

Advanced Data Structure and Algorithms Mini-Problem

STEP 1: To organize the tournament

QUESTION 1: Propose a data structure to represent a Player and its Score.

We need to store for each Player its Score. To do so, we are going to define a class called Player, and set different attributes such as the Player's Name, his ID followed by his Score. We will also implement the function `__str__` in the class in order to have a personalized display of the Player. Here is an example:

```
Id=1    Score=0    Name=John
```

QUESTION 2: Propose a most optimized data structure for the tournament (called database in the following questions).

The most optimized data structure for the tournament would be Binary Search Trees, specifically AVL Trees. Compared to Binary Search Trees, AVL Trees are balanced, that is to say the difference between the heights of the left and right subtrees of any node is at most 1. The players are stored in a structured database with a log complexity to reach an element. Since the tree is balanced, all the basic operations insertion, deletion, and search can be achieved in a $\log(N)$ time complexity where N is the number of players stored. Thus, our tree will contain nodes where each node, in our case each player, is an instance of the class Player.

We created a class Tournament with the functions typical of an AVL tree class: *get_height*, *balance_factor*, *left_rotate*, *right_rotate*, *insert*, *delete*... And also an *inorder_print* function in order to print the players of our tournament.

At the beginning of the tournament, all the scores are equal to 0.

Here is an example of a tournament after inserting 10 Players:

```
The players of our tournament are :
Id=1    Score=0    Name=John
Id=2    Score=0    Name=anonymous34
Id=3    Score=0    Name=RedKing
Id=4    Score=0    Name=banana18
Id=5    Score=0    Name=momo98
Id=6    Score=0    Name=dontkillmeplease
Id=7    Score=0    Name=distractioin
Id=8    Score=0    Name=bobby15
Id=9    Score=0    Name=blueparth
Id=10   Score=0    Name=OrangeQueen
```

QUESTION 3: Present and argue about a method that randomize player score at each game (between 0 point to 12 points).

We want a method that sets the Players's Score to an integer between 0 and 12 at the beginning of each game. In order to do so, we can use the random package available in Python.

QUESTION 4: Present and argue about a method to update Players score and the database.

We now desire to implement a method that updates Players score and the database. To update the database, we need to know which Player is affected and what is his latest score. Thus, the method will take the Player ID and its newly score. Using these inputs, there are several conditions to check. First of all, we need to check if the Player is new, that is to say, if the Player ID is not in our database. If it is the case, we will display an appropriate message: "This player doesn't play in this competition". Otherwise, if the Player already exists, we need to access its score. Then we are going to add the new score to the actual score. To add a new score, we will use the function defined in the previous question. To summarize, the function *update_database* will take the Player ID and call the random function.

For example, we will use the tournament defined in question 2 and try to update the scores of Player 1 and Player 12.

After updating the score of Player 1 we have:

Id=1	Score=9	Name=John
Id=2	Score=0	Name=anonymous34
Id=3	Score=0	Name=RedKing
Id=4	Score=0	Name=banana18
Id=5	Score=0	Name=momo98
Id=6	Score=0	Name=dontkillmeplease
Id=7	Score=0	Name=distraction
Id=8	Score=0	Name=bobby15
Id=9	Score=0	Name=blueparth
Id=10	Score=0	Name=OrangeQueen

John had a score of 0 so if we update its score, he will have a random score between 0 and 12: here it is 9. If we update again its score, John will have a score between 9 (9+0) and 21 (9+12). We obtain:

Id=1	Score=11	Name=John
Id=2	Score=0	Name=anonymous34
Id=3	Score=0	Name=RedKing
Id=4	Score=0	Name=banana18
Id=5	Score=0	Name=momo98
Id=6	Score=0	Name=dontkillmeplease
Id=7	Score=0	Name=distraction
Id=8	Score=0	Name=bobby15
Id=9	Score=0	Name=blueparth
Id=10	Score=0	Name=OrangeQueen

Now if we want to update the score of Player 12, which is not yet in our tournament, we have:

This player doesn't play in this competition

QUESTION 5: Present and argue about a method to create random games based on the database.

First of all, we are going to define a function *inorder_fill_table* that takes an AVL tree and returns an array containing the elements of the tree. This function recursively appends nodes to a list. Then we are going to call this function while passing our database. Once we have the list, we will apply the function *shuffle* from the *random* package in order to have a random distribution of the Players. Since each game is composed of 10 Players, we will create different lists of 10 Players from the list previously built.

We will add 20 new players to our tournament in order to test this method (which we called *create_games_database*).

This time, we give our players a non-null score because we want to check if our function creates games randomly (and based on ranking for Question 6).

