27/12/2020

ADSA Mini Problem Report

Among us

Lies HAOUAS and Mounir HAMMOUTI
DIA3 – ESILV – 2020/2021

Presentation

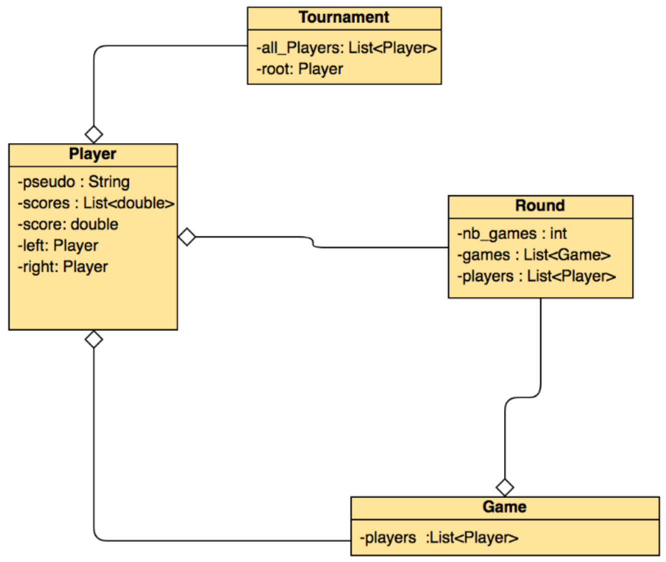
In this report, we will present how we thought about and responded to the issues raised by the topic Among Us. We chose to work in C# because it is the language that best suited our abilities.

The code is attached on the devinci platform, it is a .zip file containing the Among Us folder. This folder contains the "AmongUs.sln" file that you have to launch with your IDE (Visual Studio or other). The console interface will guide you through each step of the project.

Have a good reading!

Step 1 : To organize the tournament

Class diagram :



1) The **Player** class is used to represent a player and his score. The Player class contains :

- a nickname (the number of the player between 1 and 100)
- a score list that represents the history of scores in each game the player has participated in. The list is a good way to contain its values because we can't know beforehand the number of games a player will participate in, so the list with its dynamic size is a good choice
- a score that contains the average score of each game

2) The **Tournament** class is used to represent the tournament, the class contains :

- the list of all the tournament participants
- an AVL Tree of players

We have separated the tournament into different **rounds** representing each step of the tournament. The « round » class contains :

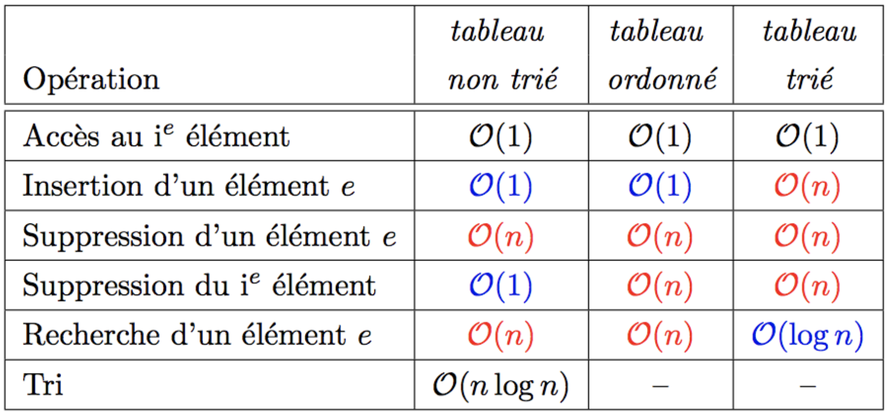
- the number of games that take place during this round (for the first 3 rounds, it is 10)
- the list of the players participating in this round
- the list of the games that take place during this round

The « **game** » class contains :

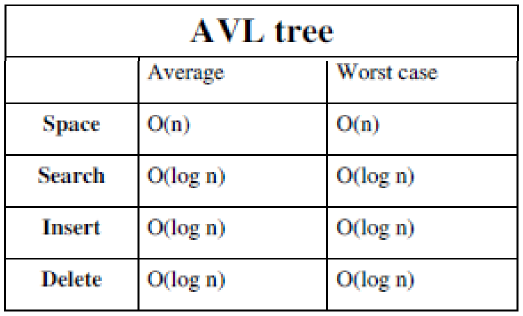
- the list of players associated with this game, a game can only contain 10 players

For the first 3 rounds (random game), we use the list of all participants to randomly assign players to games because the ranking does not count for the first 3 rounds.

