Advanced Data Structure and Algorithms

ESILV FINAL PROJECT

By LASCAR Sarah & MAIER Flora

STEP 1 :

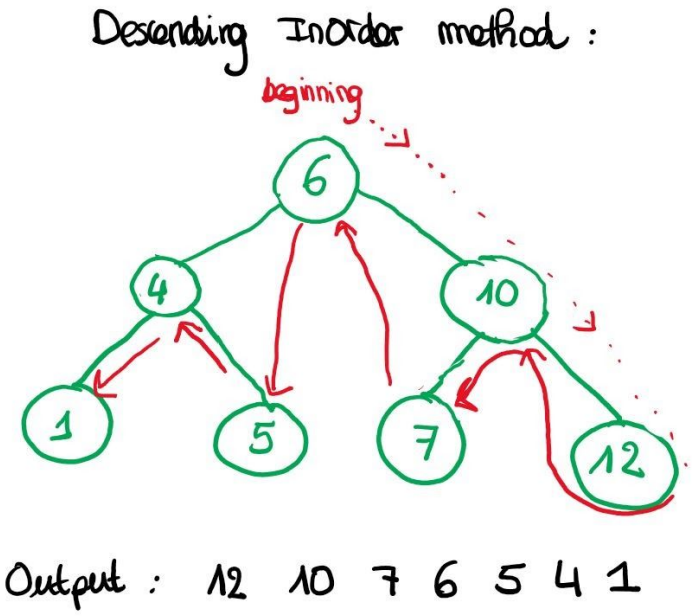
1. To represent a Player and its Score, I created the **class player** with attributes :

- id
- pts : number of points gained at the end of the game
- nb_games : total number of played games
- score : mean of points of all the games
-

   In this class we create 2 methods to display a Player, the main one, **__str__(self)** shows only the id of the player and the second one, **show_more(self)** displays the id, score and number of games already done.

2. The most optimized structure where players are stored, with a log complexity to reach an element which corresponds to a score is the AVL Tree, seen in our class. That's why I created the classes **TreeNode()** and **AVL_Tree()**. This time, a tree node is not only an integer value, but a **Player objec**t and the key we are looking at is the **score** of the player. Through this class we can display the whole Tree to have a better image of the database. The **AVL_Tree** class allows us to create the database, by inserting each node one by one.
   Everytime we insert a node, we have to check if the tree is unbalanced after it, and if it is, we need to apply one of these 4 rotation methods seen in class : **Left-Left, Right-Right, Left-Right, Right-Left**. In our case, it is possible that several Players have exactly the same score, that's why the only modification I did to our usual code was to include the **equality case** in the Left-Left and Right-Right method.
   The methods **leftRotate, rightRotate, getHeight and getBalance** are under-method to the insert one.

   The last point that is modified for our project compared to a classic AVL Tree program is that here we are going to use the **inOrder(self, root) method** to display the database depending on the score of the players, in the descending order. Plus, the method will return the sorted database as a list instead of just displaying the players. By doing this we can easily update the database depending on the ranking of the players, before creating new games.



At the early beginning of our project, we created a method **createDB()** that inserts 100 Players **in a list** with id from 1 to 100.

3. We created a **class game** with a list of players as an attribute, but we will always have 10 players in this list. Each time a game is played, the method **assign_points(self)** is called **:**
   It uses the function **random.randint()** to assign a game score between 0 and 12 to each player and increase by 1 his number of games played.

4. When we call the **assign_points(self)** function from the game class, it calls the function **update_score()** from the player class that computes the mean of all the games scores until now to obtain the final score of the player. This method uses the following formula :

   score = (pts + score * (nb_games - 1)) / nb_games

   It takes the number of points obtained from the current game and computes the mean of all the points obtained to each game regarding the actual mean (score).

5. The method **random_game(database,nb_games)** takes as an argument the database and creates a game with 10 players randomly selected.

   Actually this function is not totally random since every player needs to do 3 first games, so we first consider only the players that did less than “nb_games” games in our possibilities.
   The goal here is to go step by step to obtain 3 games done by all the players : first, everybody does 1 game, then everybody does the second game and finally they do the third and last random game. That’s why in the **tournament function** we call this method 10 times with a number of games equal to 1, then 10 times for the number

equal to 2 and finally 10 times equal to 3. In the final we did 30 random games and all the players did 3 games.

In the first lines of this method we created the lists **list_poss** corresponding to all players that did less than a certain number of games, and **list_players** that is the list of 10 players chosen randomly from **list_poss.** To choose randomly these 10 players, we use the function **random.sample (range (0, len(list_poss)-1), 10)** that selects their 10 indexes.

