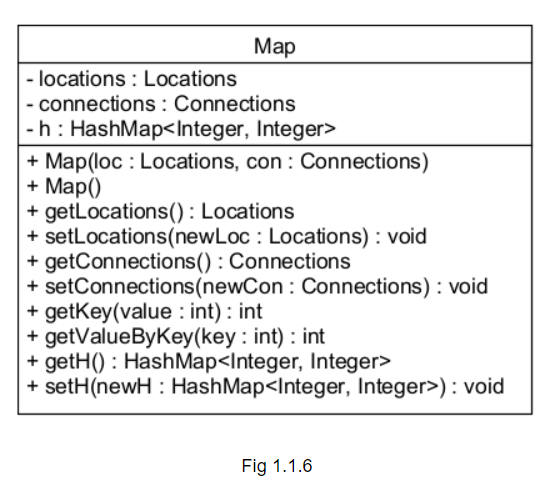
The Connections class includes 3 important lists/arrays: one of all the sources of all edges, second one of all the targets of all edges, and the last one of all the transport costs in the graph. These will be of huge help when creating the graph of the game map. This class is organized as shown below (Fig 1.1.4 and Fig 1.1.5):





The Map class, then, reunites 2 instances of the classes described above and, with the help of a HashMap, creates the first basis of the graph that will later represent our map. More explicitly, the class connects the number of the nodes with a line in the graph matrix.

An extremely good representation of this class is shown below (Fig 1.1.6 and FIg 1.1.7):





Below, the setup of the first basis for the graph matrix is shown (Fig 1.1.8):



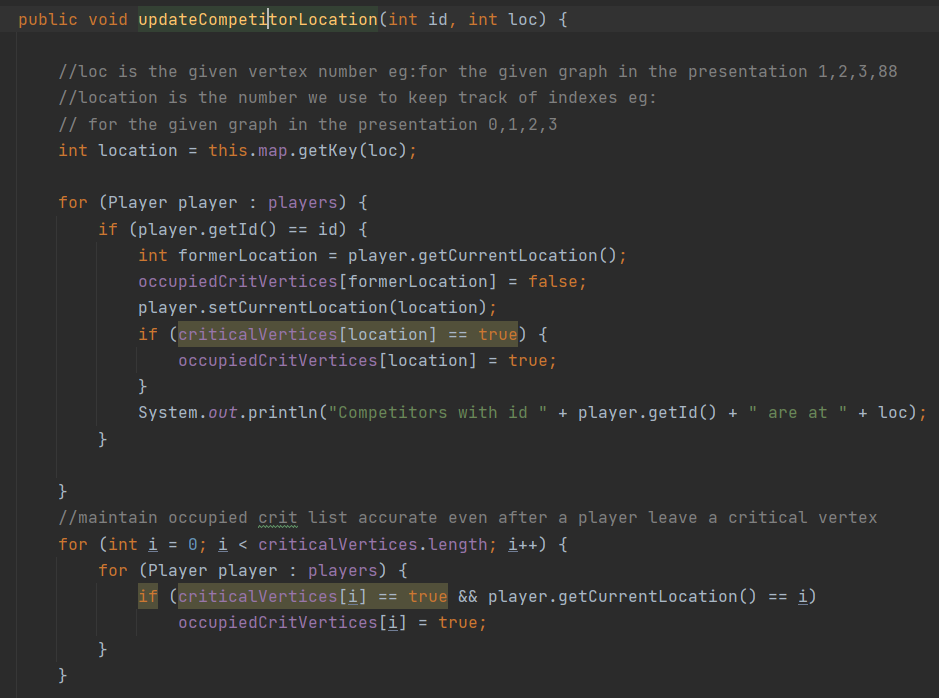
The constructor assigns every node an unique key, which represents a line in the graph matrix, matrix that will be used later to initialize the map.



The methods getKey() and getValueByKey() just access the HashMap, retrieving the wanted piece of information (Fig 1.1.9).

The Player class takes as attributes an id and a location. With the help of these 2 elements, we will be able to track down their moves, thus ensuring a better quality for our algorithm which determines the best possible nest move of our player.

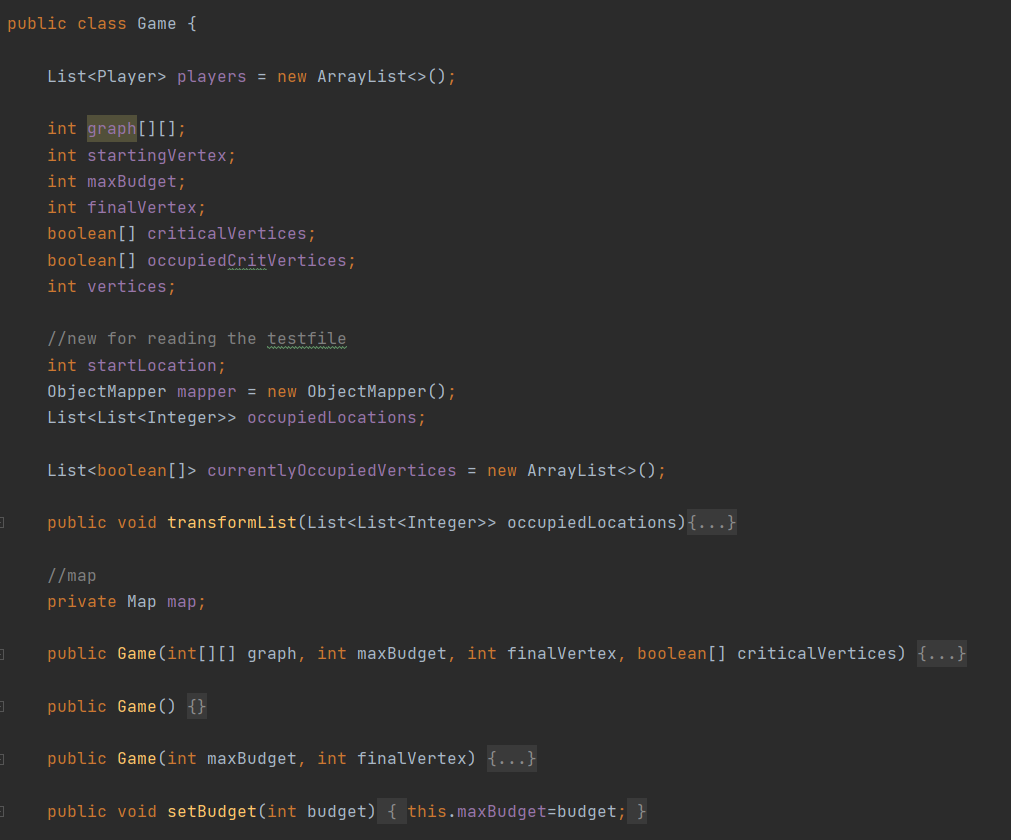
Before the requirements of the assignment were changed, we had already incorporated a method that updates the player locations both in our algorithm, but also in their own class