So our tournament is now composed of 30 players and our function should build randomly 3 teams of 10 Players. We obtain:

```
*** We create random teams ***

---The players of the team are :
Id=22   Score=2   Name=Megga
Id=15   Score=10  Name=impostor14
Id=5    Score=0   Name=momo98
Id=27   Score=5   Name=EpikWhale
Id=7    Score=0   Name=distractio
Id=16   Score=5   Name=invisibleh24
Id=2    Score=0   Name=anonymous34
Id=11   Score=4   Name=professorlayton
Id=19   Score=15  Name=Rojo
Id=10   Score=0   Name=OrangeQueen

---The players of the team are :
Id=13   Score=11  Name=ihavecovid19
Id=1    Score=11  Name=John
Id=28   Score=6   Name=Zexrow
Id=25   Score=8   Name=Crue
Id=12   Score=3   Name=darkxor
Id=24   Score=13  Name=Fatch
Id=21   Score=5   Name=Falconer
Id=14   Score=2   Name=rainbow84
Id=29   Score=4   Name=Bugha
Id=4    Score=0   Name=banana18

---The players of the team are :
Id=30   Score=3   Name=Vato
Id=20   Score=12  Name=Wolfiez
Id=9    Score=0   Name=blueparth
Id=18   Score=2   Name=Aqua
Id=17   Score=3   Name=Nyhrox
Id=26   Score=7   Name=Kreo
Id=8    Score=0   Name=bobby15
Id=23   Score=10  Name=Skite
Id=3    Score=0   Name=RedKing
Id=6    Score=0   Name=dontkillmeplease
```

QUESTION 6: Present and argue about a method to create games based on ranking.

This method is similar to the previous one, although there are some differences. First of all, we will call the function that returns a list from the tree. Basically, we have a list containing the Players along their attributes. Since we want to create games based on ranking, we will sort the list by the Players scores. From this list, we will again create different lists of 10 Players.

*** We create teams based on ranking ***

---The players of the team are :

Id=19	Score=15	Name=Rojo
Id=24	Score=13	Name=Fatch
Id=20	Score=12	Name=Wolfiez
Id=1	Score=11	Name=John
Id=13	Score=11	Name=ihavecovid19
Id=15	Score=10	Name=impostor14
Id=23	Score=10	Name=Skite
Id=25	Score=8	Name=Crue
Id=26	Score=7	Name=Kreo
Id=28	Score=6	Name=Zexrow

---The players of the team are :

Id=16	Score=5	Name=invisibleh24
Id=21	Score=5	Name=Falconer
Id=27	Score=5	Name=EpikWhale
Id=11	Score=4	Name=professorlayton
Id=29	Score=4	Name=Bugha
Id=12	Score=3	Name=darkxor
Id=17	Score=3	Name=Nyhrox
Id=30	Score=3	Name=Vato
Id=14	Score=2	Name=rainbow84
Id=18	Score=2	Name=Aqua

---The players of the team are :

Id=22	Score=2	Name=Megga
Id=2	Score=0	Name=anonymous34
Id=3	Score=0	Name=RedKing
Id=4	Score=0	Name=banana18
Id=5	Score=0	Name=momo98
Id=6	Score=0	Name=dontkillmeplease
Id=7	Score=0	Name=distraction
Id=8	Score=0	Name=bobby15
Id=9	Score=0	Name=blueparth
Id=10	Score=0	Name=OrangeQueen

QUESTION 7: Present and argue about a method to drop the players and to play game until the last 10 players.

Like we did in the previous questions, we are going to get the list from the tournament. Then, we will loop until the length of the list becomes equal (or inferior) to 10, since we need to play until the 10 last players. In the loop, we will create teams based on ranking using the appropriate function (created in Question 6). There are 10 Players per game and 3 random games then. So for each Player in our teams, we are going to call the method *update_database* 3 times. Since the method *update_database* calls the function random, our scores will be updated as if there were games being played.

Because our scores have changed, the tree itself has changed, so we need to get the actual list from the actual tournament. After that, we will sort the list built score wise and delete the 10 first elements since scores are sorted in increasing order. We will repeat this process until there are 10 players and print the 10 last players.

The id of the players still in the tournament are : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]

The leaderboard after playing is

Id=14	Score=8	Name=rainbow84
Id=7	Score=9	Name=distractation
Id=29	Score=9	Name=Bugha
Id=22	Score=10	Name=Megga
Id=9	Score=12	Name=blueparth
Id=4	Score=14	Name=banana18
Id=5	Score=16	Name=momo98
Id=6	Score=19	Name=dontkillmeplease
Id=12	Score=19	Name=darkxor
Id=18	Score=19	Name=Aqua
Id=2	Score=20	Name=anonymous34
Id=16	Score=20	Name=invisibleh24
Id=21	Score=20	Name=Falconer
Id=26	Score=21	Name=Kreo
Id=25	Score=22	Name=Crue
Id=3	Score=23	Name=RedKing
Id=11	Score=23	Name=professorlayton
Id=28	Score=24	Name=Zexrow
Id=30	Score=25	Name=Vato
Id=27	Score=27	Name=EpikWhale
Id=8	Score=28	Name=bobby15
Id=17	Score=29	Name=Nyhrox
Id=10	Score=30	Name=OrangeQueen
Id=1	Score=31	Name=John
Id=23	Score=31	Name=Skite
Id=15	Score=36	Name=impostor14
Id=24	Score=36	Name=Fatch
Id=20	Score=38	Name=Wolfiez
Id=13	Score=39	Name=ihavecovid19
Id=19	Score=47	Name=Rojo