When we join the 4th round, we build an AVL Tree of the players depending on their score. The AVL Tree gives us a complexity of O(log(n)) for search, delete, and insert operations. If we chosed a list structure, we would have have less performance to find a player, as we can see below. So the structure that we have chosen is an AVL Tree.



Opération	*tableau non trié*	*tableau ordonné*	*tableau trié*
Accès au i^e élément	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Insertion d'un élément e	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Suppression d'un élément e	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Suppression du i^e élément	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Recherche d'un élément e	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
Tri	$\mathcal{O}(n \log n)$	–	–



AVL tree		
	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

3) To randomly generate the score of a player at each game, we used the *random.Next(int maxValue)* method , which returns a positive integer strictly inferior to the value passed in argument maxValue. For the maxValue, we assigned the value of 13, because the score of a player is included in [0,12].

4) For the method that updates the score of the players, we have created the method New_Score(), in the "Game" class. This method adds to each player of the game the score they get in the current game to the list of scores they already had before. Then the average score is calculated to obtain the new rank.

5) To create the random games, we initialize a Round object, and we give as parameters: the number of games that must take place and the list of players not eliminated yet. For example, the 3 first round must have 10 random games and the 100 participant. Then we call the Start_Random() method from the "Round" class to generate the random games.

To explain how the method works, we create a copy of the participants list in the round, and we randomly select players from this list and assign them to a game. As soon as we assign a player to a game, we delete him from the copied list, to make sure we don't assign the same player to several games. To choose the players, we use a method that generates randomly a number between 0 and the size of the copied list, this number corresponds to the index of the player we're choosing.

6) To generate games based on ranking, we go through the tree (in-order traversal) to generate the sorted by score list of players. We create the games from this list by slicing the list in pieces of 10 players, until there is no more player remaining in the list.

7) To generate the ranked games, we generate a list of participants sorted from our tree by removing the last 10 player, in order to do it we use the **method Remove_Last10()** of the AVLTreePlayer class. This method goes through the tree, searches and removes the player with the minimum score, then rebalances the tree. We execute these steps 10 times then we go through the tree (in-order traversal) to generate the sorted list of players. We initialize a round object by passing the new list that contains the remaining players and we create the games from this list.

8) To display the TOP 10, we run through the whole tree using in-order traversal, and we add each node encountered in a list, this list is sorted by the average score of each player. The last 10 players of the list corresponds to the top 10.

To display the podium we use the **Max_Score_Player()** method of the AVLTreePlayer class which returns the player with the maximum score which is in the tree far right. We drop this player and add it to a list, we do it for two more players to get the 3 best players and make the podium.

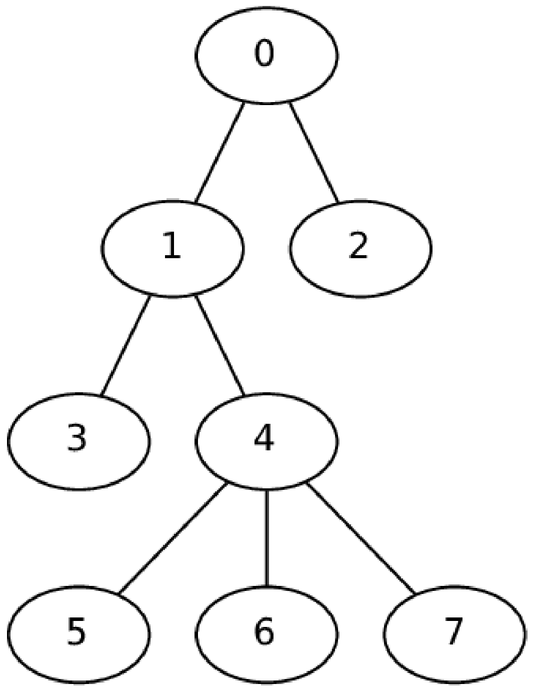
Step 2 : Professor Layton < Guybrush Threepwood < You

1) To represent the model in a graph, we use an adjacency matrix of n*n size, where n represents the number of players and the number of vertices there will be in our graph. If there is a 1 in row i and in column j, it means that player i has seen player j and this relation is represented by a link between vertex i and vertex j in the graph. If there is a 0, it means that player i has not seen player j and there will be no link between vertex i and vertex j.