Then a game is created and we call the **assign_pts()** function from the game class to create the game's results and update the points of each player.
The last thing to do now is to update the database depending on the score of each player.

We want to sort the database to have first the best players with the highest scores and then the worse ones. The best method is to now to create an empty AVL tree and to insert one by one every player of the database. Thanks to the **insert(self,root,key)** function from the **AVL_Tree** class, the created tree is going to be correctly created depending on all the players' score. After it, we apply the **inOrder(self,root)** method that returns the actualized sorted database as a list that we are going to reuse.

This **random_game(database)** method is returning the new sorted database, where all changes due to the game occured

6. After all our players did 3 random games, we want them to play games depending on their ranking. The function **create_game_ranking (database, rank_start, rank_end , willUpdate)** is choosing the 10 players in the database that have a ranking between rank_start and rank_end. To do this the easier way is to look at their index in the database that corresponds to (ranking - 1), that's why we did a loop **for i in range(rank_start-1, rank_end)** to insert the concerned players in a list.

   After we created the game and assigned the points to the players, we can choose to update or not the database. Indeed for example if we want to do a game for players with a ranking between 11 and 20, and then a between 1 and 10, if we immediately update the ranking of players from the fist game, some of them may raise a ranking between 1 and 10, as a consequence they would do 2 games instead of one. That's why we decided to add the variable **willUpdate** that allows the program to update immediately after the game is created, by setting willUpdate=True or not.

   When we want the database to be updating, we simply call the function update_database (database) that creates an **AVL_Tree** with the players from the database and return the sorted list thanks to the **inOrder(self,root)** method, no matter how many players we have in the database.

7. We created a method **tournament (database)** that applies the 3 random games to every player and then applies games depending on the ranking. The goal is to drop the 10 worst players after all the players did exactly one game, so that at the end the database owns only 10 players, the best ones.

   To do this, we need to use a loop **for j in range(9)**, so that at the end of one loop all the players did one game, the database is updated, and the 10 worst ones are ejected from the game. We chose 9 and not 10 in order to obtain a database of 10 players at the end.

   To make all the players do 1 game, we have to be careful of the rankings. If we still have a database of 100 players, the ones with ranking between 91 and 100 will play together, then the ones between 81 and 90, and so on until the rankings between 1 and 10. But if after we ejected some players we have just 50 of them left, the plays are for players with ranking starting from 41 to 51 and not anymore 91 to 100.

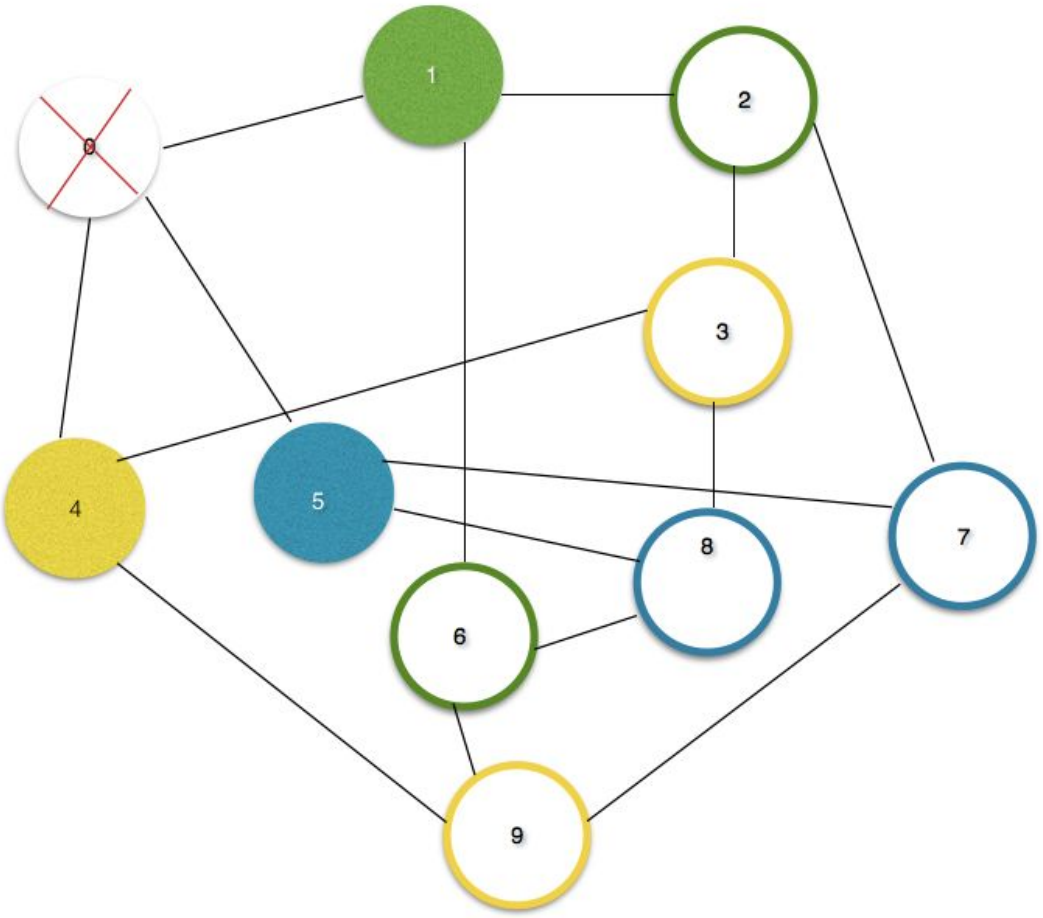
   That's why at the beginning of the j-loop we create the variables **rank_start = 91-10*j, rank_end=100-10*j** and **nb_games_to_do = rank_end//10.**
   rank_end corresponds to the number of players still in the game, depending on how much times we already ejected a bunch of 10 players, and **rank_start** is just rank_end - 10, so **[rank_start, rank_end]** corresponds to the indexes of the first players of the loop.
   If we have for example 80 players still in the game, we know that in order to make them all play one game we have to organize 8 games. The variables **nb_games_to_do** is just compiling this number of games to do with the players availables.