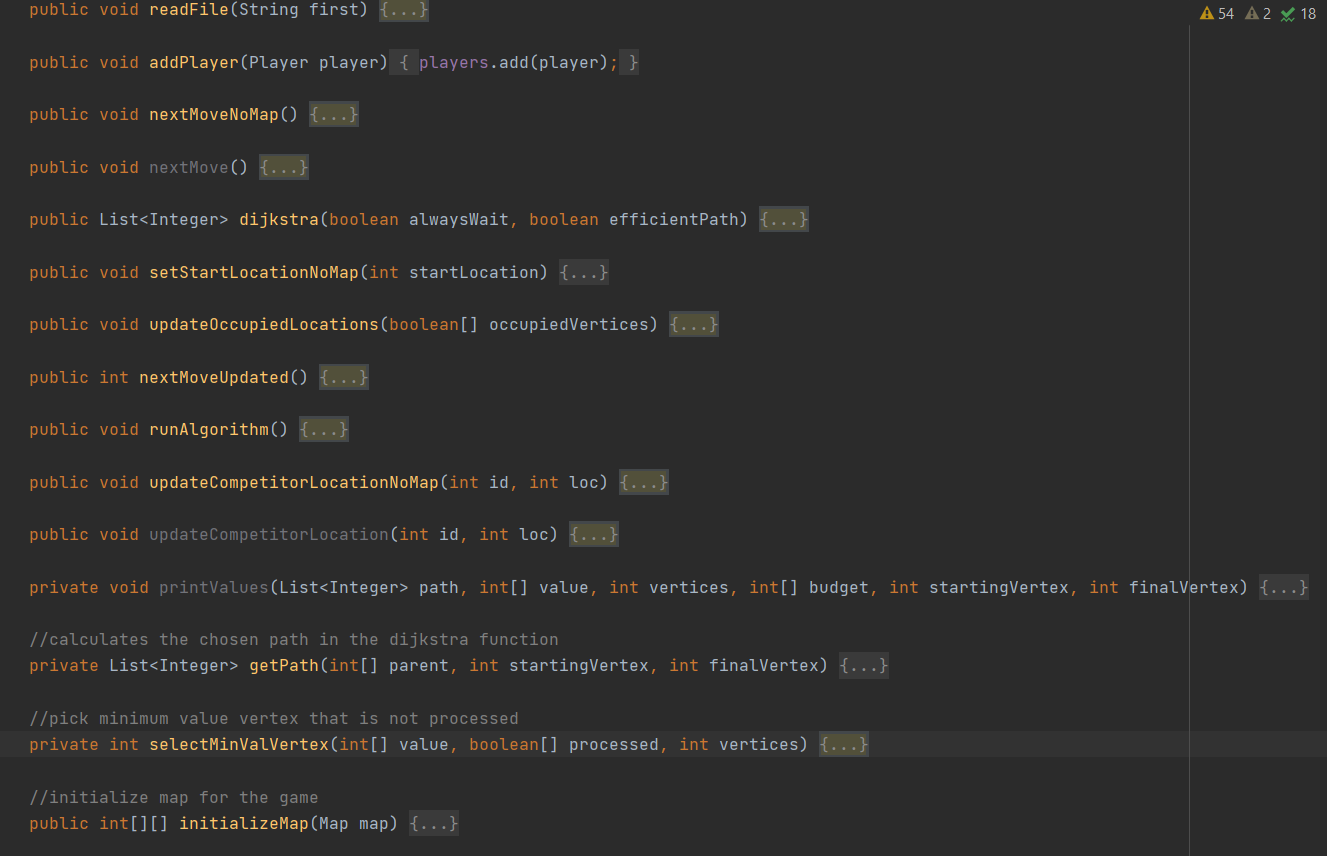


The Game class plays a vital role in connecting every component of the game itself, such as the map, the players. This class contains all the functions required for the construction of the game, including:

* **initializeMap** (map)
* **setStartLocation** (int startlocation)
* **updateCompetitorLocation** (int groupid, int location)
* **nextMove**()
* **readFile**(String path)
* **updateOccupiedLocations**()

This can be better seen below:





As the name suggests, this class contains all the brains behind the game itself. One method worth being discussed and presented here would be readFile(testfile). This is in charge of retrieving all the information required in order to start the game. The function has as parameter a String “testfile”, which represents the name of the file that should be read. By doing so, we make sure that we can change at any time the file that we want to read from. More details and explanation behind the algorithm will be discussed in the next section 1.2, Strategy.



1.2 Strategy

Throughout the strategy-setting process, an impressive number of ideas have been brainstormed. This has proved to be very useful in the current choice of a strategy for our algorithm. Of all the generated ideas, the ones worth mentioning are the following:

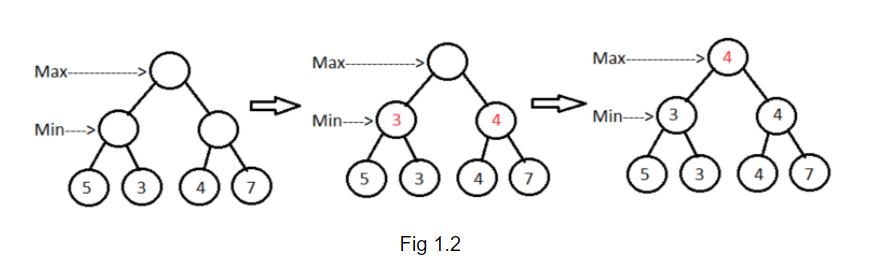
* Minimax algorithm
* All possible routes tree

A short description of the above strategies will be provided.

### Minimax algorithm

This algorithm is mainly encountered in game theory and is based on the backtracking algorithm. What makes this algorithm extremely interesting and fascinating is the fact that it is mostly used in decision making, more precisely, in finding the optimal move for a player, assuming that the opponent(s) also play(s) optimally. It is widely implemented in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

In Minimax, the two players are called **maximizer** and **minimizer**. The **maximizer** tries to get the **highest score** possible while the **minimizer** tries to do the opposite and get the **lowest score** possible. With this mindset, a tree can be created, taking into account all the players' choices as well as the states of the game. Such a tree would like like the following (Fig 1.2):



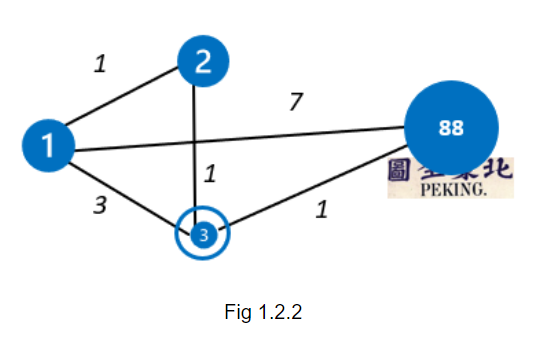
Every node represents a game state: the leaves represent the terminal states, and the root will always represent the current state of the game.

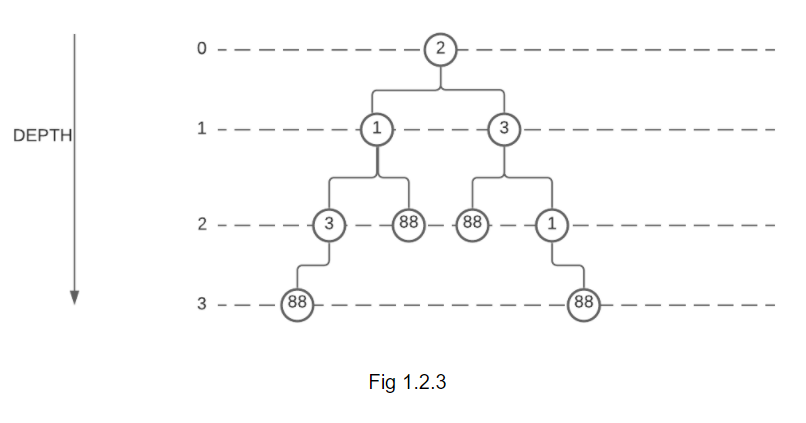
Every board state has a value associated with it, which is calculated by some evaluation function (some heuristics), which is unique for every game.

This algorithm could later be improved by applying ⍺-β pruning.

