PODOLAK Yan – ATALLAH Louis
DIA 1

Rapport – ADSA

Step 1 : To organize the tournament

Method tournament :

This method simulates the whole tournament.

→Create an empty tree.

→Fill the tree with 100 players with a score of 0

→Play 3 games randomly

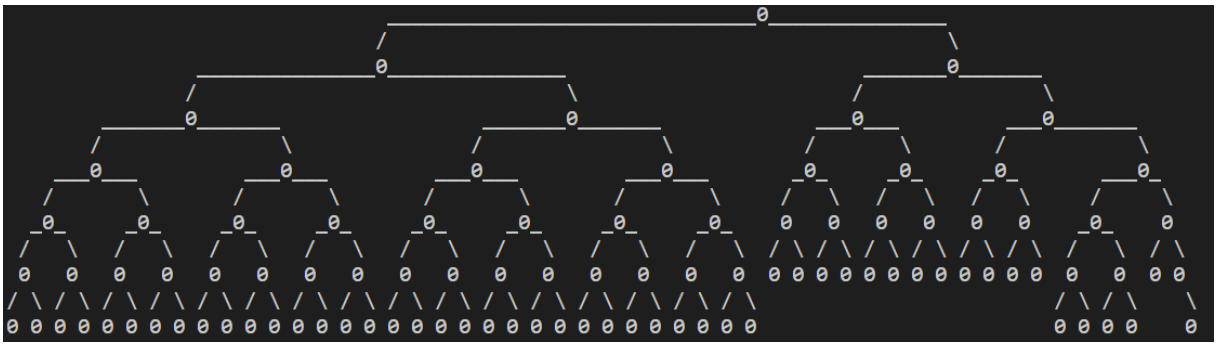
→Repeat 9 times : Order the players by their score, play a game (players regrouped in groups of 10 according to their ranking), order the players with their new score, drop the last 10

→There are 10 players remaining. Reset their score.

→Play 5 games randomly

→Display the final tree and the top 10 with the podium at the top.

1 & 2 – To represent a player and its score, we have created a class Player which contains the name of the player, its score and its category which will be modified each game. Then we used an AVL Tree to put all the players ordered by their score. The AVL Tree represents the tournament with all the players.



Here, you can see the AVL Tree of the beginning with 100 players with a score of 0.

3 – We created a method called Random_Score to randomize the score of a player. It takes a player and a Boolean win_Crewmate as parameters. The win_Crewmate Boolean determine whether if the crewmates won (then it's equal to True) of lost (==False). Then according to the category of the player (crewmate or impostor), we attribute the points :

- Between 0 and 9 if the player is an impostor and lost (amid kills and undiscovered murders)
- Between 10 and 12 if the player is an impostor and won
- Between 5 and 12 if the player is a crewmate and won

- Between 1, 3 and 4 if the player is a crewmate and lost (task achieved, impostor demasked, both)

4 – As the deletion in the tree is very complicated to deal with. Instead of changing all the scores in the Tree, we chose to build another tree with the updated scores. By doing that, it's easier to follow each step of the tournament.

→ Create a new tree

→For each player, reset its child to None and its height to 1

→Add the players to the new tree

5- We have created the method called Game_Simulation to create random games. It takes a Tree (the database) and the number of games wanted as parameters.

→Divide the players in groups of 10

→For each group, generate 2 indexes for the impostors

→Set the category of the 2 players as impostors (the others are by default crewmates)

→Generate randomly a Boolean to know if the crewmates won

→Set the scores of all the players according to who won

→ Create a new tree

→For each player, reset its child to None and its height to 1

→Add the players to the new tree

→Modify the scores by dividing them to get the mean

→Return the tree

6 & 7 – To create games based on ranking, we use the method called Game_Simulation2. It basically does the same thing than the previous method but before each game, we sort the players by their score.

Then we do as before to simulate the game.

After attributing the scores, we sort again the players and delete the 10 lasts.

Then, as in the previous method, we add the players with their updated scores in a new tree. We do all this 9 times, until there are only 10 players left.

