# Step 1: To organize the tournament

### Propose a data structure to represent a Player and its Score

A player is characterized by its name/pseudo, its score, and its rank. The name is going to identify and differentiate players. The rank is going to classify them in order to make the groups and eliminate the last ones. The score is mandatory to update the rank. We have 3 attributes so the best data structure to store them in one object player is to create a class Player.

When we initialize a Player, we give him a name, his score is set to 0 and his rank depends on how many players were initialized before him even though at the beginning it doesn’t represent anything.

We actually took the time to create each one of the 100 players of the tournament. We picked the pseudo of the top 100 streamers in the world and wrote them in a text file participants.txt. Then with a dedicated function, we read this file and put every name in a list. After that we looped 100 times the Player class initialization (Player(name, rank=ith iteration)) to create our set of 100 Player objects.

We preferred this class structure to a simple dictionary (key:value)=(name:score) because we wanted to have this third attribute being the rank and also because we felt more comfortable using class. The data persistency of a class object, the possibility to add functions within the class related to the object (Update score, Reset Score, Update rank, etc.) and the ease of use and calling are all arguments that strengthened our choice.

### Propose a most optimized data structures for the tournament (called database in the following questions)

Now that we have our Player structure, we need a tournament structure to organize and classify them through different games. The main purpose of this database is to be able to reach any player with a log complexity. And a simple list of the 100 players is not going to match this constraint because if we are looking for a player that is at the end of this list, then it is going to take longer than a log complexity to reach it.

Since we want to classify players according to their rank, we have an integer value which can be compared to others. In this case, a Binary Search Tree reveals itself to be suited to the situation. When we have to insert data in the tree, at each node (if not the root), the inserted data has two options depending on the value of the rank: if the rank of the inserted player is higher than the rank of the player in the node, then it goes in the right subtree; left otherwise. And if there is no subtree, then the inserted player creates a new one.

With this database, searching for an element becomes easier because we always have only two choices: either go right if the value we are looking for is higher than the one in the actual node, or go left, until we reach the exact same value (if it exists).

However, BST is not the best suited structure. And this is because if by any mean the values we insert in the tree are always higher than the previously inserted value, then we are going to find ourselves with a tree not really looking like a tree because it only has one right subtree at each node. Therefore, if the value we are seeking in the tree is at the end of it (the last right subtree), then our tree will be no better than a classic list and we will reach a linear complexity. In conclusion, we need to balance our tree.

In this case, the AVL tree reveals itself to be the most suited structure for our tournament. Indeed, AVL tree is a balanced Binary Search Tree, meaning the insertion of data works exactly like a BST but it differentiates itself in the balancing. If the right subtree of a node gets 2 subtrees higher than the left, then we apply rotations to rebalance the tree.

With this technic, the AVL tree search complexity cannot exceed log2(n) which is our requirement.

Finally, we have to notice that Nodes of our tree are not just composed by a left subtree, a right subtree, a height (for AVL) and a simple value data. In our case, it is more complex because our data are Players which are structures. The main difference is going to play inside the AVL tree functions : instead of comparing data (players) directly, we will need to compare players.rank.

### Present and argue about a method that randomize player score at each game (between 0 point to 12 points)

In the Player Class, we implemented an UpdateScore method which, with a simple integer randomizer without weighted score, gives a score between 0 and 12. However, to keep the simulation as real as possible, we need to make scores like 10, 11 or 12 less probable than 4,5 or 6. We can use the technique of the dice increasing probability for average score and decreasing for extreme score (0,1,2,11,12). So here we have two “dices” containing numbers between 0 and 6 (included), hence a possible set of score in the range (0,12) (12 included).

Now, it is required that players score is not the sum of all scores from all their games but rather the mean of scores of all their games. Therefore, we cannot just add the randomized score from a game to the previous score, we have to multiply the previous score by (n-1)/n, n being the number of games played, and add to this the new score from this game divided by n. Then we have the mean score.

### Present and argue about a method to update Players score and the database

After updating Players score, we must update the database because the ranking of each player has really strong chances of changing. And to do so, we cannot keep our current AVL tree and change each player’s position in it, this would be overly complicated. If the 100th player scores 12 points, then we have to check if the player above him still has a higher score. If this is not the case, then player 100 exchanges places with player 99 and so on until player 100 finds a player that has a better score than his. And if you think this would be long, we must repeat this process as much times as there are players in the Tree. The conclusion is that it would take forever, and it is not a good solution to update players score within the tree.