All possible routes tree

This approach is all about calculating in advance all possible paths from the starting point to the destination, and then adding them to a tree in order to get a better overview of the entire map. Every node of the tree will keep track of the price accumulated up until that node from the root and the depth of the tree indicates the length of the paths. For the graph in Fig 1.2.2, such a tree, with the starting point 2 and destination 88, would look like (Fig 1.2.3):

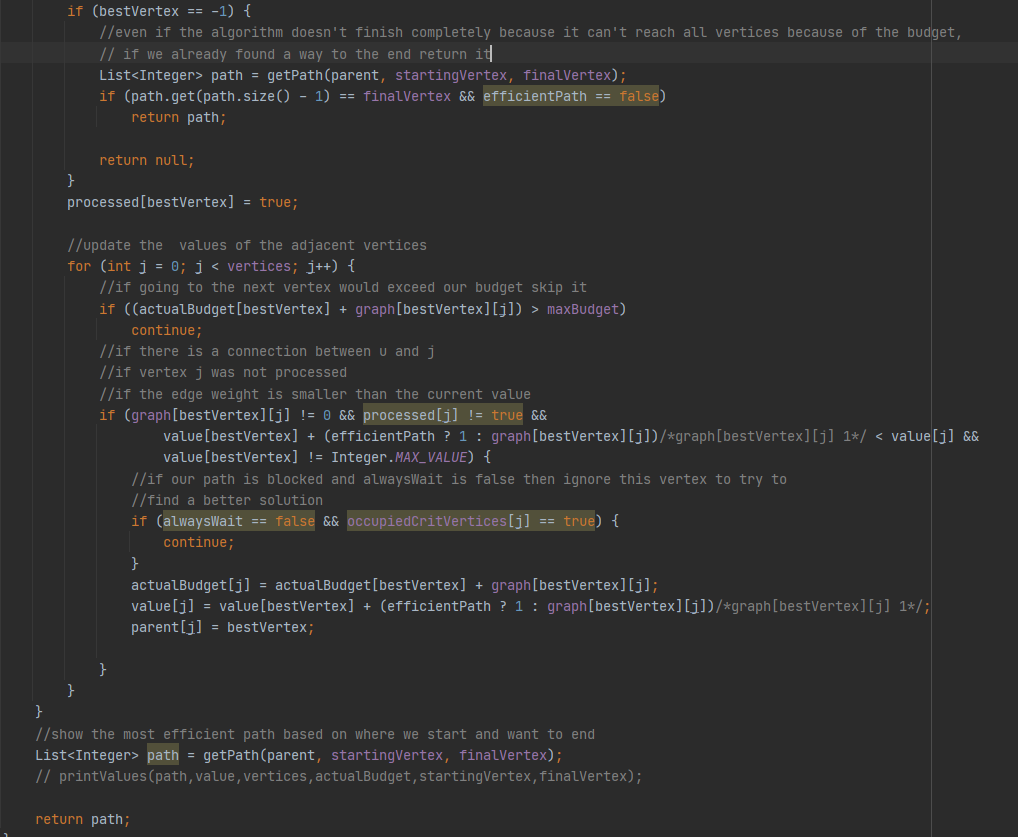


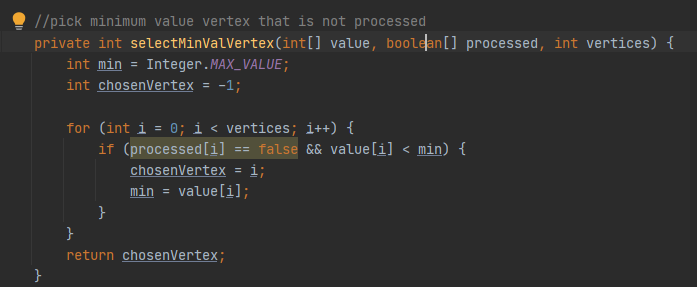


Our strategy

The strategy chosen for the game is inspired by Dijkstra's algorithm. Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V, E) for the case in which all edge weights are nonnegative. The edge weights represent transport costs, therefore, we assume that w(u, v) ≥ 0 for each edge (u, v) ∊ E.

Dijkstra’s algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex u ∊ V-S

with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.



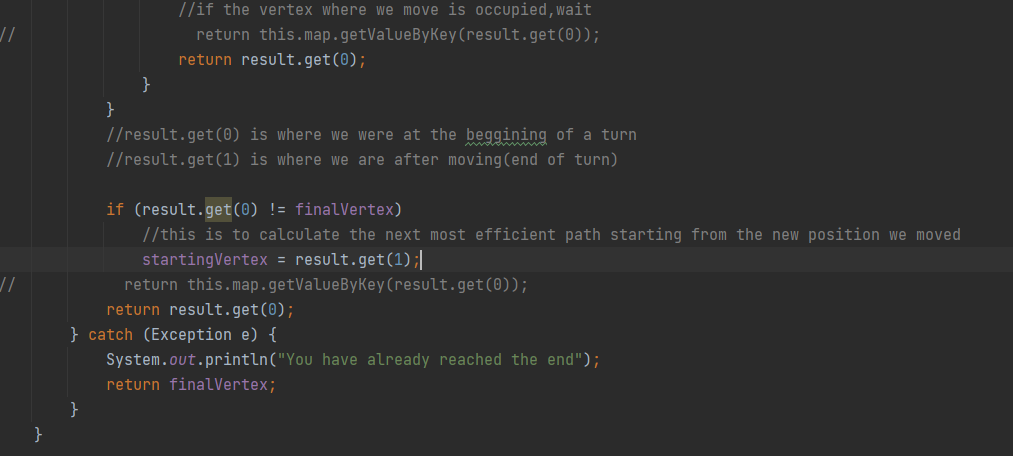
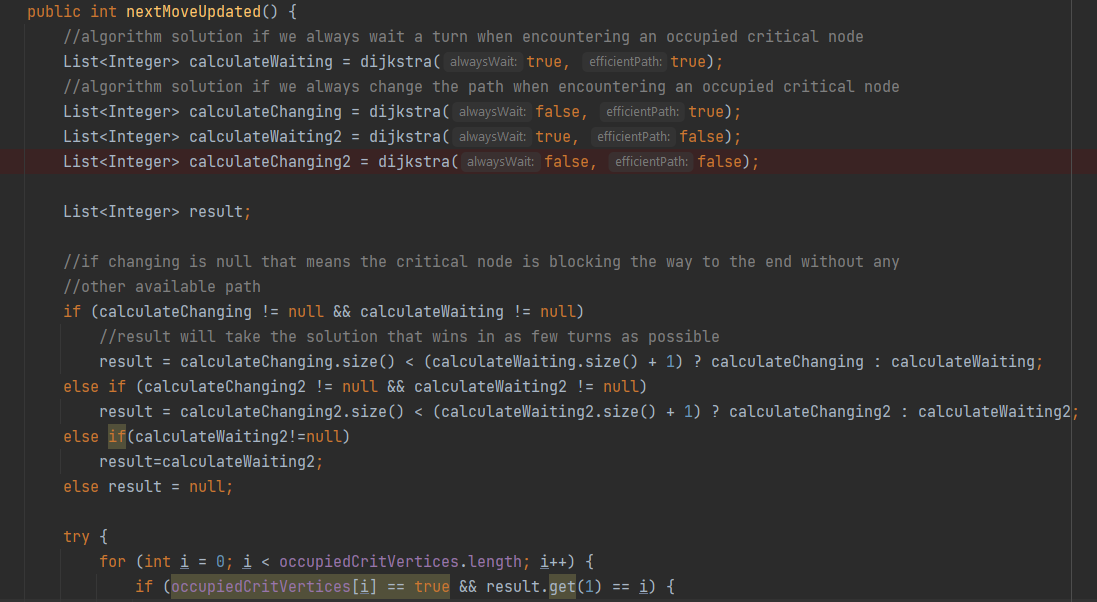
Although the beginning of our algorithm was Dijkstra, we modified it a lot to accommodate the requirements for this assignment. The direct changes were as follows:

1. Because we can start in any position and not always in the first one we had to make the algorithm choose where we want to start based on the variable startingVertex
2. We implemented another budget array similar to the value array but instead of the length it holds the budget and skips through the algorithm in case a path exceeds our budget
3. In our checking to see if a vertex is available we also used the occupiedCriticalVertices array similar to the budget one
4. We added a method that calculates the path we need to take in order to get from beginning to the end:



1. In the parameters of the methods we have 2 boolean variables:
   1. alwaysWait :
      1. if true the algorithm will wait at a critical vertex instead of choosing another path.
      2. if false the algorithm will treat that vertex as non-existent and will calculate another path
   2. efficientPath :
      1. if true the algorithm will try to find the shortest way even though it might get stuck by running out of budget
      2. if false the algorithm will always find a way to the end if the budget is enough for even 1 path, although it doesn’t guarantee it’s the shortest way

Because of these changes depending on the parameters the algorithm may not always find a way and will instead give the value for bestVertex as -1 (if there is no possible way with the given budget or the graph is disconnected). In case it gets stuck it will return a null variable or the way to finish(if it found any) to be handled in the nextMove() method.