<u>Example of a graph of 8 edges :</u>

There are 8 edges, so 8 rows and 8 columns, n = 8

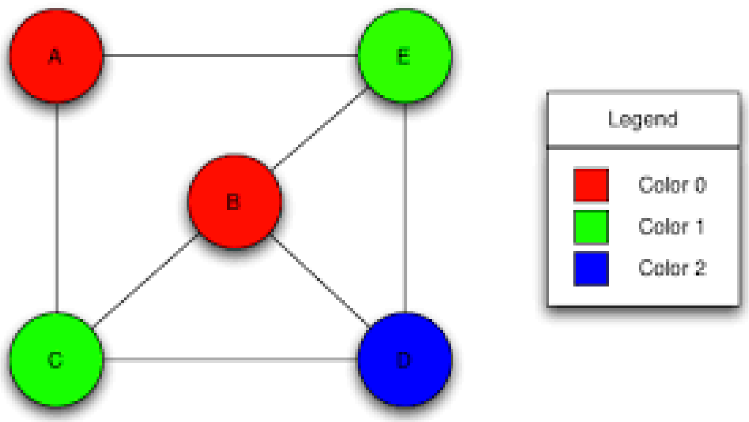
	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	1	0	0	1	1	0	0	0
2	1	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0
4	0	1	0	0	0	1	1	1
5	0	0	0	0	1	0	0	0
6	0	0	0	0	1	0	0	0
7	0	0	0	0	1	0	0	0



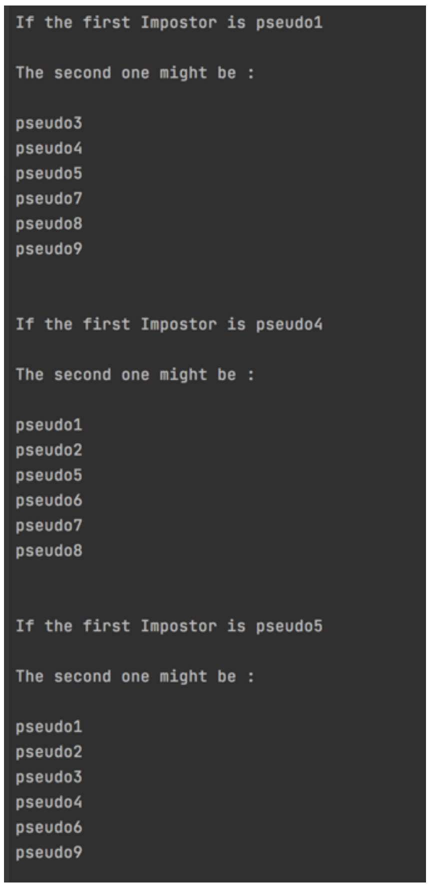
2) 3) The graph theory that we used for this problem is the coloring graph, in a coloring graph two adjacent vertices cannot have the same color. If we know that the two impostors do not walk together, because they did not meet and they have no link in the graph, we can conclude that they will have different colors. Our approach was to color the graph using the coloring graph algorithm, and then to investigate on the 3 probable impostors (the 3 players seen by the victim). We look for each impostor the players who have a different color from them and by removing the

victim and the imposter himself, we obtain the list of suspects for this imposter, we repeat this operation for each probable impostor. We thus obtain the set of suspects for each impostor.

Example of coloring graph:



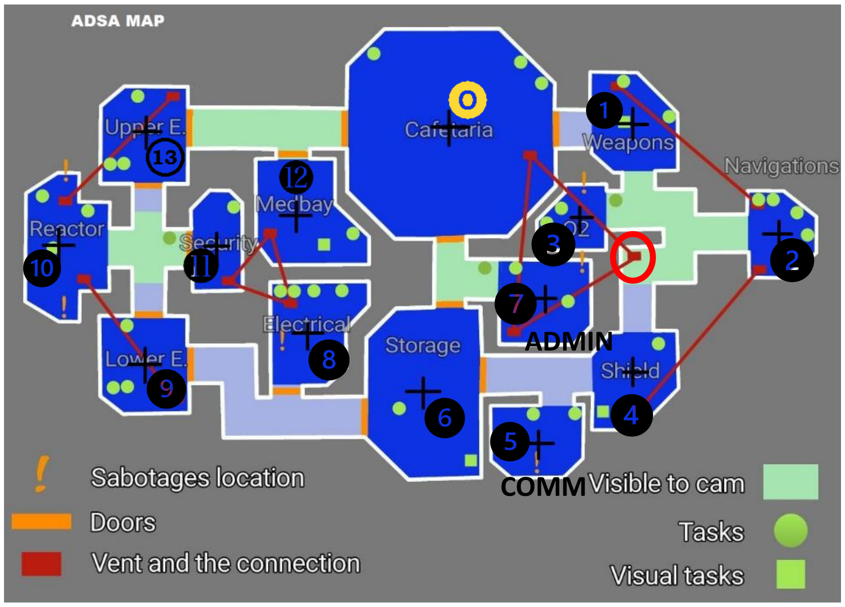
4) The results that we have is the following:



```
If the first Impostor is pseudo1

The second one might be :

pseudo3
pseudo4
pseudo5
pseudo7
pseudo8
pseudo9


If the first Impostor is pseudo4

The second one might be :

pseudo1
pseudo2
pseudo5
pseudo6
pseudo7
pseudo8


If the first Impostor is pseudo5

The second one might be :

pseudo1
pseudo2
pseudo3
pseudo4
pseudo6
pseudo9
```

Step 3 : I don't see him, but I can give proofs he vents!

1) For this step, we used the following image of the map and numbered the rooms like this:



Then, we measured the distance between each room knowing that a room is represented by its center. For the vents, we considered that they lead directly to the room, we did not take into account the distance from the vent to the center of the room. The measurements we made are in centimeters, it is also the travel time in seconds because a player moves at 1cm/sec.

We then wrote the distances between the rooms in an **adjacency matrix** respecting the numbering defined beforehand and adding the weight of the edges:

- For crewmates, the rooms are connected to each other only if there is a corridor between them. For rooms not connected by a corridor, the value of the link has been set to "infinite". The infinite value corresponds to a very large value, we have set this value to 10000.
- For the impostors, the rooms can also be connected by vents, in this case we have considered that the travel time is 0 between these rooms.

However, there is a special case for impostors, in fact there is a vent that is not located in any room. This vent is located on the right side of the map, it is circled in red on the previous image. This vent is connected to the "cafeteria" and "admin" rooms, for this particular case we have considered that the vent represents a corridor. Therefore :

- The "Cafeteria" room is connected to "O2", "Navigations" and "Shield", the "Weapons" room is already connected by a corridor.
- The "admin" room is connected to the same rooms, with the addition of "Weapons" to which it was not already connected.

The matrices created are in our "Program.cs" code, we have 2 matrices: one for the crewmates and one for the impostors taking into account the vents.

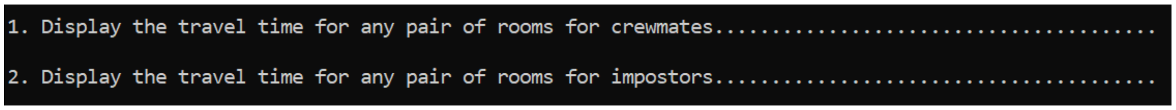
2) We are dealing with a non-oriented graph with links having weights. The problem that is asked is to determine the distance between all the possible pairs of rooms for the crewmates and the impostors. The algorithm that we thought suitable to compute all these travel times is the **Floyd-Warshall algorithm.**

The Floyd-Warshall algorithm is an algorithm that determines the distances of the shortest paths between all pairs of vertices in a graph. This algorithm takes as input an adjacency matrix giving the weight of an arc when it exists and the value ∞ otherwise. The weight of a path between two vertices is the sum of the weights on the arcs constituting this path. In our case, we have no negative weight.

The method has been coded in the "Map.cs" class with the name Shortest_path(), the prototype is: *public double[,] Shortest_path(double[,] matrix, int nb_vertices).* The "matrix" parameter corresponds to the adjacency matrix and the "nb_vertices" parameter corresponds to the number of rooms. This method returns the matrix with the travel time between each room.