However, if we just do an in-order search on the AVL tree, it will give us back the list of all players ordered by their ranking from the worst to the best. Then, we use this list to update players score simply looping iteratively and using UpdateScore function. Finally, with this list we create a new AVL tree which is going to class them upon their score and not their ranking (because it would not be updated at this time). Once all players are inserted inside the tree, another simple in-order search algorithm on the tree would give us the opportunity to rank them: the first one in the list being the 100th of the tournament and the last of the list the first.

It is important to have this subsidiary list alongside the AVL tree because it is this one that we are going to use to update scores and ranking of players.

### Present and argue about a method to create random games based on the database

The 3 first random games are warm up games. Their goal is to put players in the atmosphere of the tournament without any pressure of being kicked out at the end. To prepare them in the best way possible, they should not play again the exact same person 3 times in a row because this would not be representative of the average level: some groups of 10 might be more homogeneous than others, then giving the wrong impression to gamers (either the level is too high or too low). Hence the importance of well mixing the players in those 3 games.

For the first random game, all players score is set to 0 so we can’t know their level. Plus, nobody played nobody so far so we can create games of 10 players however we want, there would not be redundancy. Then it is sold, players will play against each other depending on their ranking or more precisely the alphabetical order because this is how their first ranking was based on. And to get players in a ranking order from the AVL Tree, we have the in-order search algorithm.

Now for the second random game, scores are updated, so are ranks and the new AVL Tree is built. We want an algorithm that is going to give us back the list of players from this tree but in a different order from the previous one. And even though doing an in-order search algorithm would not give us the same list (in the same order) as earlier, this would not respect our wish of heterogeneity among the random games. To keep the splitting as random as possible we will use the fact that the ranking changes of course but also and mainly use a different search algorithm: pre-order or post-order. Here we use the pre-order search algorithm to organize the second random game.

For the last game, we repeat the same steps as earlier but apply the post-order search algorithm instead.

Even though we didn’t really need scores and ranking for the creation of the 3 random games, we still needed to update them because they are mandatory for the rest of the tournament. Indeed, at the end of the 3 random games, the 100 players will be ranked according to their score so that the tournament and the disqualification games can begin. (the score of each player is not reset after the 3 games, it wouldn’t be fair for top players).

### Present and argue about a method to create games based on ranking

For the creation and repartition of players according to their rank, we had to make a choice:

* Rewarding the best players by making them play against the worst players
* Giving all players, good or bad, the same chance of success batching the best players together and the worst players together

The second option is simple: we get the list of players in-order from the AVL tree then split the list every ten players so that player ranked 1 play against 2, 3, 4 until 10, and players 91 to 100 play against. Unfortunately, this technic does not reward the best players, it evens disadvantages them. The 10th player for example will be the worst player from his group and will not get the chance to score as many points as he could have if he had played against worse players.

However, since we are giving points randomly to each player and we do not respect some type of structured pointing system, this logic of rewarding gets blown away. Therefore, we decided to go for the simplest way of splitting groups from 10 to 10 rank-wisely. The process is quite straight forward: get the list of rank-ordered players with the in-order algorithm, go through the list with a for loop and place players in a group (list), changing list every 10 players.

### Present and argue about a method to drop the players and to play game until the last 10 players

Serious things begin. Shuffle games are over, and the official tournament takes place: 9 rounds of one game after each the 10 last players (least score) are permanently eliminated. 9 rounds because we need to eliminate 90 players to keep the 10 best for the final step of the tournament.

The scheme is the following: for the first round, we have our AVL Tree and our affiliated rank-ordered list.

* we update the scores (with the list) after the game
* we create the new AVL Tree
* we get the list of players ordered from the lowest score to the highest
* we update the ranks
* we delete the 10 first players of the list (10 worst)

So, we repeat this action 9 times.

The reason we do not practice a deletion directly on the AVL Tree is because when we create the AVL tree after the score update, we do not know the new rank of players. So, we cannot go through the AVL tree searching for the 10 worst players according to their ranking because it will not be up to date and we could find ourselves to eliminate the 89th player because his rank was 91 before the game. Furthermore, we cannot do the same thing with the score because we cannot know the exact score for which the 10 last players are below.

Hence the need to get the new list of ordered players (in-order algorithm) from the AVL Tree to update the ranks and delete the 10 worst players from it. At this point, we could question the utility of the AVL Tree since we don’t practice any action on it like deletion. But in fact, the AVL Tree is super effective for classifying players and getting the list of players in order quickly.