8 – To display the top 10 players and the podium, we use the method Display_Top_Players_Tree which sort all the players by their score and display them from the first to the last. The 3 first are marked as in the podium.

```
******Here is the top 10 players after the final 5 games*******
rk  :   Player  -       Points
n° 1   :        21      -       11      PODIUM !
n° 2   :        5       -       11      PODIUM !
n° 3   :        88      -       11      PODIUM !
n° 4   :        49      -       10
n° 5   :        56      -       10
n° 6   :        96      -       10
n° 7   :        93      -       10
n° 8   :        1       -       10
n° 9   :        2       -       10
n° 10   :       51      -       3
```

Step 2 : Professor Layton < Guybrush Threepwood < You

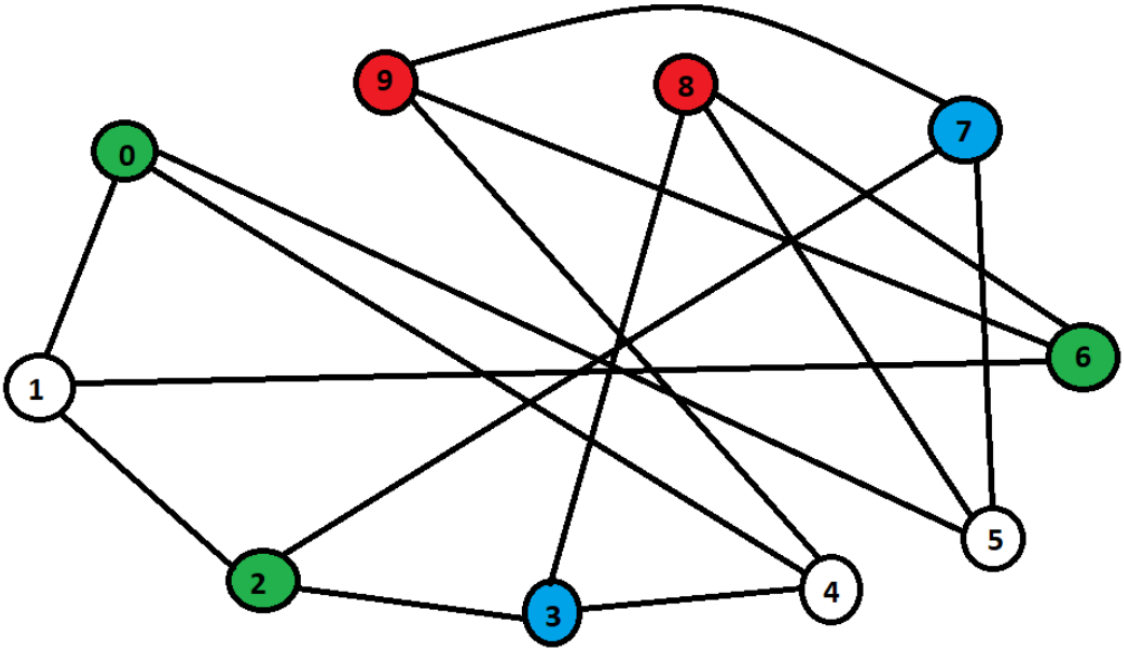
1 - In this step we are trying to find a set of probable impostors among 10 players. To tackle this issue, we have implemented a 4th-coloration graph to distinguish the players, and players they have met up.

According to this case, each player has seen 3 other players, thus we need at least 4 colors to set up a graph coloration.

It is a non-weighted an undirected graph. Here the purpose is mainly to represent the adjacency of the vertices.

A we can see below, a representation of the graph we were inspired from. Each vertex is represented by a number which is the player id, and an assigned color.

For example, the player 0 is the vertex number 0, and it is green painted. This implies that none of its neighbors are green.



To solve this problem, we instantiate 10 players. Then we create 2 dictionaries “graph” and “graph2” which contain respectively a player/the id of the player as key and the list of players/players’ id met.

We use the “graph” dictionary to build the graph by adding each edge via the “add_edge” method. Once done, we start the coloration of the graph with the “coloration” method. It is a method which coloring a graph and manage the graph coloration rules.

Here is the algorithm of our coloration method:

- *Setting a list of nodes color (-1 by default means that it is uncolored)*

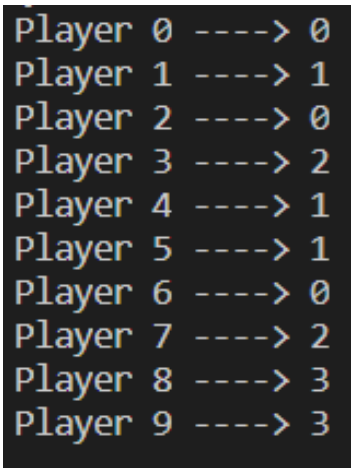
- *For k from 0 to 30 by 3 (because there are 30 lists in the list (3 per players (one per neighbor)))*

    - *Check whether the node is already colored*

    - *If it is not: gather every neighbors’ colors and take the free one*

    - *Replace in the color list by the color determined above*

It gives the following coloration:



Once we have colored the graph, it is much simpler to identify a set of impostors.