The following players are going to be ejected from the tournament :

Id=14	Score=8	Name=rainbow84
Id=7	Score=9	Name=distractation
Id=29	Score=9	Name=Bugha
Id=22	Score=10	Name=Megga
Id=9	Score=12	Name=blueparth
Id=4	Score=14	Name=banana18
Id=5	Score=16	Name=momo98
Id=6	Score=19	Name=dontkillmeplease
Id=12	Score=19	Name=darkxor
Id=18	Score=19	Name=Aqua

The id of the players still in the tournament are : [1, 2, 3, 8, 10, 11, 13, 15, 16, 17, 19, 20, 21, 23, 24, 25, 26, 27, 28, 30]

The leaderboard after playing is

Id=23	Score=23	Name=Megga
Id=19	Score=29	Name=Aqua
Id=15	Score=31	Name=rainbow84

Id=26	Score=33	Name=Kreo
Id=21	Score=34	Name=Falconer
Id=3	Score=37	Name=RedKing
Id=16	Score=37	Name=invisibleh24
Id=13	Score=40	Name=darkxor
Id=1	Score=43	Name=John
Id=2	Score=44	Name=anonymous34
Id=28	Score=46	Name=Zexrow
Id=24	Score=47	Name=Fatch
Id=30	Score=48	Name=Vato
Id=17	Score=49	Name=Nyhrox
Id=25	Score=50	Name=Crue
Id=8	Score=51	Name=bobby15
Id=27	Score=52	Name=EpikWhale
Id=10	Score=53	Name=OrangeQueen
Id=11	Score=54	Name=professorlayton
Id=20	Score=59	Name=Wolfiez

The following players are going to be ejected from the tournament :

Id=23	Score=23	Name=Megga
Id=19	Score=29	Name=Aqua
Id=15	Score=31	Name=rainbow84
Id=26	Score=33	Name=Kreo
Id=21	Score=34	Name=Falconer
Id=3	Score=37	Name=RedKing
Id=16	Score=37	Name=invisibleh24
Id=13	Score=40	Name=darkxor
Id=1	Score=43	Name=John
Id=2	Score=44	Name=anonymous34

The 10 final players are :

Id=8	Score=51	Name=bobby15
Id=10	Score=53	Name=OrangeQueen
Id=11	Score=54	Name=professorlayton
Id=17	Score=37	Name=invisibleh24
Id=20	Score=59	Name=Wolfiez
Id=24	Score=47	Name=Fatch
Id=25	Score=50	Name=Crue
Id=27	Score=33	Name=Kreo
Id=28	Score=46	Name=Zexrow
Id=30	Score=48	Name=Vato

QUESTION 8: Present and argue about a method which displays the TOP10 players and the podium after the final game.

For the last 10 players, we need to play 5 games with reinitiated ranking. In the first place, we are going to create a recursive function *reinitialize* that assigns each player's score to 0 given a tree. We will call this function with our tournament. Then, we get the array from the tree by calling the appropriate function. From this, we can print the Players before playing, then for each Player in our list, we are going to call the method *update_database* 5 times. From here, we can print the Players after playing. Since we need to have a correct display, we will sort the list score wise in decreasing order. Finally, we print the TOP 10 Players and the 3 first Players for the Podium.

Here is the result obtained with the 10 Players left in the previous question:

As we reinitialized all scores, the leaderboard before playing the final game is :

Id=8	Score=0	Name=bobby15
Id=10	Score=0	Name=OrangeQueen
Id=11	Score=0	Name=professorlayton
Id=17	Score=0	Name=invisibleh24
Id=20	Score=0	Name=Wolfiez
Id=24	Score=0	Name=Fatch
Id=25	Score=0	Name=Crue
Id=27	Score=0	Name=Kreo
Id=28	Score=0	Name=Zexrow
Id=30	Score=0	Name=Vato

The leaderboard after playing 5 games :

Id=8	Score=25	Name=bobby15
Id=10	Score=36	Name=OrangeQueen
Id=11	Score=31	Name=professorlayton
Id=17	Score=22	Name=invisibleh24
Id=20	Score=32	Name=Wolfiez
Id=24	Score=21	Name=Fatch
Id=25	Score=37	Name=Crue
Id=27	Score=30	Name=Kreo
Id=28	Score=34	Name=Zexrow
Id=30	Score=30	Name=Vato

The TOP 10 Players is :

Id=25	Score=37	Name=Crue
Id=10	Score=36	Name=OrangeQueen
Id=28	Score=34	Name=Zexrow
Id=20	Score=32	Name=Wolfiez
Id=11	Score=31	Name=professorlayton
Id=27	Score=30	Name=Kreo
Id=30	Score=30	Name=Vato
Id=8	Score=25	Name=bobby15
Id=17	Score=22	Name=invisibleh24
Id=24	Score=21	Name=Fatch

The PODIUM is :

Id=25	Score=37	Name=Crue
Id=10	Score=36	Name=OrangeQueen
Id=28	Score=34	Name=Zexrow

CONGRATULATIONS !!!