### Present and argue about a method which display the TOP10 players and the podium after the final game.

When only the 10 best players are remaining, the tournament organization changes. All scores are reset to 0 to not disadvantage the 10th player who has a lower score than the 1st. Players are going to fight for the podium around five games. After each game, scores are always updated, but ranks and also the AVL tree don’t need to be because we don’t attach any importance to the classification of players until the fifth and last game is played since there are no disqualifications.

Therefore, after the final game, with the list of 10 players, we create our AVL Tree inserting players according to their score and not to their not updated rank. Then we get the same list but ordered from the worst to the best with the in-order algorithm. Finally, we loop through the list starting from the end and display the rank, score and name of players from the best (1st) to the worst (10th).

# Step 2: Professor Layton < Guybrush Threepwood < Us

### Represent the relation (have seen) between players as a graph, argue about your model.

Use graph and representation matrix of a graph (CMO2 p25). Matrix of Boolean, n\*n with n the number of players. If the first player has seen the players 2, 4 and 5, then the value of the matrix is set to 1, same for and .

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 |  | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 |  |  | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 2 |  |  |  | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 3 |  |  |  |  | 1 | 0 | 0 | 0 | 1 | 0 |
| 4 |  |  |  |  |  | 0 | 0 | 0 | 0 | 1 |
| 5 |  |  |  |  |  |  | 0 | 1 | 1 | 0 |
| 6 |  |  |  |  |  |  |  | 0 | 1 | 1 |
| 7 |  |  |  |  |  |  |  |  | 0 | 1 |
| 8 |  |  |  |  |  |  |  |  |  | 0 |
| 9 |  |  |  |  |  |  |  |  |  |  |

### Thanks to a **graph theory problem**, present how to find a set of probable impostors.

Logically, if a crewmate dies, then the impostor has to be one of the members he has previously seen. Therefore, if 0 is found dead, then the impostor is either 1, 4 or 5. With this probable set of impostors we can estimate who is going to be the second impostor. Indeed, if for example, 1 is the impostor, and we know impostors don’t see each other at all, the set of probable impostors is all the players 1 didn’t see : 3, 4, 5, 7, 8 and 9. In our case, it isn’t really helpful, but in a common game players ought to see more than 3 other players increasing our chances of finding the impostors with this technic.

Furthermore, we can’t make assumptions after the first kill. A better and more efficient way would be to store this information and then add it to the new information we get after the second kill. To do so, we can create a coefficient of Impostorness that increases of a certain amount depending on the number of other players the dead one saw. To take back our example, if 0 dies, then 1, 4 and 5 see their coefficient increased by 1 divided by 3. After the second kill, if 9 dies and he saw 1 and 4 then those players see their coefficient increased by 0.5. At this point, 1 and 4 both have coefficient equal to 0.83. They can both be impostors or only one of them. There are only 6 crewmates remaining for 2 impostors, if impostors manage to kill two more players, then the game is finished.

To avoid this situation, we need to bust one impostor before this situation happens. And we just saw that the Impostorness coefficient is not enough.

We also know impostors never walk together which means they never see. So, what we can do is saving in a global matrix who saw who after each kill. We can simply sum up all Boolean matrices.

Following these steps, if we arrive at a point where a suspected player has seen every other player, except one, then we can highly deduce that this player and the one he has never seen are the impostors. Reciprocally, if a player is a suspect with a high Impostorness coefficient but has seen every other player alive, then he cannot be an impostor.

This technic combined with the Impostorness coefficient can get us to find impostors way quicker.

Let’s get back to our example. 0 was killed and he saw 1, 4 and 5:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | IC | sum |
| 0 |  | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 |  |  | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1/3 | 1/3 |
| 2 |  |  |  | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 3 |  |  |  |  | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 4 |  |  |  |  |  | 0 | 0 | 0 | 0 | 1 | 1/3 | 1/3 |
| 5 |  |  |  |  |  |  | 0 | 1 | 1 | 0 | 1/3 | 1/3 |
| 6 |  |  |  |  |  |  |  | 0 | 1 | 1 | 0 | 0 |
| 7 |  |  |  |  |  |  |  |  | 0 | 1 | 0 | 0 |
| 8 |  |  |  |  |  |  |  |  |  | 0 | 0 | 0 |
| 9 |  |  |  |  |  |  |  |  |  |  | 0 | 0 |

After the first kill, we cannot conclude on who are the impostors.