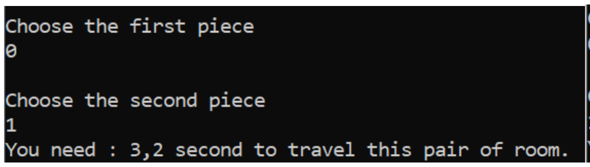
3) After having calculated the adjacency matrices containing all the travel times between the rooms, we proposed in our program to calculate the travel times between two rooms chosen by the user (for a crewmate or an impostor).

By choosing crewmate or impostor, and by entering the index of the two rooms you want, you get the travel time between the two rooms. These are options 1 and 2 on our graphical interface.

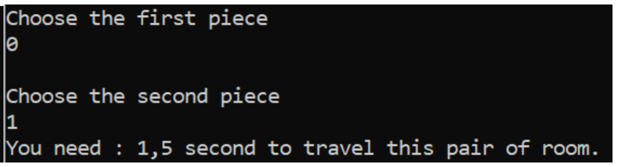


```
1. Display the travel time for any pair of rooms for crewmates......................................

2. Display the travel time for any pair of rooms for impostors......................................
```

The travel times are retrieved from the adjacent matrices using the indexes, for example between the "cafeteria" (index 0) and "weapons" (index 1) we have 3.2 seconds of travel time for a crewmate and 1.5 seconds for an impostor.



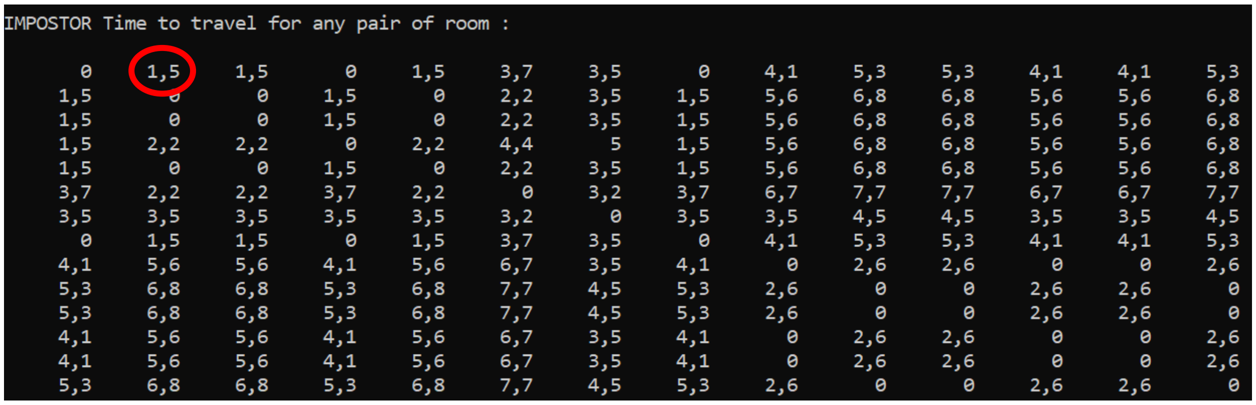
```
Choose the first piece
0

Choose the second piece
1
You need : 3,2 second to travel this pair of room.
```



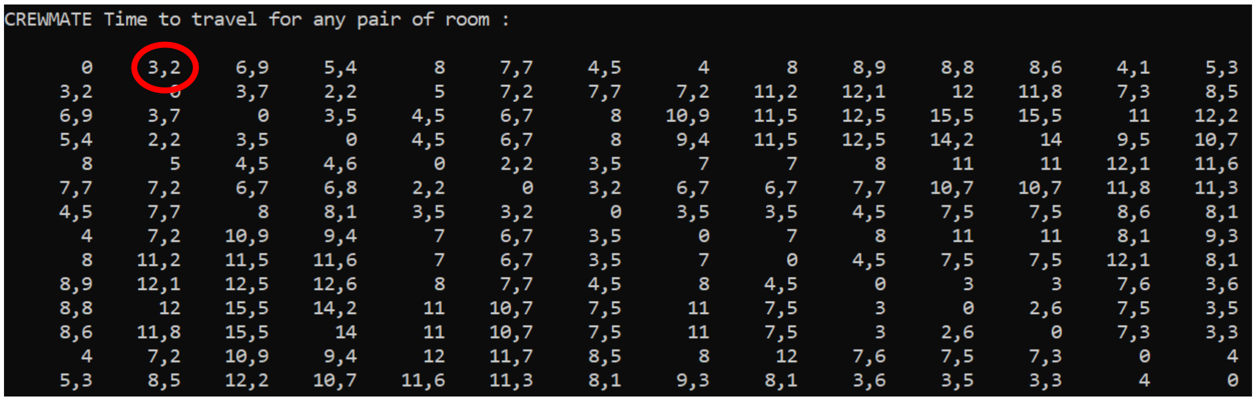
```
Choose the first piece
0

Choose the second piece
1
You need : 1,5 second to travel this pair of room.
```