   Now that we have the number of players still in the game and the number of games we have to do, we have to create another loop to start all these games. In the loop **for i in range(nb_games_to_do)** we call the function **create_game_ranking(database, rank_start, rank_end, False)** to make all the players of the database do one game. At each loop **rank_start** and **rank_end** are decreasing by 10, until all the games are done. We are not updating the database while all players did not do their game, that's why we set the attribute False in this method and we call the method **update_database(database)** out of the i-loop.

   After everybody did one game and 10 players had been ejected, and after the database had been updated, 9 times, we have the 10 best players left in the database, that did in total 12 games, 3 random ones and 9 normal ones.
   Our method **tournament(database)** is returning this new database of only players.

8. Now that we obtained our 10 best players of the tournament, we call, in the main, the method **final_game(database)**. It makes the players do all together 5 games, each time the results are directly updated in the database.To do this we simply used the function **create_game_ranking(database,1,10,True),** since we just have 10 players. Then we display the database to see the TOP10 players after this final game and we display the 3 first players of it to show the PODIUM.

STEP 2 :



1. The relation “have seen” between players can be represented as an undirected graph. Indeed, if one player 1 sees another player 2, then this implies that player 2 has also seen player 1.
   This graph will be also an unweighted graph because there is only the relation “have seen” between players, and this latter is already represented by the edge between the vertices of the graph.
   Above the representation of the graph where each player is represented by a vertex, and the relation “have seen” between two player by an edge.

2. We saw in question 1 the graph that we use to represent the "having seen" relations between two players.
   We will now see how this graph will allow us to find a set of probable impostors.
   First, player 0 is crossed out because he cannot be one of the potential impostors since he has been killed.
   Second, we're going to color each of the players who saw player 0 with a unique color. They will be the supposed impostors for whom we will look for a set of potential impostors for each.

Finally, we circle the players who are linked to a supposed impostor of the same color as the latter is colored. All players who are not circled in this color will be potential impostors in a team with the supposed impostor.
For example, on our map, we colored player 1 (who is a supposed impostor) green. He saw players 0, 2 and 6. Now 0 is dead, we only circle 2 and 6 in green.
Thus, the players: 3,4,5,7,8,9 are not circled in green, so they are potential impostors who team up with 1.

3. In order to find a set of potential impostors, we will start by assuming that player 1, 4 and 5 are each in turn impostors. This assumption is made because they are the only ones to have seen player 0 who was reported dead. Then, we will look for the impostors potentially in a team with the supposed impostor player. For example: we have assumed player 1 is an impostor, we are looking for who can potentially team up with him.
   For this we will assume that they are all potential imposters at the start (except player 0 who is dead and player 1, 4 or 5 who is already supposed to be an impostor and therefore cannot team up with himself) and we put them in a list.
   The statement explains to us that two impostors never move together, so we consider that they have never seen each other, so there is no relation “have seen” between them.
   Thus, for each player on the list we check if he has seen the supposed impostor. If so, we remove him from the list, else we keep him. As soon as two players have been removed from this list, we can move on to the next supposed impostor player, because 1, 4 and 5 are only linked to two players who are still alive.
   The algorithm returns the list of potential impostors according to the supposed impostor.