4 was killed and he saw 1 and 7.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | IC | Sum |
| 0 |  |  |  |  |  |  |  |  |  |  |  |  |
| 1 |  |  | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1/2 | 5/6 |
| 2 |  |  |  | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| 3 |  |  |  |  | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 4 |  |  |  |  |  | 0 | 0 | 1 | 0 | 1 | 0 | 1/3 |
| 5 |  |  |  |  |  |  | 0 | 0 | 1 | 0 | 0 | 1/3 |
| 6 |  |  |  |  |  |  |  | 0 | 1 | 1 | 0 | 0 |
| 7 |  |  |  |  |  |  |  |  | 1 | 1 | 1/2 | 1/2 |
| 8 |  |  |  |  |  |  |  |  |  | 0 | 0 | 0 |
| 9 |  |  |  |  |  |  |  |  |  |  | 0 | 0 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 0 |  | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 |  |  | 2 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 2 |  |  |  | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 3 |  |  |  |  | 1 | 0 | 1 | 0 | 1 | 0 |
| 4 |  |  |  |  |  | 0 | 0 | 1 | 1 | 1 |
| 5 |  |  |  |  |  |  | 0 | 0 | 1 | 0 |
| 6 |  |  |  |  |  |  |  | 0 | 2 | 1 |
| 7 |  |  |  |  |  |  |  |  | 1 | 1 |
| 8 |  |  |  |  |  |  |  |  |  | 0 |
| 9 |  |  |  |  |  |  |  |  |  |  |

Here, the dead crewmate only saw 2 other players, so the probability one of them is the impostor is high (0.5). We add those probabilities to the sum of coefficients and get 5/6 for 1 and ½ for 7 meaning 1 has more chances to be the impostor. Yet, we can’t deduce with certainty that 1 is the impostor. But if we could prove 7 is innocent, then we would be sure 1 is an impostor. And one way to do this is to verify if 7 has seen all other players. If that is the case, then he can’t be an impostor since two impostors cannot see each other. This technic only works when two impostors are remaining.

Furthermore, when an impostor is unmasked, then we have to update all previously calculated Impostorness coefficient. Indeed, if 1 is unmasked, then we are sure all players he saw during the game are not impostors. If those exonerated had a Impostorness coefficient different from 0, then we have to redistribute their value to those not innocent.

If we look at the matrix of the sum of meetings, we see that 8 saw every one except 9 and 1 who is a serious suspect. Therefore if 1 is proven to be an impostor then its Impostorness coefficient would be transferred and split between all players he didn’t see and if it remains only one player then he has to be the impostor.

### Argue about an algorithm solving your problem.

Busted\_impostor=0

While (Busted\_impostor!=2)

If (crewmate gets killed)

New\_matrix= Get\_Matrix\_meetings()

We get the list of players the killed player saw

List\_Suspects=Suspects(New\_Matrix)

If this list contains only one player, then he has to be an impostor

If (lengthList\_Suspects == 1)

Impostor=List\_Suspects[0]

Busted\_Impostor+=1

We update Impostorness coefficients

Update\_Impostorness\_Coeff(New\_Matrix\_With\_Dead)

Sum\_matrix

### Implement the algorithm and show a solution.

Implement the previous algorithm

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

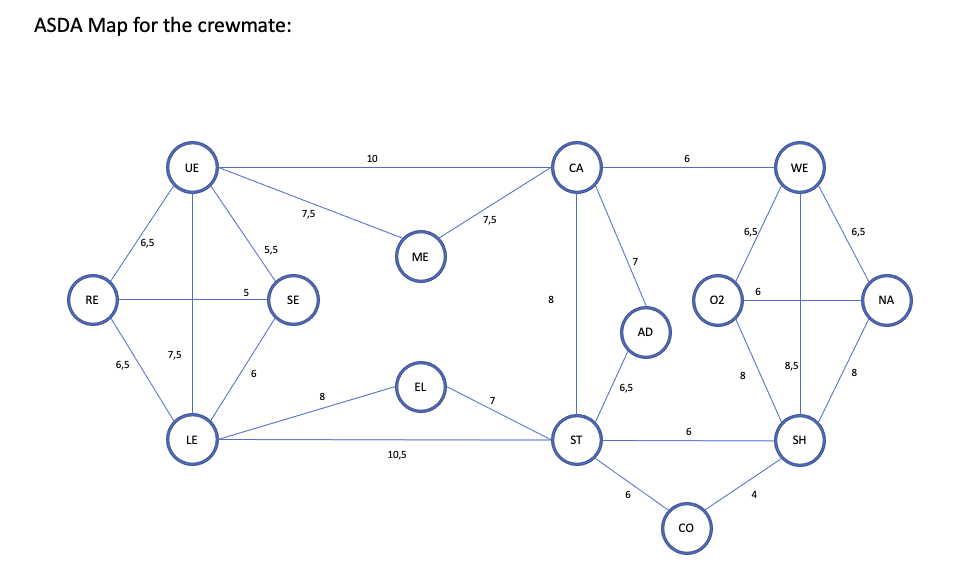
### Presents and argue about the two models of the map.