These travel times are retrieved thanks to the Travel() method in the "Map.cs" class which has the prototype: *public void Travel(double[,] mat_distance)* with as input the mat_distance parameter which corresponds to the matrix of travel times.

4) To find the interval in which we can suspect that it is an impostor, we first displayed the two travel time matrices like this :



```
IMPOSTOR Time to travel for any pair of room :

     0    1,5    1,5      0    1,5    3,7    3,5      0    4,1    5,3    5,3    4,1    4,1    5,3
   1,5      0      0    1,5      0    2,2    3,5    1,5    5,6    6,8    6,8    5,6    5,6    6,8
   1,5      0      0    1,5      0    2,2    3,5    1,5    5,6    6,8    6,8    5,6    5,6    6,8
   1,5    2,2    2,2      0    2,2    4,4      5    1,5    5,6    6,8    6,8    5,6    5,6    6,8
   1,5      0      0    1,5      0    2,2    3,5    1,5    5,6    6,8    6,8    5,6    5,6    6,8
   3,7    2,2    2,2    3,7    2,2      0    3,2    3,7    6,7    7,7    7,7    6,7    6,7    7,7
   3,5    3,5    3,5    3,5    3,5    3,2      0    3,5    3,5    4,5    4,5    3,5    3,5    4,5
     0    1,5    1,5      0    1,5    3,7    3,5      0    4,1    5,3    5,3    4,1    4,1    5,3
   4,1    5,6    5,6    4,1    5,6    6,7    3,5    4,1      0    2,6    2,6      0      0    2,6
   5,3    6,8    6,8    5,3    6,8    7,7    4,5    5,3    2,6      0      0    2,6    2,6      0
   5,3    6,8    6,8    5,3    6,8    7,7    4,5    5,3    2,6      0      0    2,6    2,6      0
   4,1    5,6    5,6    4,1    5,6    6,7    3,5    4,1      0    2,6    2,6      0      0    2,6
   4,1    5,6    5,6    4,1    5,6    6,7    3,5    4,1      0    2,6    2,6      0      0    2,6
   5,3    6,8    6,8    5,3    6,8    7,7    4,5    5,3    2,6      0      0    2,6    2,6      0
```

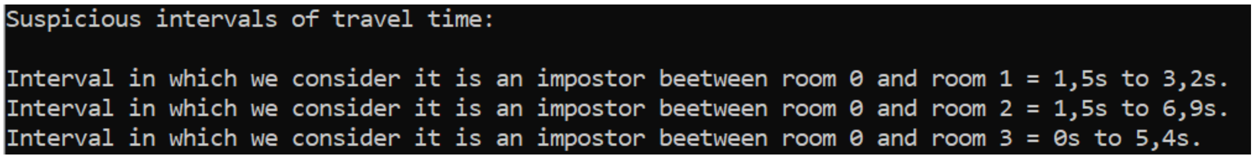


```
CREWMATE Time to travel for any pair of room :

     0    3,2    6,9    5,4      8    7,7    4,5      4      8    8,9    8,8    8,6    4,1    5,3
   3,2      0    3,7    2,2      5    7,2    7,7    7,2   11,2   12,1     12   11,8    7,3    8,5
   6,9    3,7      0    3,5    4,5    6,7      8   10,9   11,5   12,5   15,5   15,5     11   12,2
   5,4    2,2    3,5      0    4,5    6,7      8    9,4   11,5   12,5   14,2     14    9,5   10,7
     8      5    4,5    4,6      0    2,2    3,5      7      7      8     11     11   12,1   11,6
   7,7    7,2    6,7    6,8    2,2      0    3,2    6,7    6,7    7,7   10,7   10,7   11,8   11,3
   4,5    7,7      8    8,1    3,5    3,2      0    3,5    3,5    4,5    7,5    7,5    8,6    8,1
     4    7,2   10,9    9,4      7    6,7    3,5      0      7      8     11     11    8,1    9,3
     8   11,2   11,5   11,6      7    6,7    3,5      7      0    4,5    7,5    7,5   12,1    8,1
   8,9   12,1   12,5   12,6      8    7,7    4,5      8    4,5      0      3      3    7,6    3,6
   8,8     12   15,5   14,2     11   10,7    7,5     11    7,5      3      0    2,6    7,5    3,5
   8,6   11,8   15,5     14     11   10,7    7,5     11    7,5      3    2,6      0    7,3    3,3
     4    7,2   10,9    9,4     12   11,7    8,5      8     12    7,6    7,5    7,3      0      4
   5,3    8,5   12,2   10,7   11,6   11,3    8,1    9,3    8,1    3,6    3,5    3,3      4      0
```