By using the “Find_Impostors” method we can determine the probable impostors amid the coloration graph.

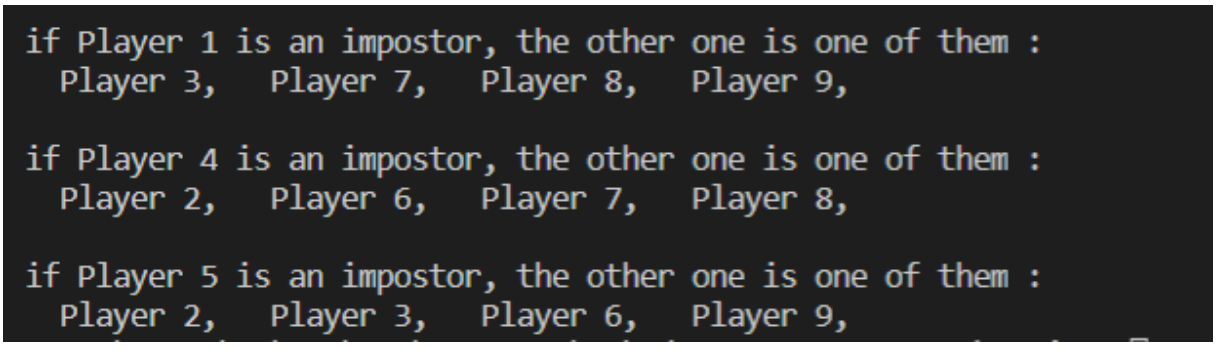
The algorithm of this method is shown below:

- *List of suspects (I.e the players that have seen the dead player)*

- *Foreach suspect in the suspects list:*

    - *If the player is not a neighbor of the suspect, nor the same color and not the dead player:*
        - *Add the player to the set*

The result is shown below:



According to the death of the player 0 and the implementation of the “Find_Impostors” method, we have reached a result from the method explain above.

It takes the three suspects who are the players that have met the dead player and we check for each one his color and his meetings. After such comparing, if it grabs all the criteria, the player is added to the set.

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

1 - In this step, we model the 2 maps by creating 2 graphs. The crewmate map has no vents, while the impostor map is the same map as the crewmates' but with the vents. Hence, the impostor map is just a copy of the crewmate one, then we add the vents.