To represent the map, we are going to use undirected graph for both models.

**First model: crewmate without vents**

In the first model, players can go in each room in both ways (from B to A and A to B), each way (aka edges) between rooms (aka nodes) is weighted by the amount of time needed to go from one room to another. The weight is going to be the same both way, and anyone can travel from a room: to another both way that is why we use a undirected graph.

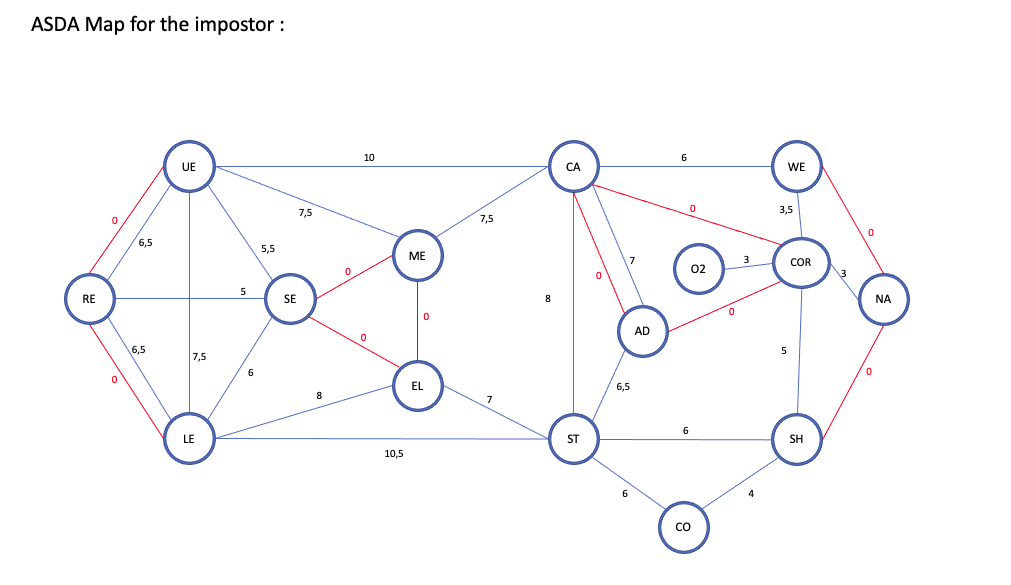
The weight is going to be in second. To calculate weights, we measured the distance between rooms (1cm on the map equals 1 sec in the graph)



**Second model: impostor with vents**

This graph is a subgraph of the previous one, meaning we are going to take the same one and add edges that represent vents. Vents are new edges that impostor can take to travel faster through rooms and because the time to travel in vents is null, the weights on these new edges are all 0.

In fact, we are also going to first delete all the edges that connect the room “WE”, “O2”, “NA” and “SH” because we are going to add a new room “COR” for corridor. We need to create this new room because one vent leads to no room but just a corridor. So, once we delete the edges and add the new room, we add new edges from this new room with cost that respect and correspond to the crewmate map. With this method, we add a new room (“COR”) to allows a new vent in respect of the previous map so the cost for all path stay the same even with this new ‘fake room’.



We can see that on the impostor map we indeed added a ‘fake room’ which is ‘COR’ for corridor. We added this room because the impostor has a vent which leads to a corridor and not a room, so we needed to add this ‘fake room’ so that the impostor map respect as much as possible the original map. This ‘fake room’ will not be counted in the list of rooms to do our research to unmask the impostor, it is only there as a path for the vent to respect the original map. This ‘fake room’ doesn’t change the weight of the path between the real rooms, which is very important for the rest of the project.