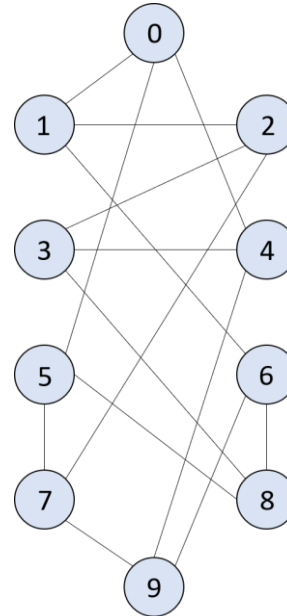
STEP 2: Professor Layton < Guybrush Threepwood < You

QUESTION 1: Represent the relation (have seen) between players as a graph, argue about your model.

We want to create a network, that is to say a graph, based on the relations between players.

A graph is composed of vertices and edges. In our case, each vertex will represent a player and each edge will represent a relation between them. There is a relation between 2 players if they have seen each other.

On the right side, you can see the relations between each player. The degree of each vertex is 3 because each player has seen 3 other players.



We decided to create a class Graph.

It is made of vertices and edges. The vertices are saved in a dictionary: the keys are the names of the players and the values are their associated number. As we have 10 players, we will have 10 values in our dictionary, from 0 to 9. The relations between the players are saved in a list named edges. Each element of this list is also a list representing one relation, following this structure: [Player 1, Player 2, Weight]. As our graph is unweighted, all the weights here will be equal to 1. (But we use this class Graph also for steps 3 and 4, with weighted graphs, that's why we already integrated the idea here).

Our class Graph has several functions: *add_Nodes* to insert one or several players in our graph, *add_Edge* and *add_Edges* to add edges to the graph. All the graphs that we will use for this project are undirected, so whenever we add the edge [Player 1, Player 2, Weight] in the graph, the edge [Player 2, Player 1, Weight] is also added.

Moreover, our class Graph contains a function to generate the adjacency matrix.

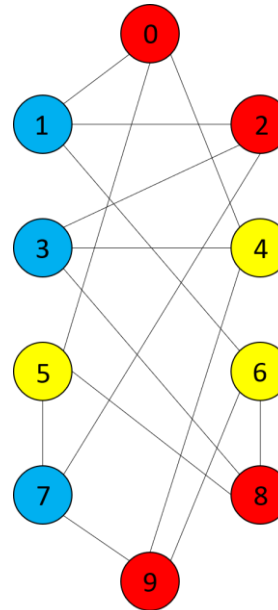
QUESTION 2: Thanks to a graph theory problem, present how to find a set of probable impostors.

The Coloring graph problem might help us to find a set of probable impostors. First of all, we begin by assigning a color to a vertex. Then, we are going to see the relations between this vertex and other vertices. If there is a relation from one vertex to another one, that is to say if these players have seen each other, then we would assign another color to the second vertex. If there are no relations between them, in other words, if the players haven't seen each other, then we would keep the same color for the second vertex. When it comes to assigning a color, we must choose a color that is less frequent in the graph.

On the right side, you can see the colored graph obtained by beginning with the vertex 0.

We can note that the chromatic-number is 3, so our graph is 3-colored. Moreover, this graph reflects our situation very well. For example, Player 3 has seen Player 4 so we need a different color for Player 4. However, the latter hasn't seen Player 5, so we will keep the same color for Player 5.

From this graph, we can find a set of probable impostors. We know that Player 0 is dead. Since Player 0 was seen by Player 1, 4 and 5, the first impostor is one them. For each of them, we need to see which vertex has the same color, for instance Player 1 and Player 3 may be impostors since they share the same color.



QUESTION 3: Argue about an algorithm solving your problem.

With the graph above, we can find the following couples of impostors: (1,3), (1,7), (4,5), (4,6), (5,6).

However, there are still many couples of impostors possible that our colored graph can't deliver to us, such as (1,8), (1,9), (4,2) ... To find those couples, we would need to look for many other colored graphs and that would not be efficient in terms of complexity. Therefore, we decided to solve this problem using a different approach.

First of all, we will use the class Graph created in the first question, and then add the vertices and the edges. From here, we can build the adjacency matrix for our graph. This matrix will be very useful to know the relations between players.

We are going to define a method that generates 2 sets of impostors given a matrix and a dead player, in our case Player 0. So we will create 2 lists of probable impostors. We will look for any Player that has seen the Player dead, if it is the case, then we will add it to our list (Player 1, Player 4 and Player 5). Next, we will again loop on our players, and if any of them is not dead, then we will add it to our second list. This method returns a list composed of the 2 lists built. The idea is that the first impostor is part of the first list and the second impostor is part of the second list.

Since we know that the second impostor didn't see the first one, we need to look at the relations between players to remove the "wrong" couples of impostors. Therefore, we will implement another method called *generate_edges* that, as indicated by his name, will find the relations between players given a graph and store them into tuples. The tuples are going to be added to a list called edges and the method will return this list.

Finally, we are going to implement a last function that calls the methods created previously. Given the 2 lists of impostors created with *generate_two_sets_impostors*, we will create all the possible couples of impostors. There are just some conditions: the second impostor should be different as the first one, and the two impostors shouldn't have seen each other (the couple shouldn't be part of the list generated with *generate_edges*). The method ends by returning *couple_impostors*.