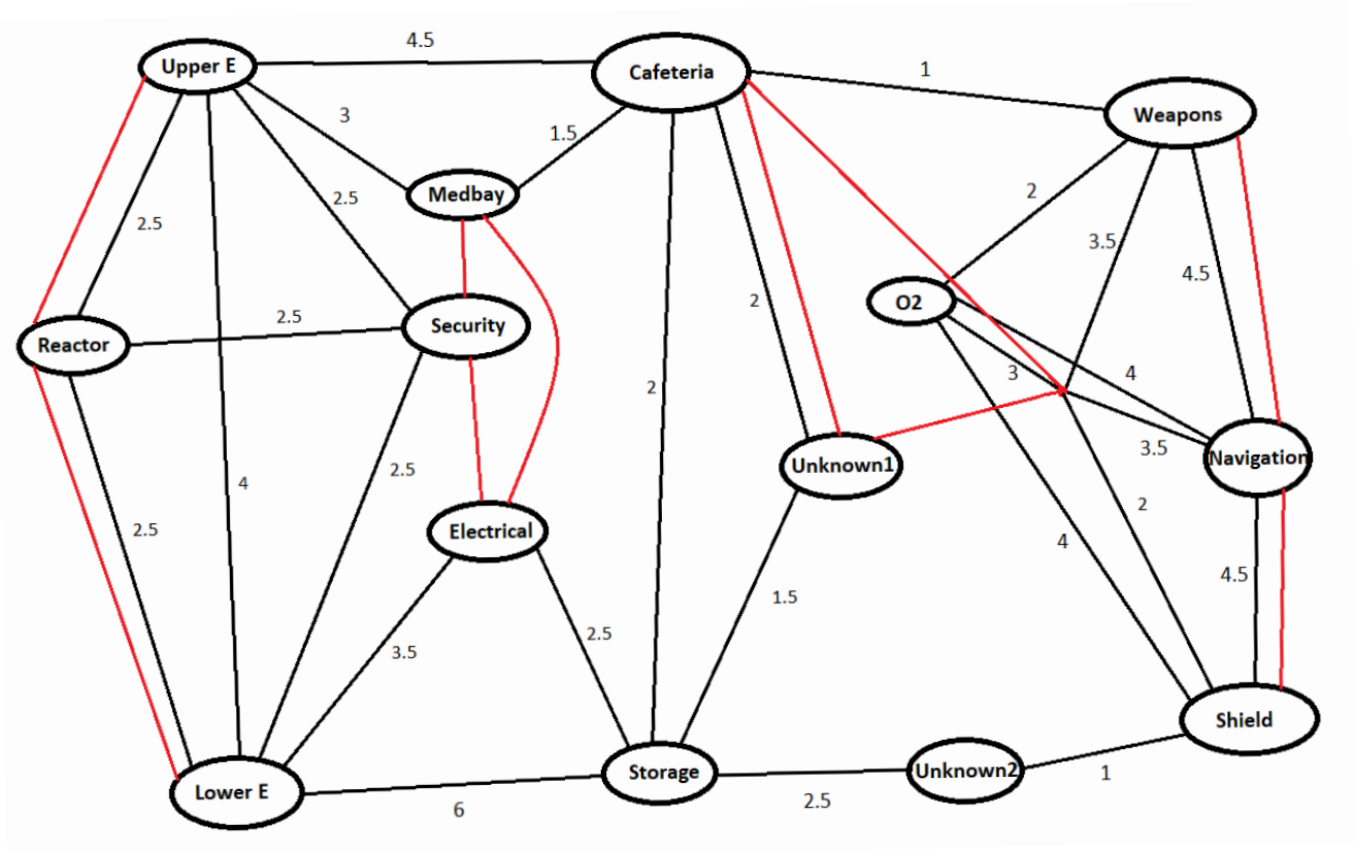
To create the graph, we initialize the matrix with a very big value except the diagonal with 0.

After that, we replace the values in the adjacency matrix by the weight of the edges.

To reproduce the impostor map we use the "deepcopy" function, and we add the zeros in the adjacency matrix to represent that there is no time between the two rooms concerned.

At this moment the 2 maps are created, we implement a dictionary with index of the rooms as keys and the rooms' names as values. It is mainly used for the display.

Below, the illustration of the map with the vents:



2 - We need to determine the time to travel between each pair of rooms. In order to solve that we will use the Floyd-Warshall algorithm because it is conceived to return the weight from one room to another.

To implement the algorithm, we use the algorithm given by the course. The purpose is to find the shortest distance between a direct road and an intermediary road.

Below the algorithm implemented:

- *Foreach node (element) in the matrix*
    - *If the intermediary road is shorter*
        - *Replace the element in the matrix*

This method displays the matrix of the shortest way (in seconds) through each pair of rooms.

3 & 4 - Here are the two models' timespan:

As a crewmate:

start/end	Reactor	Upper E	Lower E	Security	Medbay	Electrical	Cafeteria	Storage	Unkown1	Unknown2	O2	Weapon	Navigation	Shiel
Reactor	0.0	2.5	2.5	2.5	5.5	6.0	7.0	8.5	9.0	11.0	10.0	8.0	12.5	12.0
Upper E	2.5	0.0	4.0	2.5	3.0	7.5	4.5	6.5	6.5	9.0	7.5	5.5	10.0	10.0
Lower E	2.5	4.0	0.0	2.5	7.0	3.5	8.0	6.0	7.5	8.5	11.0	9.0	13.5	9.5
Security	2.5	2.5	2.5	0.0	5.5	6.0	7.0	8.5	9.0	11.0	10.0	8.0	12.5	12.0
Medbay	5.5	3.0	7.0	5.5	0.0	6.0	1.5	3.5	3.5	6.0	4.5	2.5	7.0	7.0
Electrical	6.0	7.5	3.5	6.0	6.0	0.0	4.5	2.5	4.0	5.0	7.5	5.5	10.0	6.0
Cafeteria	7.0	4.5	8.0	7.0	1.5	4.5	0.0	2.0	2.0	4.5	3.0	1.0	5.5	5.5
Storage	8.5	6.5	6.0	8.5	3.5	2.5	2.0	0.0	1.5	2.5	5.0	3.0	7.5	3.5
Unkown1	9.0	6.5	7.5	9.0	3.5	4.0	2.0	1.5	0.0	4.0	5.0	3.0	7.5	5.0
Unknown2	11.0	9.0	8.5	11.0	6.0	5.0	4.5	2.5	4.0	0.0	5.0	5.5	5.5	1.0
O2	10.0	7.5	11.0	10.0	4.5	7.5	3.0	5.0	5.0	5.0	0.0	2.0	4.0	4.0
Weapon	8.0	5.5	9.0	8.0	2.5	5.5	1.0	3.0	3.0	5.5	2.0	0.0	4.5	6.0
Navigation	12.5	10.0	13.5	12.5	7.0	10.0	5.5	7.5	7.5	5.5	4.0	4.5	0.0	4.5
Shield	12.0	10.0	9.5	12.0	7.0	6.0	5.5	3.5	5.0	1.0	4.0	6.0	4.5	0.0

As an impostor:

start/end	Reactor	Upper E	Lower E	Security	Medbay	Electrical	Cafeteria	Storage	Unkown1	Unknown2	O2	Weapon	Navigation	Shiel
Reactor	0.0	0.0	0.0	2.5	2.5	2.5	4.0	5.0	4.0	7.0	7.0	6.0	6.0	6.0
Upper E	0.0	0.0	0.0	2.5	2.5	2.5	4.0	5.0	4.0	7.0	7.0	6.0	6.0	6.0
Lower E	0.0	0.0	0.0	2.5	2.5	2.5	4.0	5.0	4.0	7.0	7.0	6.0	6.0	6.0
Security	2.5	2.5	2.5	0.0	0.0	0.0	1.5	2.5	1.5	4.5	4.5	3.5	3.5	3.5
Medbay	2.5	2.5	2.5	0.0	0.0	0.0	1.5	2.5	1.5	4.5	4.5	3.5	3.5	3.5
Electrical	2.5	2.5	2.5	0.0	0.0	0.0	1.5	2.5	1.5	4.5	4.5	3.5	3.5	3.5
Cafeteria	4.0	4.0	4.0	1.5	1.5	1.5	0.0	1.5	0.0	3.0	3.0	2.0	2.0	2.0
Storage	5.0	5.0	5.0	2.5	2.5	2.5	1.5	0.0	1.5	2.5	4.5	3.5	3.5	3.5
Unkown1	4.0	4.0	4.0	1.5	1.5	1.5	0.0	1.5	0.0	3.0	3.0	2.0	2.0	2.0
Unknown2	7.0	7.0	7.0	4.5	4.5	4.5	3.0	2.5	3.0	0.0	3.0	1.0	1.0	1.0
O2	7.0	7.0	7.0	4.5	4.5	4.5	3.0	4.5	3.0	3.0	0.0	2.0	2.0	2.0
Weapon	6.0	6.0	6.0	3.5	3.5	3.5	2.0	3.5	2.0	1.0	2.0	0.0	0.0	0.0
Navigation	6.0	6.0	6.0	3.5	3.5	3.5	2.0	3.5	2.0	1.0	2.0	0.0	0.0	0.0
Shield	6.0	6.0	6.0	3.5	3.5	3.5	2.0	3.5	2.0	1.0	2.0	0.0	0.0	0.0

We can observe that the whole map is reachable much faster as an impostor than as a crewmate.

Hence, we can also find the impostors by comparing the travel time between each player.

Step 4: Secure the last tasks

1 - To work on this step, we use the map we previously implemented. As this step has a goal to doing all tasks, we retake the crewmate map to determine a path passing only once by each room because tasks concern only the crewmate. It implies we don't need to have the vents in the map. This characteristic will make a difference to find the path because thanks to the vents, new edges are added and that means other paths could be found.

2 & 3 - To find this path, we must only pass once through each room. To solve this issue, we have chosen the Hamiltonian path technic. It consists of passing through each node of a graph and joining the last node to the first one. It is a kind of cycle.

To deal with our problem we use a variant of the Hamilton method. It is the same as the original, but we remove the link between the last room visited and the starting room.