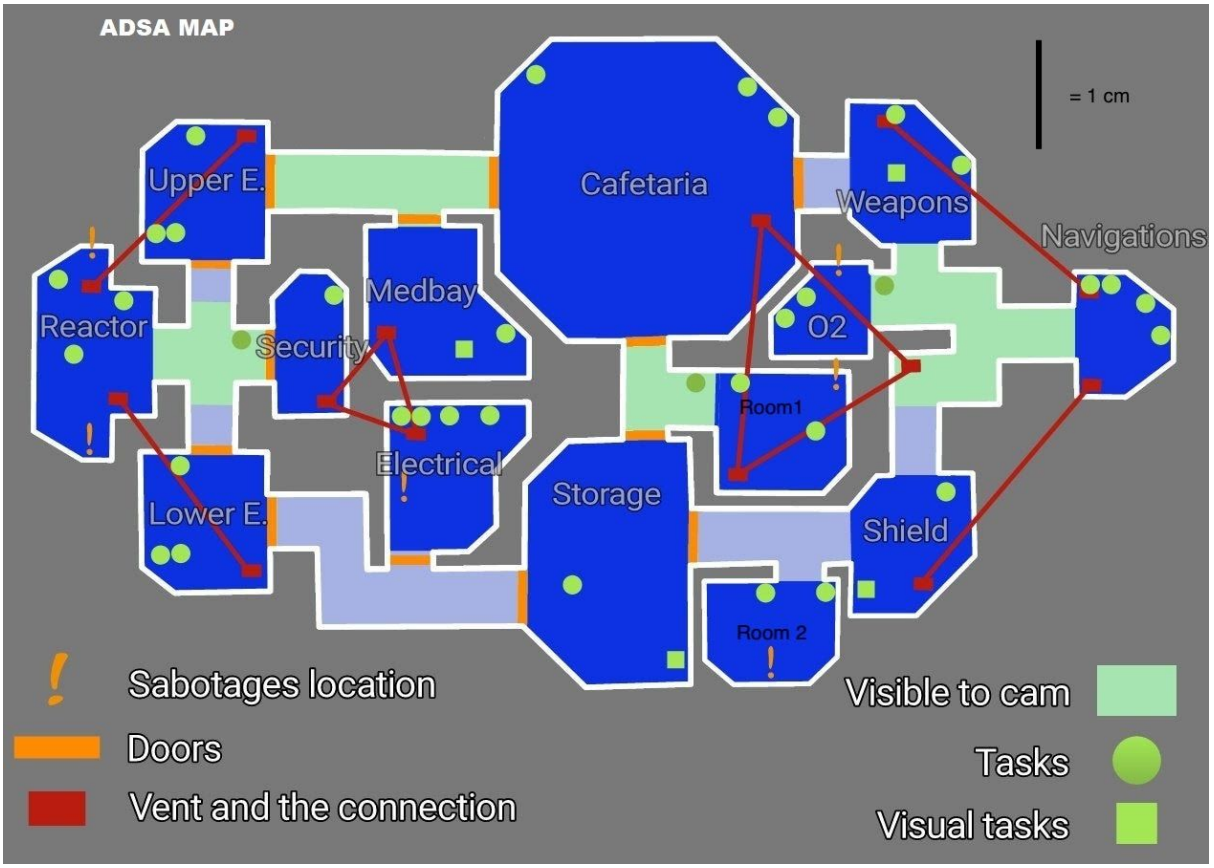
4. To implement this algorithm, we represent the graph of the relation “have seen” between the players as an adjacency matrix 10x10. Each line represents a player and the columns represent the relation there is with the player whose number is that of the column:
   - 1 if the two players saw each other;
   - 2 if not.

   We call this matrix **G.**
   The algorithm is named **potential_impostors.**
   The solution is:

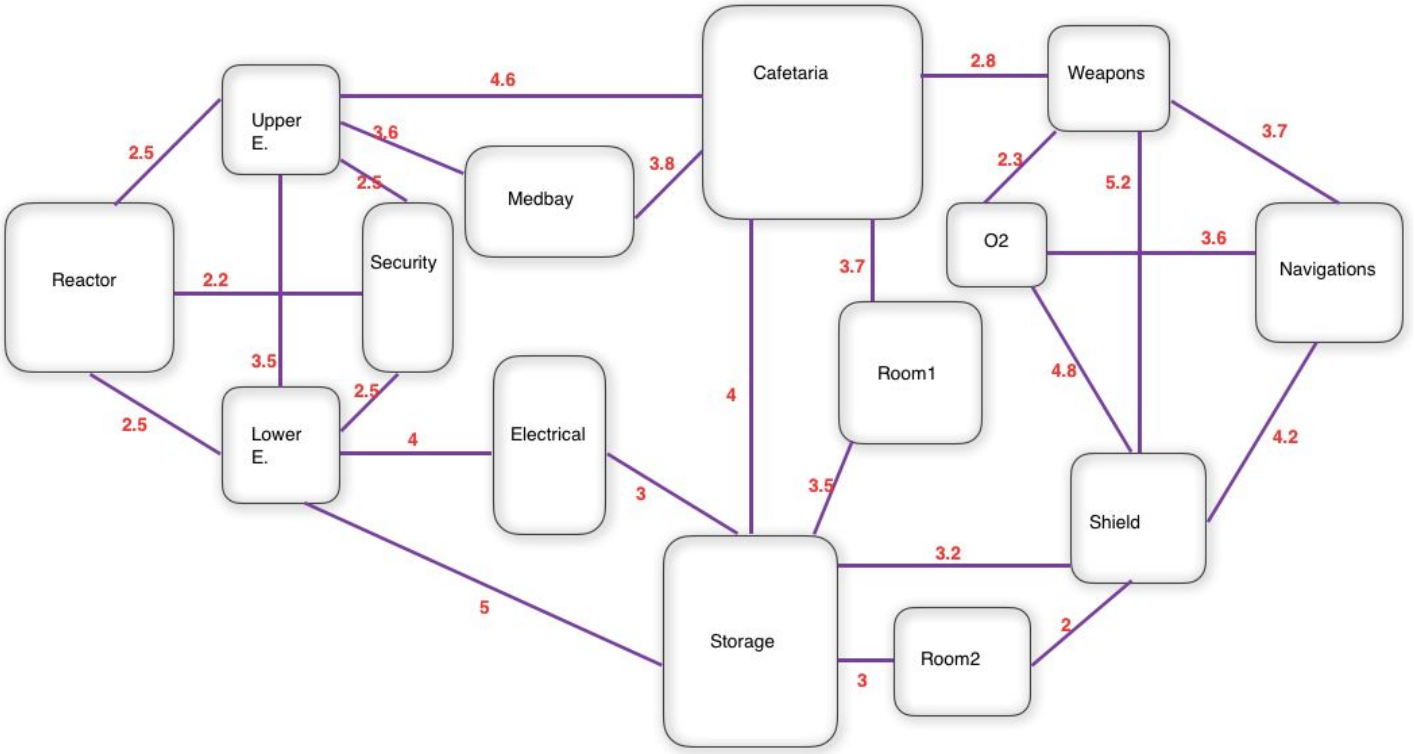
```
>>> potential_impostors(G)
The potential impostors if player 1 is impostor are : [3, 4, 5, 7, 8, 9]
The potential impostors if player 4 is impostor are : [1, 2, 5, 6, 7, 8]
The potential impostors if player 5 is impostor are : [1, 2, 3, 4, 6, 9]
```