QUESTION 4: Implement the algorithm and show a solution.

First, we create our graph. Then we generate the two sets of impostors. Impostor 1 is part of this list: [1, 4, 5] and Impostor 2 is part of this list: [1, 2, 3, 4, 5, 6, 7, 8, 9].

After that, we generate all the relations between players (who saw who): It will be a list of impossible couples of impostors because we know that impostor 2 didn't see impostor 1.

Finally, we apply our last function *generate_couple_impостors* and we find 15 possible couples of impostors:

[(1, 3), (1, 4), (1, 5), (1, 7), (1, 8), (1, 9), (4, 2), (4, 5), (4, 6), (4, 7), (4, 8), (5, 2), (5, 3), (5, 6), (5, 9)]

As expected, in every possible couple, there is at least one player who saw the player 0 (Player 1, 4 or 5).

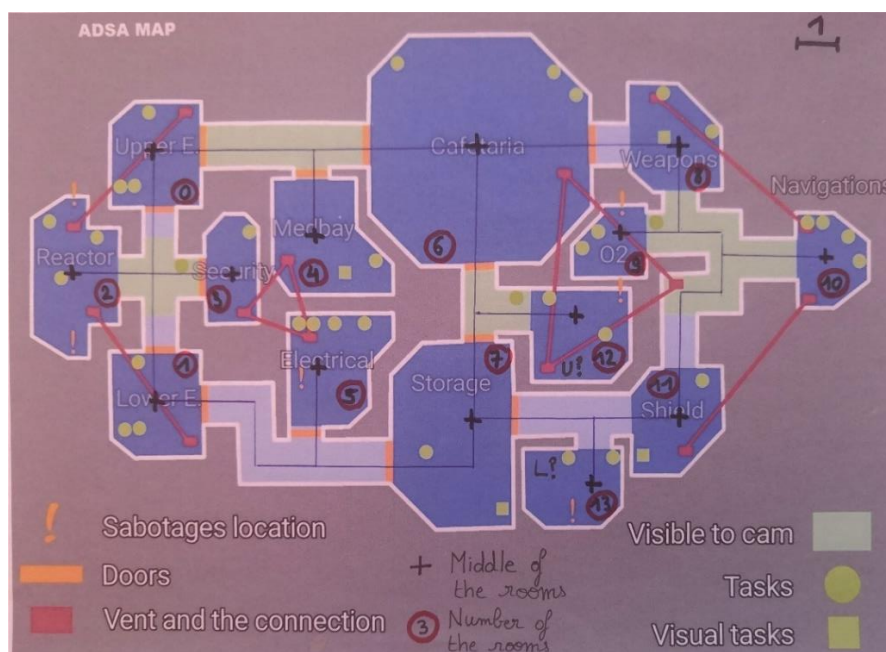
STEP 3: I don't see him, but I can give proofs he vents!

QUESTION 1: Present and argue about the two models of the map.

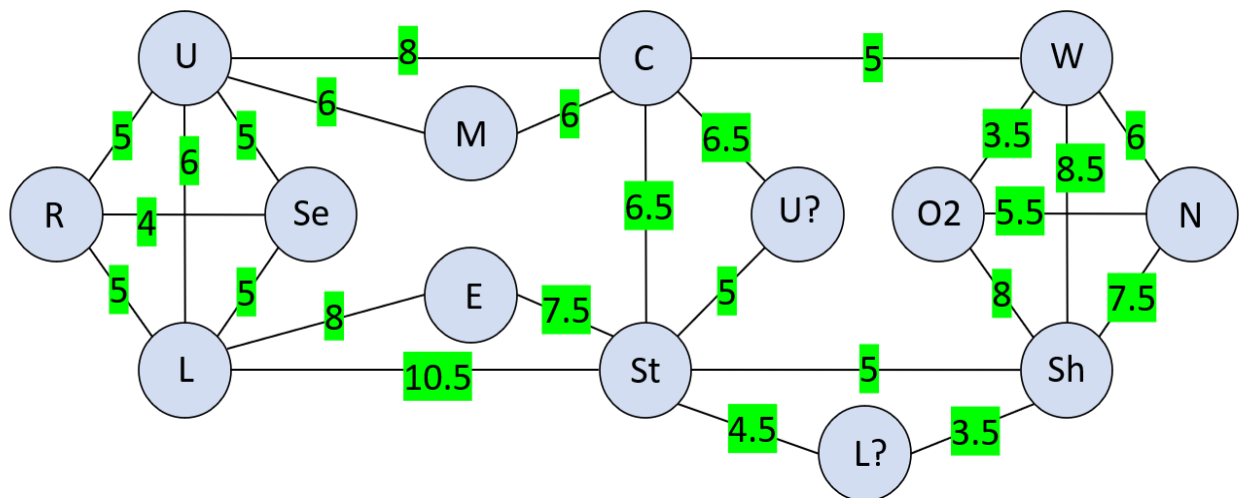
A crewmate can only walk through the map using the corridors, but an impostor can also travel with vents. So we need to represent the map twice: one with the crewmates' point of view (only with corridors), and one with the impostors' point of view (with corridors and vents).