This method is the “brains” of our path choosing.

Firstly we run dijkstra 4 times with all possible inputs. That will give us 4 different variables with possible different lengths that will all reach the end. Then we check to see which of these returns the lowest length. That one is our best choice so we choose it as the final result for this turn.

Now we have to check whether the result where is wait or move is the best one

1. If waiting is the correct choice return the result keeping everything the same
2. If changing is better first increase the starting vertex to our next move so we run the algorithm next turn from the new position, then return the position so we actually move

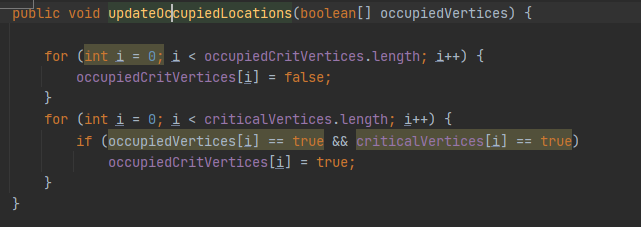
The way it is right now the algorithm will only reach the catch block when we are at the last position and we try to execute result.get(1) which is an array with only one element, thus giving us the final vertex and informing that we finished.

This is the method which automates the entire process of playing based on the new requirements we got in the edit



The try-catch block is only there for our algorithm to work even if we don’t provide a critical vertices file.

This final updateOccupiedLocations method is a lot simpler than the one we made originally because the requirements were updated to provide us with a list of arrays containing the players’ locations



This concludes our approach to this assignment’s problem.

# 

## 2. Code Analysis

## 

## 

## This section introduces the topics: asymptotic notation and performing test. The algorithm will be therefore analyzed based on its time complexity as well as on its performance.

## 

## 

## TOPICS:

#### 2.1 Time Complexity

2.2 Performance Test

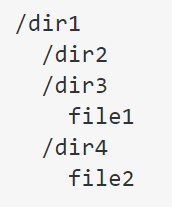
#### 

# 

2.1. Time Complexity

The time complexity of our code can be split into 2 main parts: reading the file and initializing everything we need to run the algorithm, and the actual algorithm part which chooses the path.

For reading the file, the average case time complexity should be Θ(logN), while the worst case would be O(N), where N = number of directories in the tree. Such a directory tree would look like the following:



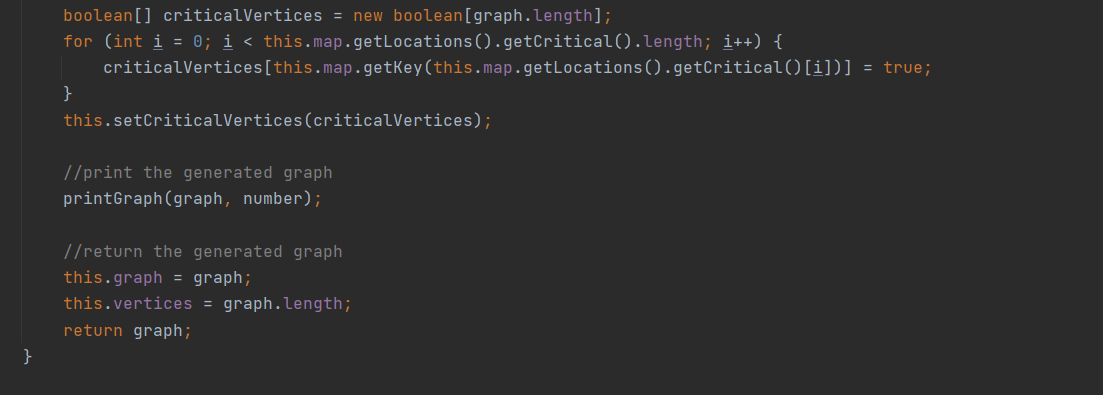
In order to retrieve the starting location as well as the budget from the testfile, we made use of the function parseInt() so we could convert the strings into natural numbers. Therefore, we will assume that it is linear by the number of characters representing the value being parsed.

The setters used in the function readFile(String testfileName) have a time complexity of O(1).



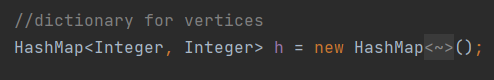
The method initializeMap(map) plays an important role, as it creates the game graph matrix based on the input from the file as well as prepares everything for the game itself. It has a time complexity of O(), where n = the number of vertices from the graph (or |V|).





The map object initialization from the readFile() method is worth mentioning, because in the class constructor there are more things to be taken into consideration.

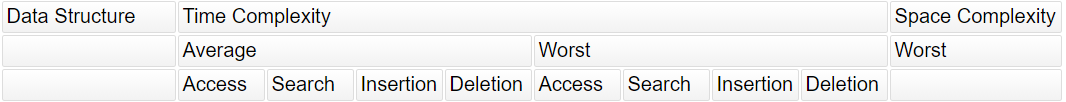
In order to keep track of the vertices and to simplify the process of putting them into the matrix correspondingly, a hashmap was used.



This data structure was used to facilitate the process of assigning a vertex to a line in the matrix. Therefore, the key represents the line number in the matrix for a certain element, and the value of the key represents that element.



A hashmap was chosen, because of its preferable time complexity:





The sorting function takes about O(n ・ ) average time complexity, but, in the worst case, it can reach up to O(). The while loop has O(n) time complexity, where n = total number of vertices in the graph. The last loop provides for the asymptotic notation a time complexity of O(m), where m = the length of the sources of the given edges.

The first part of the algorithm, “Reading + Initialization”, in other words: “SETUP”, takes about O() time complexity.

To analyze the algorithm part:

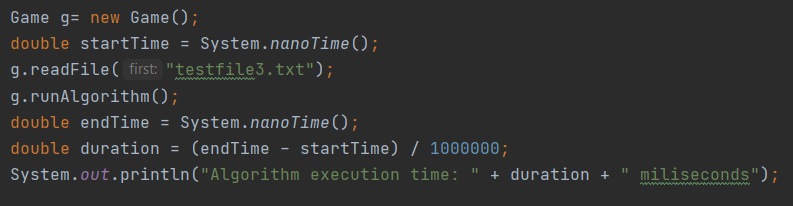
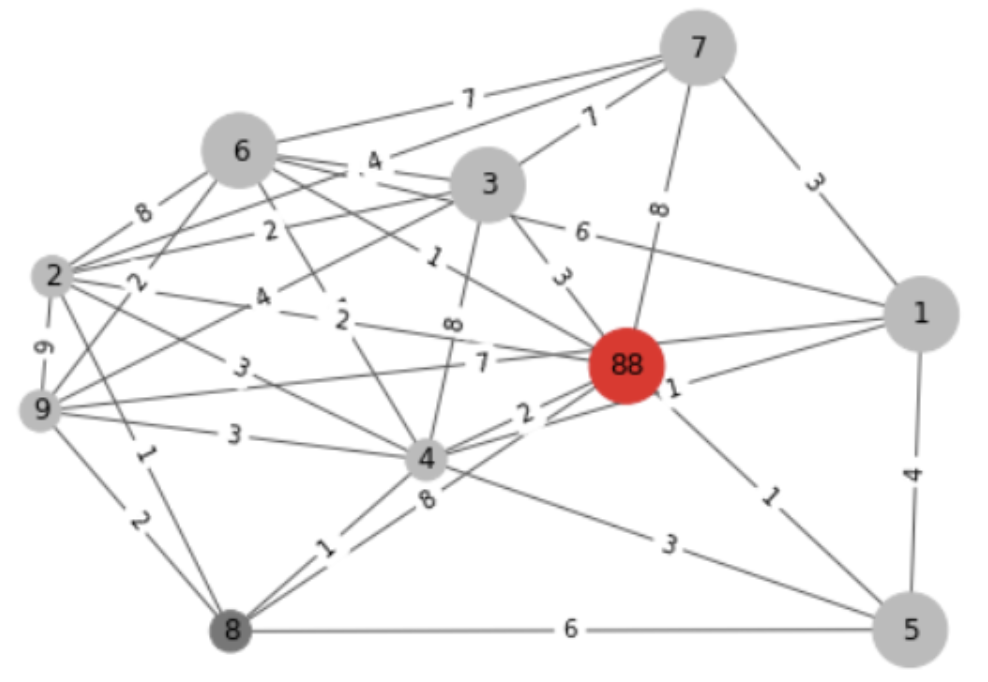
1. we start at runAlgorithm() which contains a while loop that will keep giving us new locations by executing the nextMoveUpdated() until we reach the end : O(n)
2. Inside the while loop we have nextMoveUpdated() which executes 4 dijkstra’s. ( 4\* O(n^2))
3. The only time difference between the usual dijkstra’s algorithm for the shortest path and our dijkstra is that our has an extra getPath() method that loops through the parent array to give us a path from start to finish.(O(n)) This getPath() method is called once in the outer for loop and once in the main body of dijkstra() which means it will not affect the time complexity. (O(n^2) per dijkstra)
   1. O(n) for the first for loop
      1. O(n) for getPath()
   2. O(n) for the second for loop
      1. O(n) nested for loop because of the selectMinValue function
      2. O(n) nested for loop below selectMinValue function.
   3. O(n) for last getPath()
   4. Total per dijkstra: O((n\*n)+n\*(n+n)+n) which simplifies to O(n^2) per dijkstra
4. Next in the nextMoveUpdated() method is a for loop through the critical vertices (O(n))

The entire calculation for the algorithm part is as follows: O(n\*(4\*(n^2)+n)) which can be simplified to O(n^3)

2.2. Performance Test

Given different input sizes, the average running time of the algorithm was the following:

1. For the graph given in the testfile number 3 from Sharepoint, the following execution time was found:



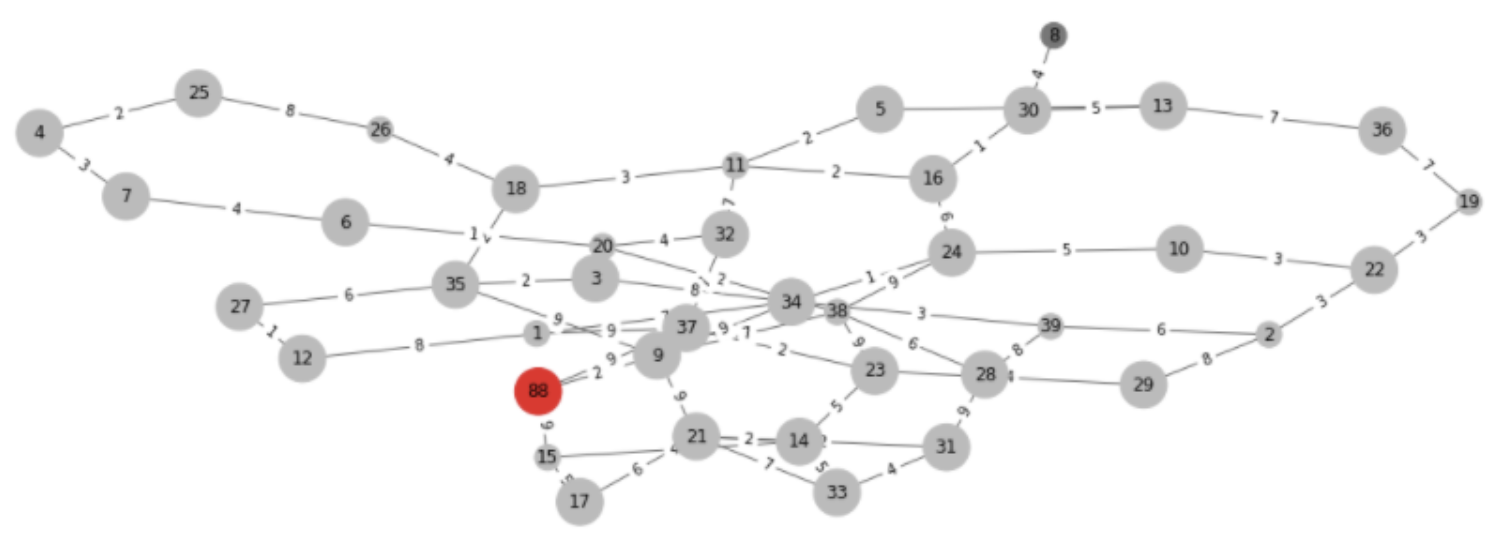
|  |  |  |  |
| --- | --- | --- | --- |
| First execution time | Second execution time | Third execution time | Average execution time |
| 117.1752≃117.2 ms | 82.5849≃82.6 ms | 91.4341≃91.4 ms | 97.066≃97.1 ms |

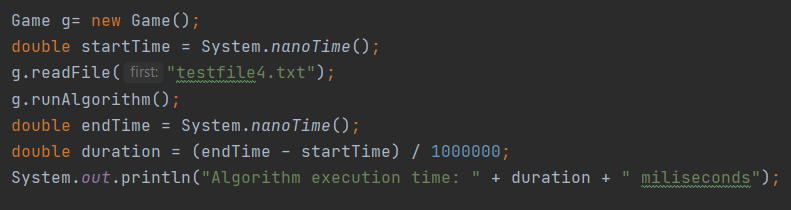
 (First execution time)

(Second execution time)

(Third execution time)

1. For the graph given in the testfile number 4 from Sharepoint, the following average execution time was found:





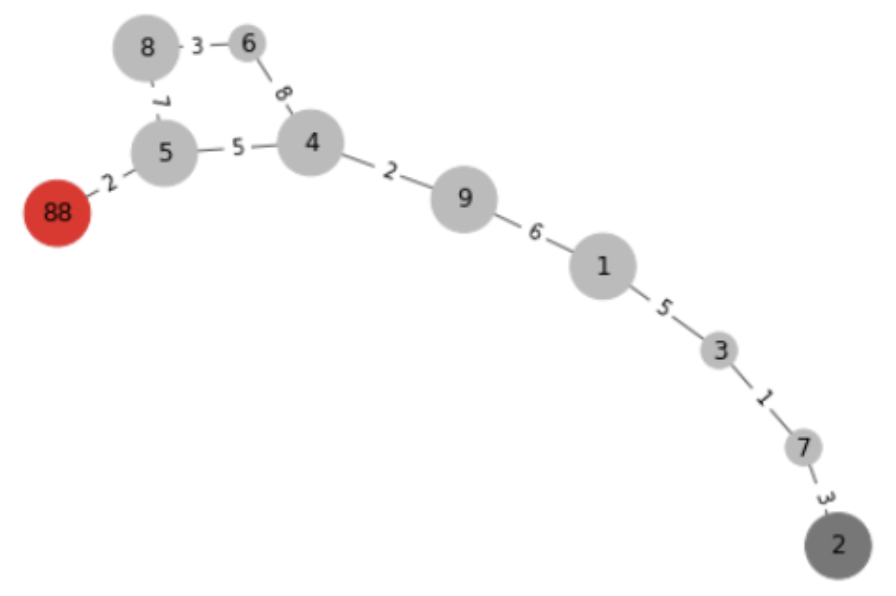
|  |  |  |  |
| --- | --- | --- | --- |
| First execution time | Second execution time | Third execution time | Average execution time |
| 118.6287≃118.6 ms | 162.1285≃162.1 ms | 135.306≃135.3 ms | 138.666≃138.7 ms |