**How to unmask the impostor?**

To unmask who is the impostor, we need to calculate the time a player takes to go from one room to another. If this time is less than the time a crewmate is supposed to have in the best case, then the player is an impostor.

### Argue about a **pathfinding algorithm** to implement.

The goal here is to find a pathfinding algorithm that best fit the model with the minimum complexity. Since both our graph doesn’t contain negative weights, we don’t need to use the Floyd-Warshall and Bellman-Ford algorithm, and since we have more edges than nodes it is better to use the Dijkstra algorithm for our problem. We need to take the shortest path between two room so that we can compare them between an impostor and a crewmate to find the impostor.

Dijkstra’s algorithm allows us to have the minimum cost from an initial node to all the nodes of the map. The goal now is to implement this method

### Implement the method and show the time to travel for any pair of rooms for both models.

**Method:**

The goal here is to implement the Dijkstra algorithm and show the result.

They are 5 different files in this part for 5 different functions: GraphClass, Create\_map\_function, Q2\_Dijkstra, Q3\_Show\_result and the main.

**GraphClass:**

This file is here to create a class of graph. This class will allow us to create a graph and have access to its nodes and edges. This class doesn’t take any parameter, but it has 2 variables:

* Dico\_nodes : which is a dictionary that takes the name of the node as a key and the value of the node as a value (the value of the node will be use in the Dijkstra algorithm and is initialized to zero).
* List\_edges : which is a list composed of every edges that compose the graph. Each element of this list is an edge store in this form: (start node, end node, cost).

To add node and edge to our graph we add two function to this class:

* Add\_node : to add a node to dico\_node of a graph, we add a node with the name of the node
* Add\_edge : to add an edge list\_edge of a graph, we add an edge with its start node, end node and its cost (weight)

This file allows us to create an instance of the crew mate graph and one for the impostor. We will use those graphs for the Dijkstra’s algorithm.

**Create\_map\_function**

This file allows us to create the 2 different maps (impostor and crewmate) from the GraphClass. To do so, we use the 2 function from the graph class to add node and edges to a graph. We created these 2 graphs in respect of the 2 diagram showed below.

**Q2\_dijkstra**

In this file we have the main function of all this part: the Dijkstra algorithm. Dijkstra function takes 2 parameters: a graph and an initial node. This function will return a list of all the node of the graph with their minimum cost to travel from the initial node. For more details on how this function works, please take a look at the comment of the python’s program that explain clearly how it works.

**Q3\_show\_result**

This file is here to present the time to travel for any pair of room for both models. It is composed of 2 main functions:

* All\_time : take a graph for parameter and return the list of all the pair of room possible with their cost to travel from one to another. We use the Dijkstra function here to calculate all the different cost.
* Print\_the\_reslut : allows us to take the list of result from the previous function and show the result in a good way.

**Main**

In this file we can test all the previous functions:

* Question 2: to show the Dijkstra algorithm result.
* Question 3: to display the result of the Dijkstra algorithm for any pair of room for both graphs.

### Display the interval of time for each pair of room where the traveler is an impostor.

For this part we add another file: Q4\_Compare.

**Q4\_compare**

The goal here is to display the difference of time to travel between any pair of room between a crew mate and an impostor. Here we use the function from the previous python file for both model and display the difference.

We also add this function to the main file so we can show our result with the function Question 4.

# Step 4: Secure the last tasks

### Presents and argue about the model of the map.

We are going to use the same graph than in the previous part, but only the crew mate map since only crew mate has mission to do.

Then we are going to store all the last node to visit in a list.

We only need to use the Hamiltonian path and not the Hamiltonian cycle since we don’t need to go back to the original node to finish the different task.

Since we always ‘re start’ a game after an emergency call at the Cafeteria, we will take the room “CA” as the start node for every Hamiltonian path.

The goal is also to find the shortest path to finish all the task.

### Thanks to a **graph theory problem**, present how to find a route passing through each room only one time.

We are going to use an algorithm of the traveling salesman problem. Finding the shortest or fastest way/itinerary between multiple tasks remaining.

To do so, we use the Hamiltonian path (cf CMO3) : undirected graph, each place must be visited once and only once (visit every vertex/node exactly once)

### Argue about an algorithm solving your problem.

Explain functioning of Hamiltonian algorithm

### Implement the algorithm and show a solution.

Implement previous algorithm with python.