Knowing that an impostor is either faster or as fast as a crewmate, we defined the interval as : Impostor travel time - Crewmate travel time.

For example, as circled in red on the images above, the time interval in which we can suspect that the player is an impostor between room 0 and 1 (cafeteria and weapons) is: 1.5-3.2sec.

Our program displays the intervals between all possible rooms like this:



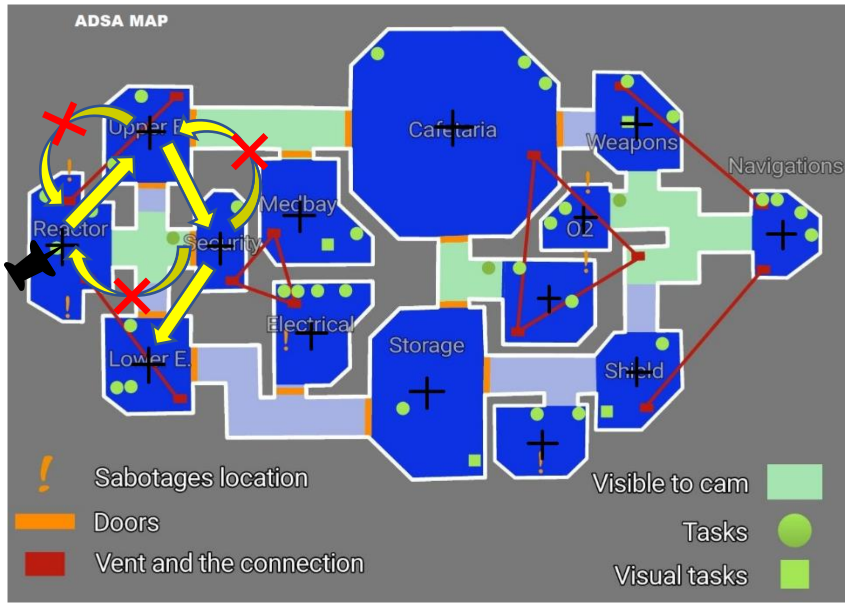
```
Suspicious intervals of travel time:

Interval in which we consider it is an impostor beetween room 0 and room 1 = 1,5s to 3,2s.
Interval in which we consider it is an impostor beetween room 0 and room 2 = 1,5s to 6,9s.
Interval in which we consider it is an impostor beetween room 0 and room 3 = 0s to 5,4s.
```

Step 4 : Secure the last tasks

1) For the map model, we used the same model as in the previous step. We took the adjacency matrix for crewmate that we filled with the weights if the rooms are connected and the value "infinite" if they are not connected.

2) Les joueurs souhaitent passer par toutes les salles le plus rapidement possible et sans repasser par la même salle. La solution qui nous semble adapté à ce problème est le **chemin Hamiltonien**. Un **chemin hamiltonien** d'un graphe orienté ou non orienté est un chemin qui passe par tous les sommets une fois et une seule. Pour ce problème, nous ne voulons pas de cycle car revenir au point de départ n'est pas nécessaire.

3) Players wish to go through all the rooms as quickly as possible and without passing through the same room again. The solution that seems to be adapted to this problem is the Hamiltonian Path. A Hamiltonian path of a directed or non-directed graph is a path that visits each vertex exactly once. For this problem, we do not want a cycle because going back to the starting point is not necessary.