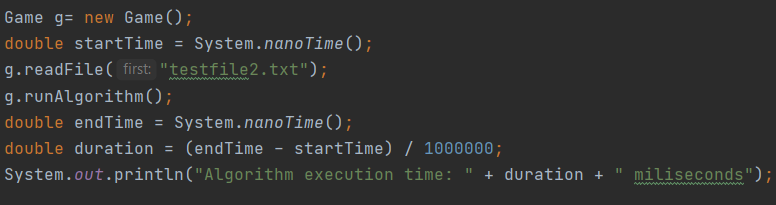
(First execution time)

(Second execution time)

(Third execution time)

1. For the graph given in the testfile number 2 in Sharepoint the following average execution time was found:





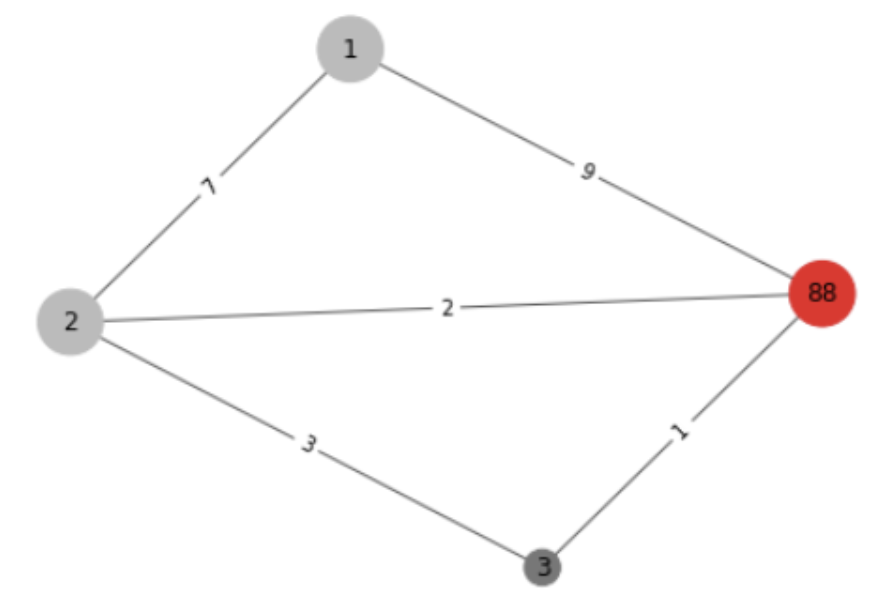
|  |  |  |  |
| --- | --- | --- | --- |
| First execution time | Second execution time | Third execution time | Average execution time |
| 84.913≃84.9 ms | 84.4174≃84.4 ms | 85.6192≃85.6 ms | 84.966≃85.0 ms |

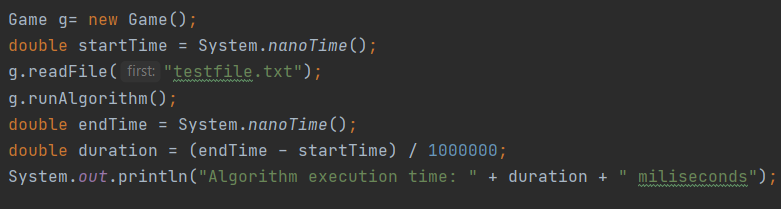
(First execution time)

(Second execution time)

(Third execution time)

1. For the graph given in the testfile number 1 from Sharepoint, the following execution time was found:





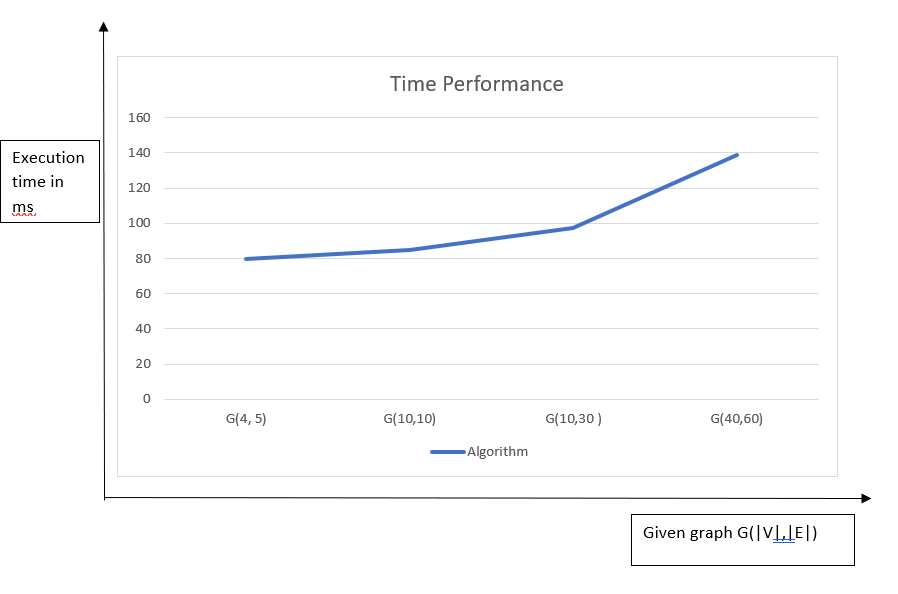
|  |  |  |  |
| --- | --- | --- | --- |
| First execution time | Second execution time | Third execution time | Average execution time |
| 79.7819≃79.8 ms | 79.7099≃79.7 ms | 79.8463≃79.8 ms | 79.766≃79.8 ms |

(First execution time)

(Second execution time)

(Third execution time)

Based on the above results, the Time Performance graph (timeplot) could be drawn:



## 

## 

## 3. Quality Assurance

## 

## 

## This section focuses on proving the correctness of the algorithm through several tests.

## 

## 

## TOPICS:

#### 3.1 Unit Tests

#### 

# 

# 

# 

# 

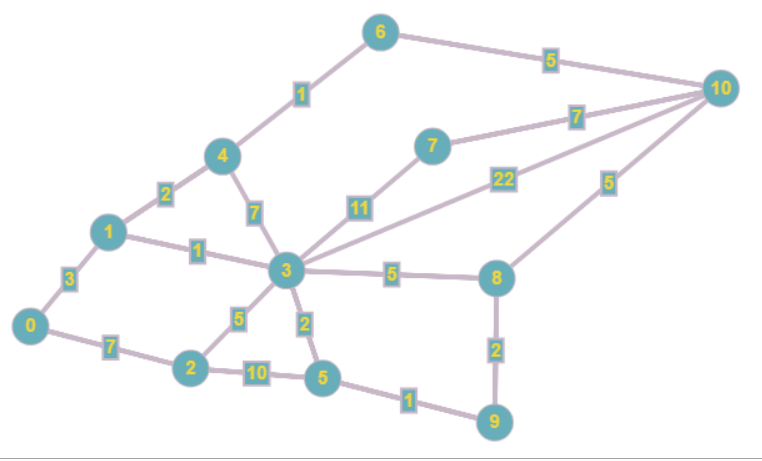
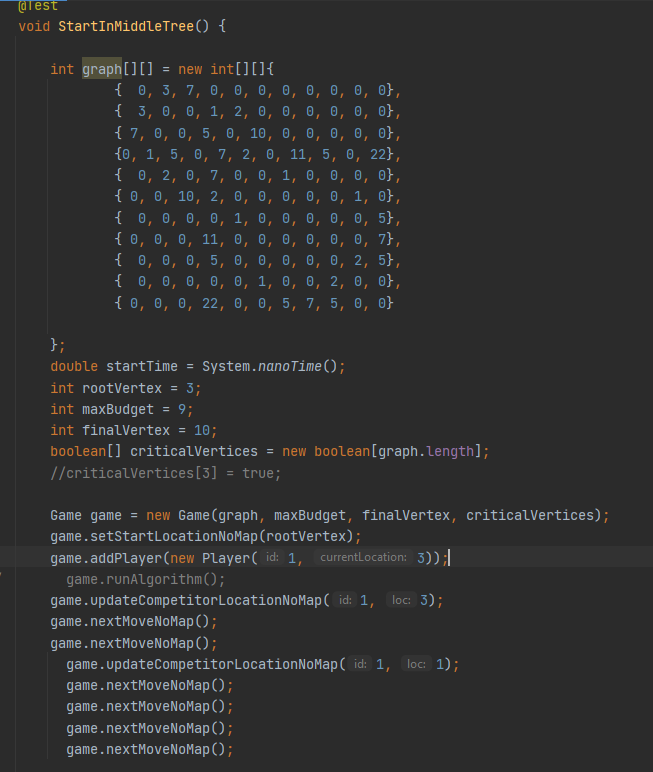
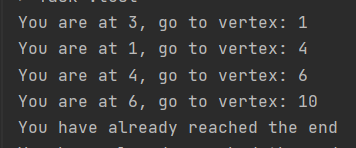
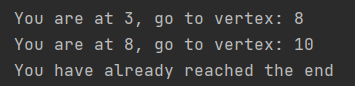
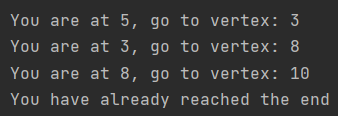
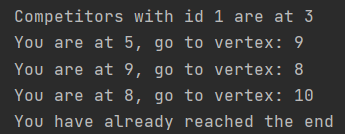
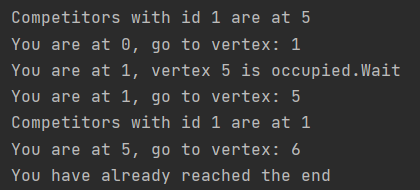
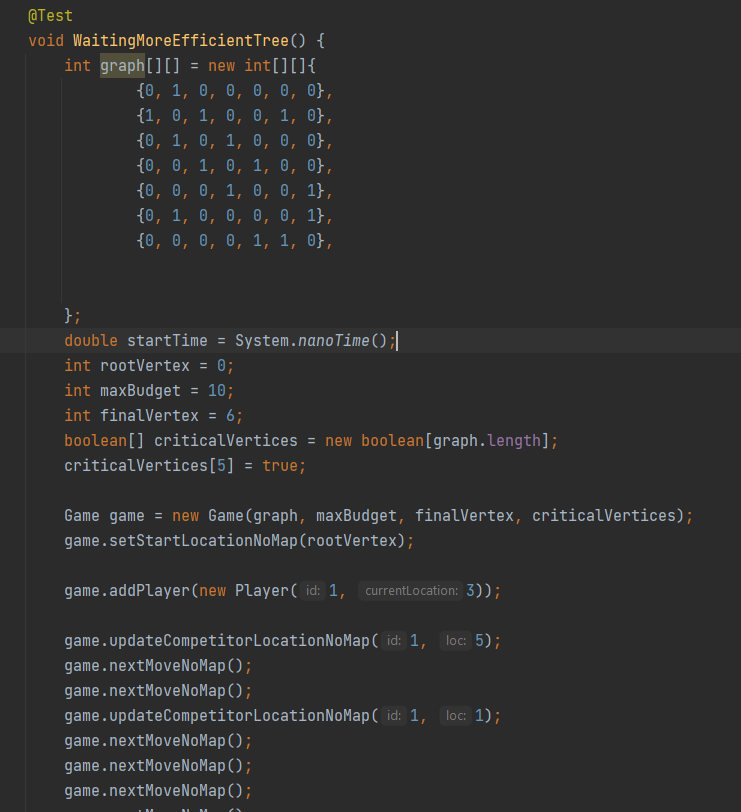
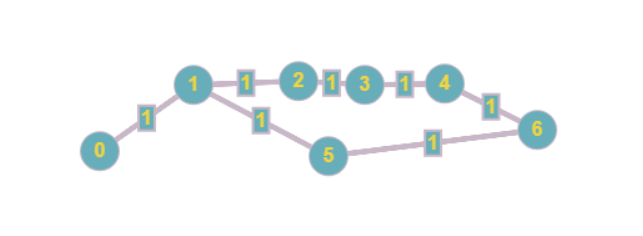
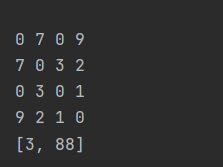
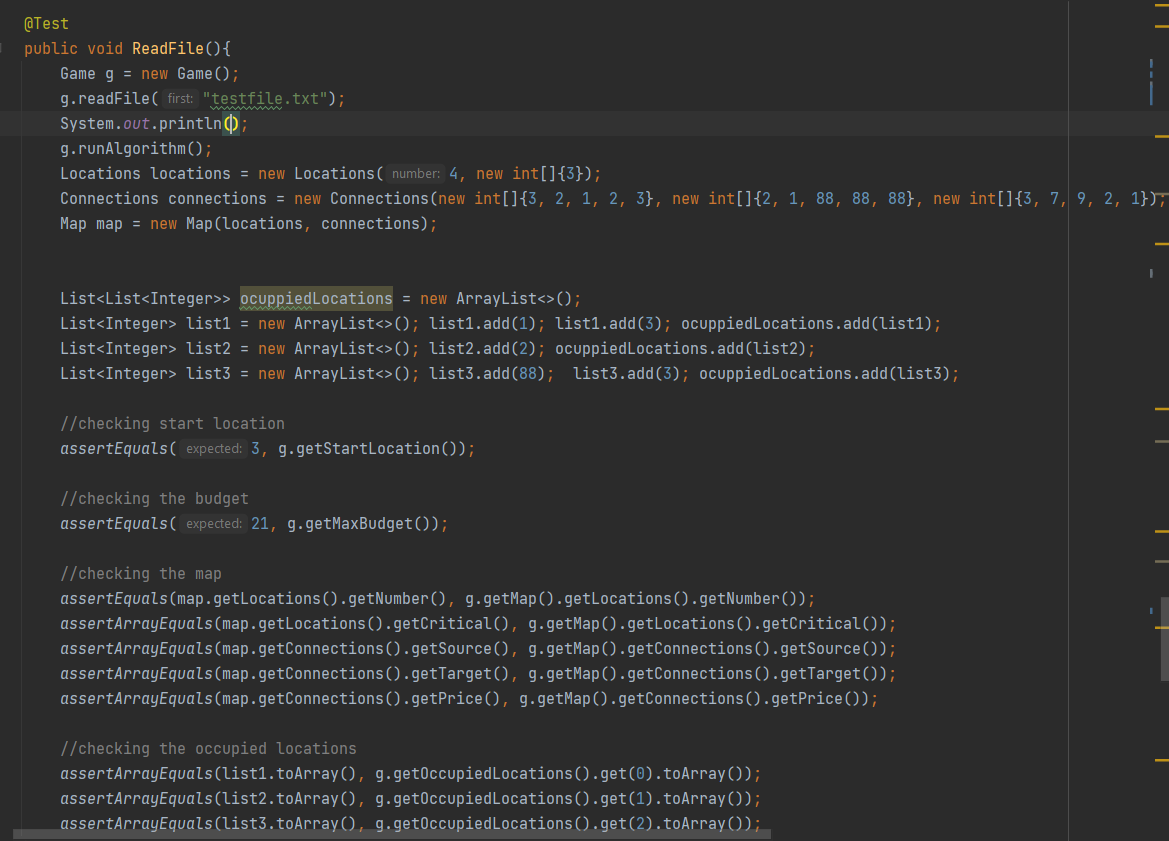
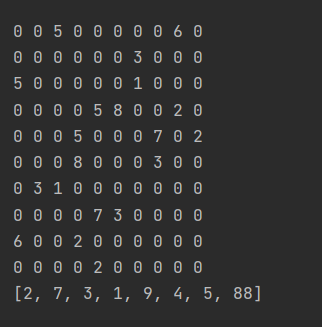
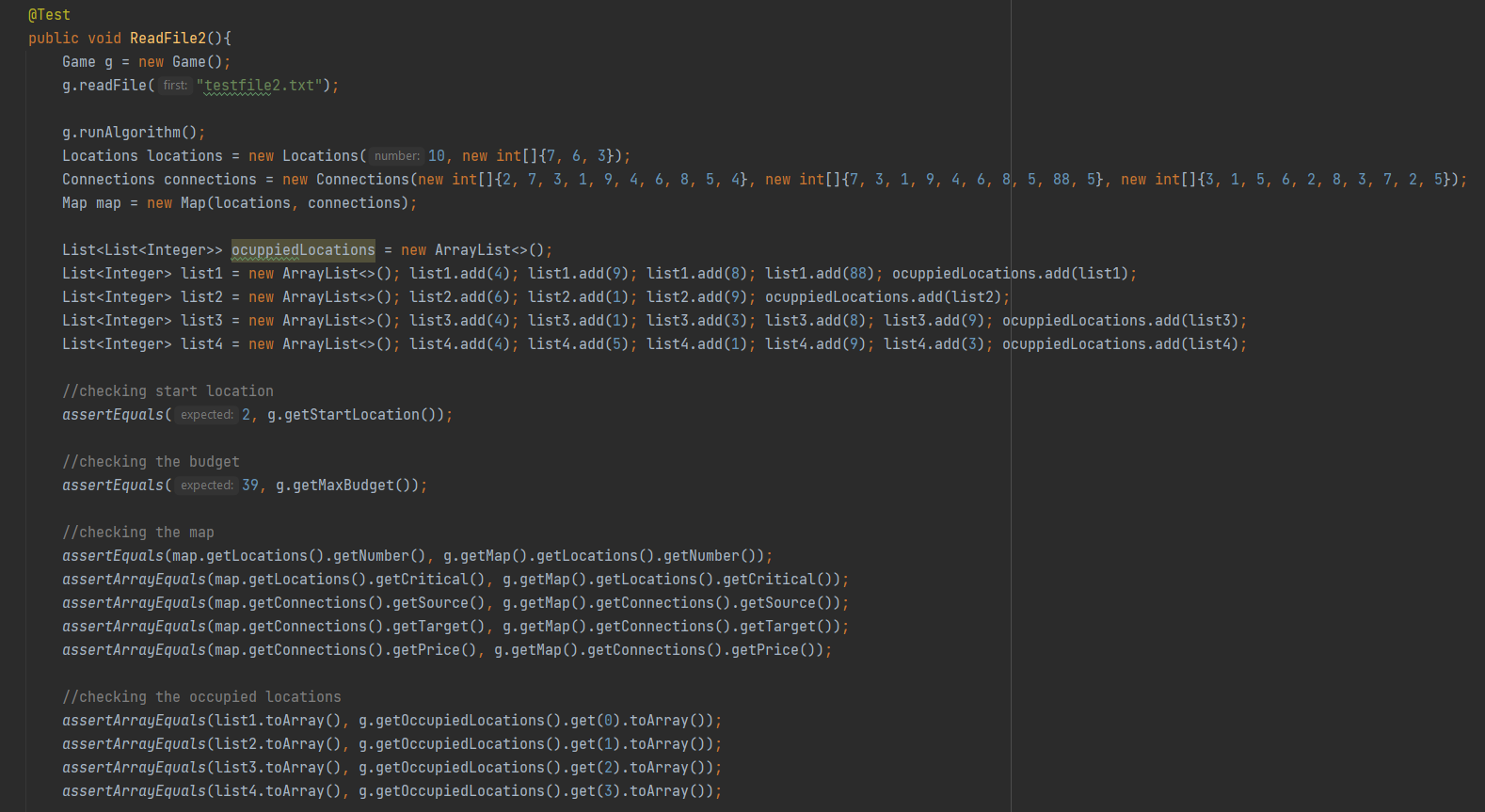
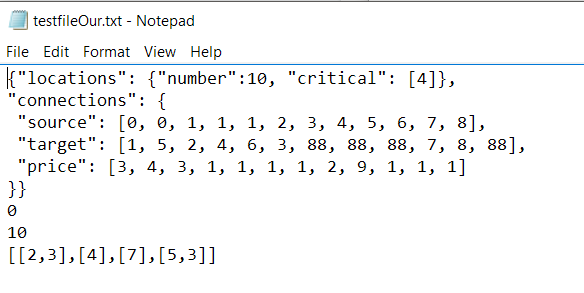
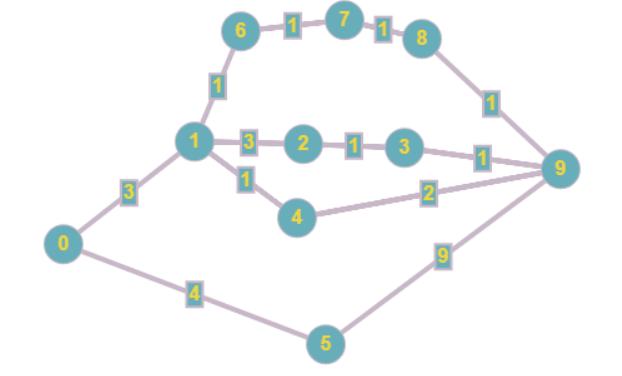
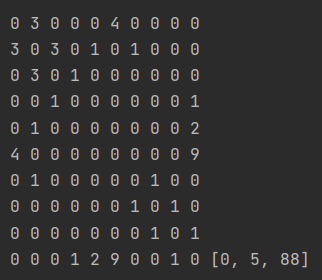
# 