STEP 3

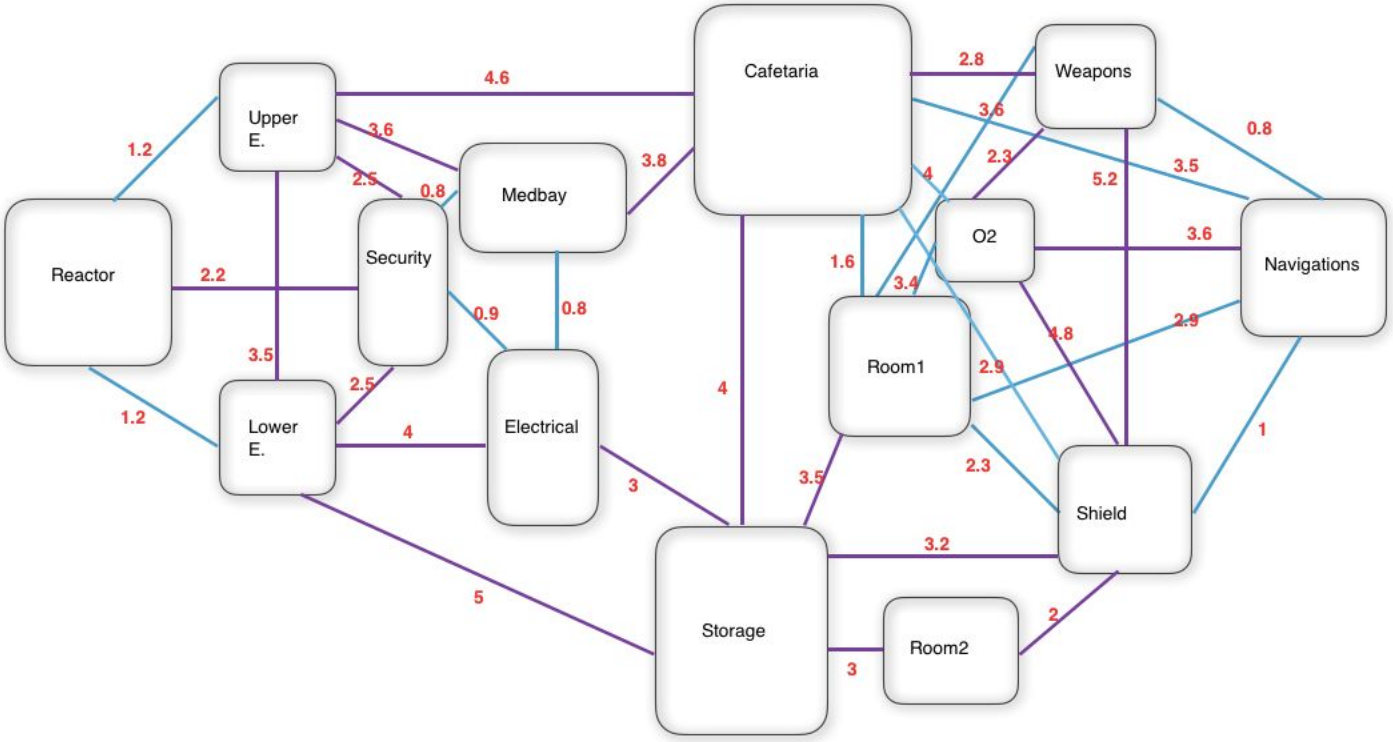
1. We will represent each of the models of the map (crewmates and impostors) by an undirected and weighted graph. Indeed, if we can go from point A to point B, then necessarily we can go from point B to point A.
   The weight of each edge represents the time in seconds that a player takes to move from one room to another.
   The statement tells us that players can move one centimeter per second. We therefore take a scale (which we have placed on the map) in order to be able to measure the distance between the adjacent rooms.
   There were two missing room names, so we renamed them "room 1" and "room 2".
   To measure the distance between two rooms we go from the middle of room A to the middle of room B.



The graph of the Crewmates :



The graph of the Impostors :



The paths that have been created or added are in blue on this graph.

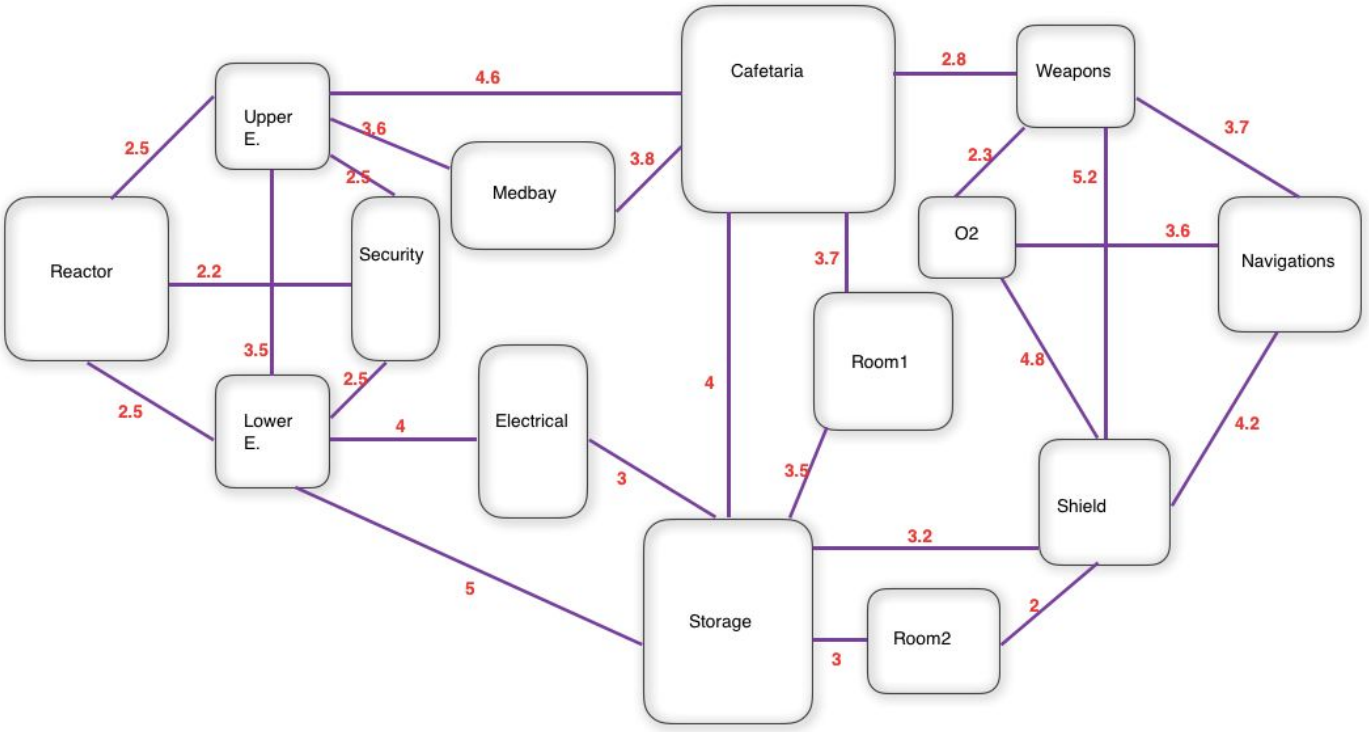
2. The goal is to find the shortest path for each pair of rooms.
   To do this, we use the floyd-warshall algorithm that we studied in class. This algorithm will give us for each pair of pieces the minimum time in seconds that players (be it an impostor or a crewmate, just the graph will change) will take to move between two pieces of the map.