For both models, we can consider that the map is a graph. Each vertex represents a room and the edges represent the corridors (and the vents) with an associated weight. To create these two graphs, we will use the class we created during STEP2: it is relevant because it deals with weighted and undirected graphs.

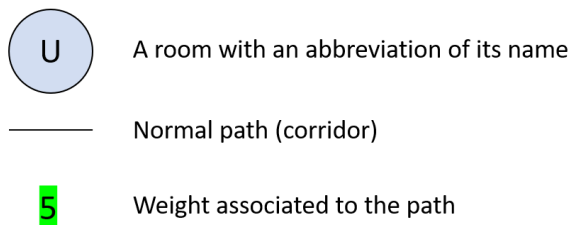
To find the weights, we printed the map and represented the middle of each room. Then we drew the links between the different rooms and measured them. 1cm on the map corresponds to a time of traveling of 1second so a weight of 1.



First graph: crewmates' model



Legend:



We reported every distance in a matrix. Our adjacency matrix looks like this:

	Upper E.	Lower E.	Reactor	Security	Medbay	Electrical	Cafeteria	Storage	Weapons	O2	Navigations	Shield	Upper ?	Lower ?
Upper E.	0	6	5	5	6	inf	8	inf	inf	inf	inf	inf	inf	inf
Lower E.	6	0	5	5	inf	8	inf	10.5	inf	inf	inf	inf	inf	inf
Reactor	5	5	0	4	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf
Security	5	5	4	0	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf
Medbay	6	inf	inf	inf	0	inf	6	inf	inf	inf	inf	inf	inf	inf
Electrical	inf	8	inf	inf	inf	0	inf	7.5	inf	inf	inf	inf	inf	inf
Cafeteria	8	inf	inf	inf	6	inf	0	6.5	5	inf	inf	inf	6.5	inf
Storage	inf	10.5	inf	inf	inf	7.5	6.5	0	inf	inf	inf	5	5	4.5
Weapons	inf	inf	inf	inf	inf	inf	5	inf	0	3.5	6	8.5	inf	inf
O2	inf	inf	inf	inf	inf	inf	inf	inf	3.5	0	5.5	8	inf	inf
Navigations	inf	inf	inf	inf	inf	inf	inf	inf	6	5.5	0	7.5	inf	inf
Shield	inf	inf	inf	inf	inf	inf	inf	5	8.5	8	7.5	0	inf	3.5
Upper ?	inf	inf	inf	inf	inf	inf	6.5	5	inf	inf	inf	inf	0	inf
Lower ?	inf	inf	inf	inf	inf	inf	inf	4.5	inf	inf	inf	3.5	inf	0

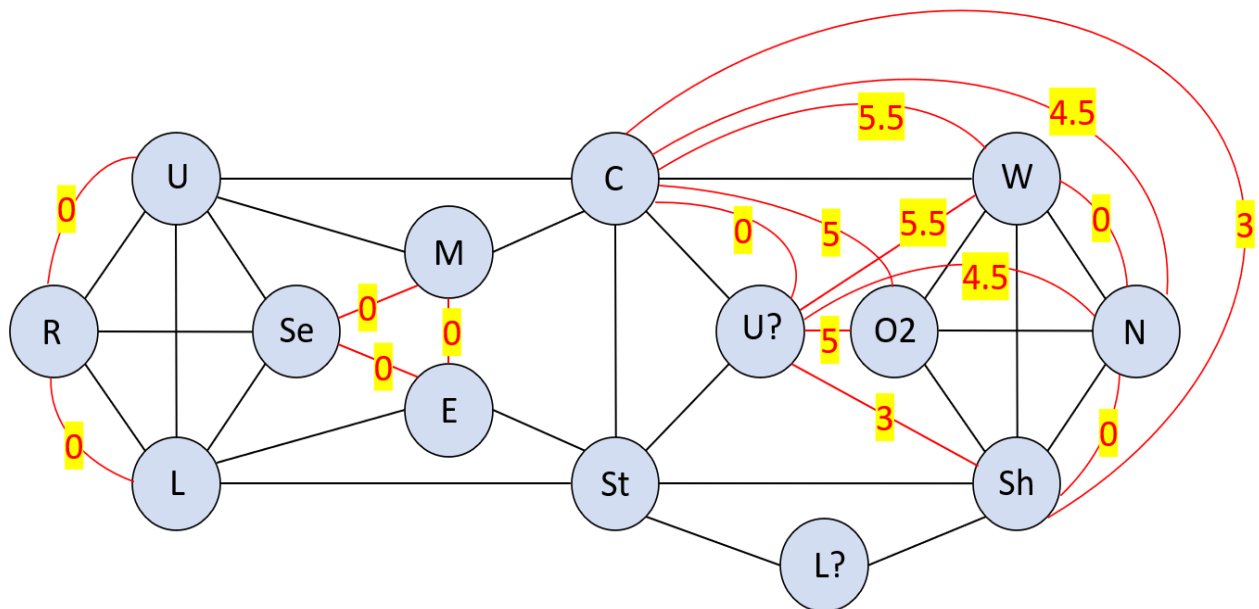
Second graph: impostors' model

This graph is an improvement of the first one.