Indeed, here we don't need to have a cycle.

We choose a starting room, and the algorithm returns a path including every rooms or if there is no path existing it returns no path.

In our test solution we have decided to start from the “Medbay” room which is indexed at 4 because we are certain that it exists a Hamiltonian path from this room.

As said previously, we retake the crewmate map but here we don’t need a weighted graph, thus we make some preprocessing before calling the Hamilton method.

When we copy the crewmate adjacency matrix, it is full of 0, weights, and INF (which is our very big value). Here, we just need a matrix that represents whether there is an edge between two rooms or not, hence it is binary. That’s why we replace the INF value by zeros, and the non-zero values by 1. 1 indicates there is an edge while 0 doesn’t.

Once the adjacency matrix has been prepared, we can fit our method to solve and show the pass the crewmates have to follow.

The Hamilton class is composed of three methods that are dependent on each other.

The two first methods are used in the Hamilton method which is main one.

The method called “check” enables to verify the adjacency of a room to another and also to check whether to room has already been passed or not (I.e if the room is already in the path). It returns a boolean to indicate if it is a valid next room or not.

Algorithm of check method:

- *If the room is already in the path*
    - *Return false*
- *If the 2 elements are not adjacent*
    - *Return false*
- *Else*
    - *Return true*

Then we have the “Hamilton_aux” method which returns also a boolean and assign the order of the rooms in the path.

First, we check if the vertex is the final one. If it is, then we return true which means the end of the algorithm.

Second, we instantiate a list of the indexed rooms except the starting room because we don't need to repass on it.

Then for each room in the list we assign its position in the path. If at a moment there is no path found between a room and the rooms still available, it returns false. That means there is no path from the starting room. It is a recursive method.

The algorithm of the "Hamilton_aux" method:

- *If it is the last vertex*
    - *Return true*
- *List of all rooms except the starting one*
- *Foreach room in the list*
    - *If check method is true*
        - *Assign the position to the room*
        - *If Hamilton_aux with the next position is true*
            - *Return true*
- *Return false*

The third method id the Hamilton one, it displays the path computed in the previous method.

In this method we initialize the path list with –1 as value. We set the starting room index to the first element in the path list and we start the "Hamilton_aux" method with the position 1 (the second in the list).

The algorithm of the Hamilton method:

- *Create a path list of –1*

- *Set the starting room to the first element*

- *If Hamilton_aux is false*

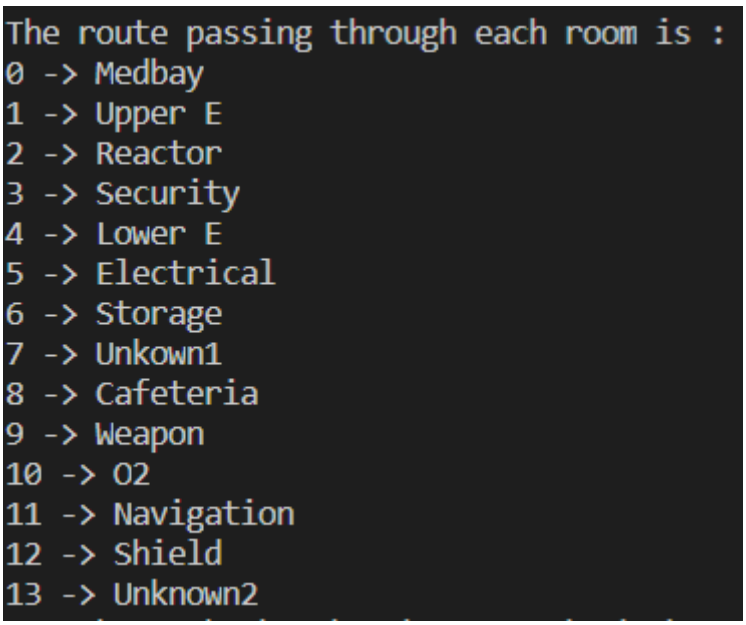
    - *No such path*

- *Else*

    - *Print the path*

4 - To test this solution method, we have decided to take the medbay room as the starting room. We recall that the algorithm gives only one solution of the several existing.

Here is the order to travel through each room:



As we can, the path includes the 14 rooms of the map and none of them are visited twice.

(Note that it is not the same display as it is in the .py file because of its inconvenience to paste it in the report).