3. We represent graphs as dataframes. The graph of crewmates is called "**df_Crewmate**" that of impostors "**df_Impostors**".
   To have the minimum time between each piece, whether it is for an impostor or a crewmate, we take the dataframe matrices and we run them in the algorithm we called "**floyd**". We then place them each in a dataframe: "**df_pathfindingImpostors**" and "**df_pathfindingCrewmates**".

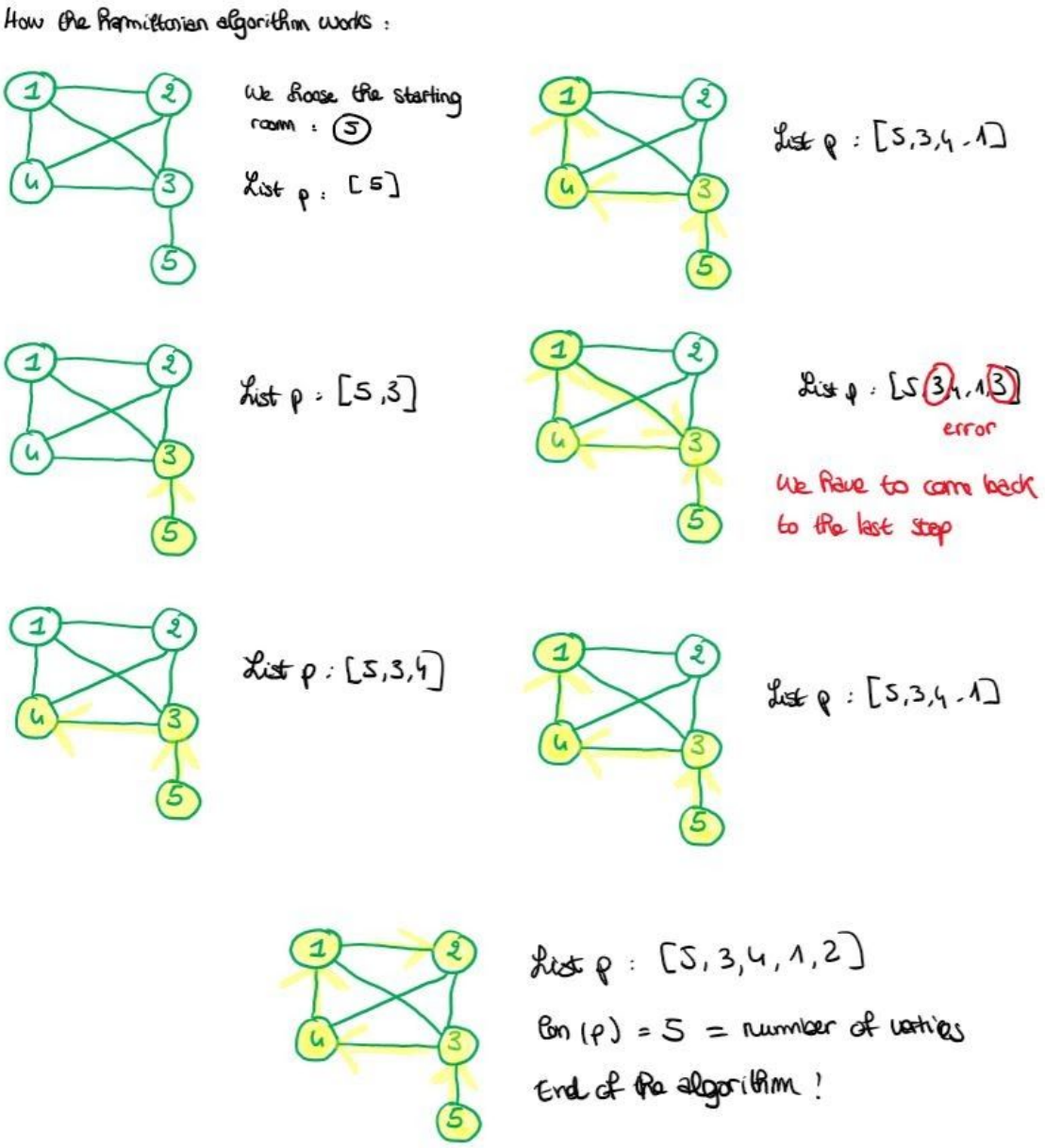
4. In order to unmask an impostor we can see the time it takes to move from one room to another. This is because since it has certain shortcuts, it may therefore take less time to go from one certain room to another. Thus, we subtract the time a crewmate takes to get from one room to another and the time the impostor takes. The "**Interval_time**" dataframe shows the lead time that the impostor will have over the crewmate.