We replaced some weight with 0. For example, the corridor between Reactor and Upper E. which had a weight of 5 has now a weight of 0 because when the impostor takes a vent, it takes no time (and there is actually a vent between Reactor and Upper E.).

We also added weights of 0 between rooms which were not directly connected before. For example, Security and Electrical rooms had no corridors between them before. Thanks to the vent, they are connected and the edge which represents it has a weight of 0.

Moreover, we added non null weights between some rooms which were not directly connected before. For example, Cafeteria and Shield rooms were not connected but a vent can bring an impostor located in the Cafeteria to a corridor next to Shield. Then the impostor still needs to walk a bit to join Shield but the weight is not very high.



Legend:

-  A room with an abbreviation of its name
-  Normal path (corridor)
-  Vent path
-  Weight associated to the vent path

The table below represents the distance matrix of the new graph. The weights which are different from the first table are highlighted in yellow.

	Upper E.	Lower E.	Reactor	Security	Medbay	Electrical	Cafeteria	Storage	Weapons	O2	Navigations	Shield	Upper ?	Lower ?
Upper E.	0	6	0	5	6	inf	8	inf	inf	inf	inf	inf	inf	inf
Lower E.	6	0	0	5	inf	8	inf	10.5	inf	inf	inf	inf	inf	inf
Reactor	0	0	0	4	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf
Security	5	5	4	0	0	0	inf	inf	inf	inf	inf	inf	inf	inf
Medbay	6	inf	inf	0	0	0	6	inf	inf	inf	inf	inf	inf	inf
Electrical	inf	8	inf	0	0	0	inf	7.5	inf	inf	inf	inf	inf	inf
Cafeteria	8	inf	inf	inf	6	inf	0	6.5	5	5	4.5	3	0	inf
Storage	inf	10.5	inf	inf	inf	7.5	6.5	0	inf	inf	inf	5	5	4.5
Weapons	inf	inf	inf	inf	inf	inf	5	inf	0	3.5	0	8.5	5.5	inf
O2	inf	inf	inf	inf	inf	inf	5	inf	3.5	0	5.5	8	5	inf
Navigations	inf	inf	inf	inf	inf	inf	4.5	inf	0	5.5	0	0	4.5	inf
Shield	inf	inf	inf	inf	inf	inf	3	5	8.5	8	0	0	3	3.5
Upper ?	inf	inf	inf	inf	inf	inf	0	5	5.5	5	4.5	3	0	inf
Lower ?	inf	inf	inf	inf	inf	inf	inf	4.5	inf	inf	inf	3.5	inf	0

In the adjacency matrix, each line represents a room.

0: Upper E
1: Lower E
2: Reactor
3: Security
4: Medbay
5: Electrical
6: Cafeteria

7: Storage
8: Weapons
9: O2
10: Navigations
11: Shield
12: Upper?
13: Lower?

“Upper?” and “Lower?” are the names we gave to the rooms which had no names. Upper? is just above Lower?.

QUESTION 2: Argue about a pathfinding algorithm to implement.

There are several algorithms able to solve shortest-path problems: Dijkstra Algorithm, Bellman-Ford Algorithm and Floyd-Warshall Algorithm. However, each of them do not share the same time complexity, nor do they fulfill the same task. Dijkstra’s Algorithm solves the single-source shortest path problem with non-negative edge weight, whereas Bellman-Ford solves the same kind of problems but deals with negative weights. Finally, Floyd-Warshall Algorithm, unlike Dijkstra and Bellman-Ford which are both single-source shortest-path algorithms, computes the shortest distances between every pair of vertices in the input graph. Since we need to calculate the time to travel for any pair of rooms, the most eligible algorithm to implement is Floyd-Warshall Algorithm.

QUESTION 3: Implements the method and show the time to travel for any pair of rooms for both models.

Time to travel for any pair of rooms with the crewmate's model:

```
[[ 0.  6.  5.  5.  6. 14.  8. 14.5 13. 16.5 19. 19.5 14.5 19. ]
 [ 6.  0.  5.  5. 12.  8. 14. 10.5 19. 22.5 23. 15.5 15.5 15. ]
 [ 5.  5.  0.  4. 11. 13. 13. 15.5 18. 21.5 24. 20.5 19.5 20. ]
 [ 5.  5.  4.  0. 11. 13. 13. 15.5 18. 21.5 24. 20.5 19.5 20. ]
 [ 6. 12. 11. 11.  0. 20.  6. 12.5 11. 14.5 17. 17.5 12.5 17. ]
 [14.  8. 13. 13. 20.  0. 14.  7.5 19. 20.5 20. 12.5 12.5 12. ]
 [ 8. 14. 13. 13.  6. 14.  0.  6.5  5.  8.5 11. 11.5  6.5 11. ]
 [14.5 10.5 15.5 15.5 12.5  7.5  6.5  0. 11.5 13. 12.5  5.  5.  4.5]
 [13. 19. 18. 18. 11. 19.  5. 11.5  0.  3.5  6.  8.5 11.5 12. ]
 [16.5 22.5 21.5 21.5 14.5 20.5  8.5 13.  3.5  0.  5.5  8. 15. 11.5]
 [19. 23. 24. 24. 17. 20. 11. 12.5  6.  5.5  0.  7.5 17.5 11. ]
 [19.5 15.5 20.5 20.5 17.5 12.5 11.5  5.  8.5  8.  7.5  0. 10.  3.5]
 [14.5 15.5 19.5 19.5 12.5 12.5  6.5  5. 11.5 15. 17.5 10.  0.  9.5]
 [19. 15. 20. 20. 17. 12. 11.  4.5 12. 11.5 11.  3.5  9.5  0. ]]
```