For example, if we start our path in the Reactor room then this will be our starting point and the algorithm will work like this:

A. The program will check each neighbor if it has not been visited and will choose a room among its neighbors. In the image above, the players go to "Security" and cannot return to "Reactor". After this step, our circuit is Reactor-Security, we also add the travel time.

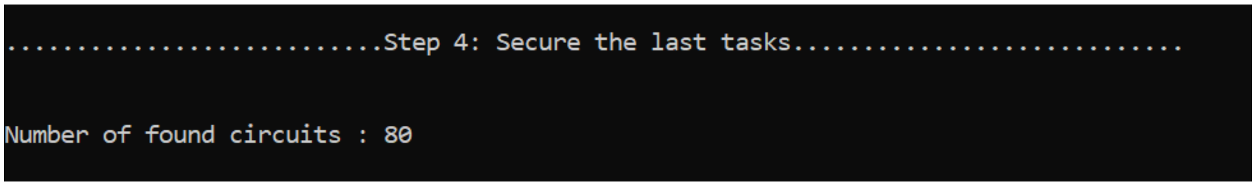
B. At this point, the program realizes that it can no longer go through "Reactor" or "Security" because they have already been visited, so it chooses "Lower". At this stage, our circuit is Reactor-Security-Lower, we add the corresponding travel time.

C. The program repeats steps A and B in a loop, until all rooms are covered (the number of rooms in the circuit is 14). Otherwise, it stops when all neighbor rooms have already been visited, in which case it is no longer possible to move forward: we try a new Hamiltonian path.

The program tries every rooms as a starting point to get all existing Hamiltonian paths on the map.

This algorithm is realized by the method Circuits_One_Room() in the class "Map.cs". The method's prototype *is public void Circuits_One_Room(Room current_room, Circuit circuit)*: it takes as parameter the room it is currently in and the circuit that we fill as we go along. The method returns the Hamiltonian path and adds it to the list of existing paths.

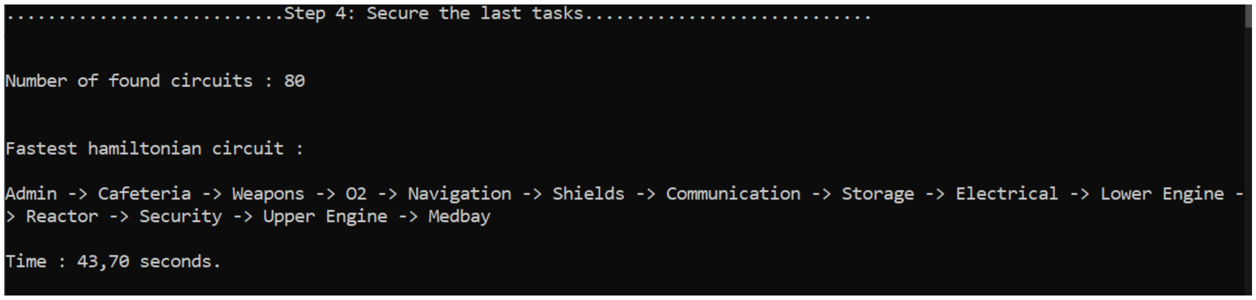
By running the program, we get 80 possible Hamiltonian paths as shown in the interface :



```
..........................Step 4: Secure the last tasks...........................


Number of found circuits : 80
```

4) After having found all the Hamiltonian paths, we have to find the shortest of them all. Our program goes through all the Hamiltonian paths and looks for the fastest one, it returns the shortest path when it has found it. Then it displays the shortest path and the time it took to go through all the rooms on this path.

The interface displays the result in this way :



```
..........................Step 4: Secure the last tasks...........................


Number of found circuits : 80


Fastest hamiltonian circuit :

Admin -> Cafeteria -> Weapons -> O2 -> Navigation -> Shields -> Communication -> Storage -> Electrical -> Lower Engine -
> Reactor -> Security -> Upper Engine -> Medbay

Time : 43,70 seconds.
```

The solution is: Admin -> Cafeteria -> Weapons -> O2 -> Navigation -> Shields -> Communication -> Storage -> Electrical -> Lower Engine -> Reactor -> Security -> Upper Engine -> Medbay for a travel time of 43.7 seconds. It is also possible to display all 80 circuits on the interface.