# 

# 

# 3.1. Unit Tests

For our unit tests we will use the test files provided by the teachers as well as our own to better test specific cases. Some of the unit tests might have more simple graphs intended to test only one specific feature. Other unit tests may have a different code structure from the normal one to show the steps needed turn by turn more explicitly.

1. This tree provides the possibility to test the shortest route if the budget allows, in case the budget is tight show a route within the budget even if it is not the shortest. It also allows the possibility to change the path in case of encountering a critical vertex. Here 10 is the ending and the unit test is run step by step for explanation purposes, while changing the starting location
   1. root=3, maxBudget=9 the algorithm goes as follows: 3-1-4-6-10
   2. root=3,maxBudget=10 route 3-8-10 
   3. root=3, maxBudget=22 route 3-10
   4. root=5, maxBudget=12 route 5-3-8-10, could also be 5-9-8-10
   5. root=5 maxBudget=12 but 3 is an occupied critical vertex all the time, algorithm reroutes to go to 5-9-8-10 instead
2. This tree is used to simulate a situation where waiting is better than changing to another route
3. This test uses the first test file provided and tests if out map that is used to convert the data from the file to data used by the algorithm is correct, as well as showing the results of the algorithm(starting location 3 with budget 21)
4. This test is similar to the previous file but used the information the 2nd provided test with startingLocation 2 and budget 39
5. This test shows the results from the 3rd provided file starting in 8 with budget 105
6. This test shows the results from the 4th provided test file containing the huge graph starting in 8 with budget 110
7. The last unit test is made with a file containing data chosen by us for testing purposes(vertex 9 is the ending(treated as 88), and 0 the beginning)
   1. with budget 25 route is 0-5-88 (adjacency matrix also shown)
   2. with budget 10 and critical vertex 4 occupied when we need to move there the route is 0-1-1-4-88

All in all the above mentioned unit tests take care of all possible situations that can be encountered and always give us a n appropriate result.