STEP 4

1. We need to find the quickest path to go through all the rooms of the map only one time, starting from the room that we want.
   To do this, we are first modelling our map as an undirected and weighted graph, like in the STEP 3. Indeed, in this problem we have no restrictions about the direction of the path : for example, to first visit the cafeteria and then the weapons or to visit them in the other sens are both allowed.
   Since we want the quickest path, we weighted each edge by the time in seconds that a player takes to move from one room to another, knowing that players can move one centimeter per second. We measured this time exactly the same way as in the STEP 3, so we have exactly the same graph :



2. To find a route passing through each room only one time, we are referring to a Hamiltonian Path. Our goal here is to visit each room, represented as vertices, by using the corridors, represented as weighted edges, but we can visit each vertex only one time and we need to visit them all.



3. An algorithm solving this problem is, without surprise, the hamiltonian's algorithm to find the paths to browse all the room. This is a recursive algorithm that has 2 break conditions : either the path is not good, because it added a room that was already visited or the path is good and we have the 14 rooms visited one and only one time.

   The hamiltonian algorithm is working as following :

   Our function **hamliton_path** takes as argument Graph, the Crewmate matrix, p the list of the visited rooms and paths, the list of all the possible paths.
   At the beginning of our algorithm the list p contains only the current room, from where we want to start the path, and paths will contain all the possible paths starting from this exact room.

   First, we look at the last room in the visited room list, and find all its neighbors. For each neighbor, we add the neighbor to the visited room and we call the recursive

function with this new list p, jusqu'à ce que one of the two break conditions is reached.

To find the neighbors of a room, we created the **proches_voisins(Graph, current_room)** function.To know if a room is neighbor or not to our current room, it has to not be the current room and to not have a distance between the 2 rooms equal to the infinity.

To have all the hamiltonian paths for all the starting rooms, we created the **all_paths(Graph)** function that simply calls the **hamilton_path** function for each room as a start.
As a utility function we also created the **weight(Graph,path)** function that compiles the weight of the chosen path, by summing each vertex's weights of this path.

Regarding the statement, we want the shortest paths to browse only one time each room starting from whatever the room we chose.Thanks to our **all_paths** function, we obtain all the paths for all the rooms, but we want the minimum one for each room.
To do this, we did the **min_of_paths(graph)** function.
In short, we have the list of all the paths of all the rooms and another list for the weights corresponding to these paths.
We find the positions from where the starting room is changing, that allows us to see for each path starting from the same room the shortest one, by a usal algorithm comparing the weights of each path.
We return the list of shortest paths for each room with the corresponding weights.

4. When running the algorithm, we can notice that not all the rooms can have a hamiltonian path. We found the shortest hamiltonian path for the starting rooms : Medbay, Electrical, Room1, Room2, O2, Weapons and Navigations.

   For example, the shortest path starting by the room "Electrical", with the corresponding time (weight) will be display as follow :

```
['Electrical', 'Lower E', 'Reactor', 'Security', 'Upper E', 'Medbay', 'Cafeteria', 'Weapons', 'O2', 'Navigations', 'Shield', 'Room2', 'Storage', 'Room1']
Time needed :  40.0 seconds
```