Time to travel for any pair of rooms with the impostor's model (with vents):

```
[[ 0.  0.  0.  4.  4.  4.  8. 10.5 11. 13. 11. 11.  8. 14.5]
 [ 0.  0.  0.  4.  4.  4.  8. 10.5 11. 13. 11. 11.  8. 14.5]
 [ 0.  0.  0.  4.  4.  4.  8. 10.5 11. 13. 11. 11.  8. 14.5]
 [ 4.  4.  4.  0.  0.  0.  6.  7.5  9. 11.  9.  9.  6. 12. ]
 [ 4.  4.  4.  0.  0.  0.  6.  7.5  9. 11.  9.  9.  6. 12. ]
 [ 4.  4.  4.  0.  0.  0.  6.  7.5  9. 11.  9.  9.  6. 12. ]
 [ 8.  8.  8.  6.  6.  6.  0.  5.  3.  5.  3.  3.  0.  6.5]
 [10.5 10.5 10.5  7.5  7.5  7.5  5.  0.  5.  8.5  5.  5.  5.  4.5]
 [11. 11. 11.  9.  9.  9.  3.  5.  0.  3.5  0.  0.  3.  3.5]
 [13. 13. 13. 11. 11. 11.  5.  8.5  3.5  0.  3.5  3.5  5.  7. ]
 [11. 11. 11.  9.  9.  9.  3.  5.  0.  3.5  0.  0.  3.  3.5]
 [11. 11. 11.  9.  9.  9.  3.  5.  0.  3.5  0.  0.  3.  3.5]
 [ 8.  8.  8.  6.  6.  6.  0.  5.  3.  5.  3.  3.  0.  6.5]
 [14.5 14.5 14.5 12. 12. 12.  6.5  4.5  3.5  7.  3.5  3.5  6.5  0. ]]
```

We can see that there are way more “0” in this model, and the times to travel are way smaller for a lot of routes.

QUESTION 4: Display the interval of time for each pair of rooms where the traveler is an impostor.

In the matrix below we can see the interval of time for each pair of rooms where the traveler is an impostor. Indeed, the values in the matrix represent the “win of time” the impostors got thanks to the vent (it’s just the difference between the 2 matrices of Question 3).

For example, if the impostors take the vents between room 4 (Medbay) and room 5 (Electrical), they save 20seconds, which is really a lot!

When there is a 0, it means they took as much time as the crewmates, because there was no vent so they can’t be unmasked.

```
[[ 0.  6.  5.  1.  2. 10.  0.  4.  2.  3.5  8.  8.5  6.5  4.5]
 [ 6.  0.  5.  1.  8.  4.  6.  0.  8.  9.5 12.  4.5  7.5  0.5]
 [ 5.  5.  0.  0.  7.  9.  5.  5.  7.  8.5 13.  9.5 11.5  5.5]
 [ 1.  1.  0.  0. 11. 13.  7.  8.  9. 10.5 15. 11.5 13.5  8. ]
 [ 2.  8.  7. 11.  0. 20.  0.  5.  2.  3.5  8.  8.5  6.5  5. ]
 [10.  4.  9. 13. 20.  0.  8.  0. 10.  9.5 11.  3.5  6.5  0. ]
 [ 0.  6.  5.  7.  0.  8.  0. 1.5  2.  3.5  8.  8.5  6.5  4.5]
 [ 4.  0.  5.  8.  5.  0. 1.5  0.  6.5  4.5  7.5  0.  0.  0. ]
 [ 2.  8.  7.  9.  2. 10.  2.  6.5  0.  0.  6.  8.5  8.5  8.5]
 [ 3.5  9.5  8.5 10.5  3.5  9.5  3.5  4.5  0.  0.  2.  4.5 10.  4.5]
 [ 8. 12. 13. 15.  8. 11.  8.  7.5  6.  2.  0.  7.5 14.5  7.5]
 [ 8.5  4.5  9.5 11.5  8.5  3.5  8.5  0.  8.5  4.5  7.5  0.  7.  0. ]
 [ 6.5  7.5 11.5 13.5  6.5  6.5  6.5  0.  8.5 10. 14.5  7.  0.  3. ]
 [ 4.5  0.5  5.5  8.  5.  0.  4.5  0.  8.5  4.5  7.5  0.  3.  0. ]]
```

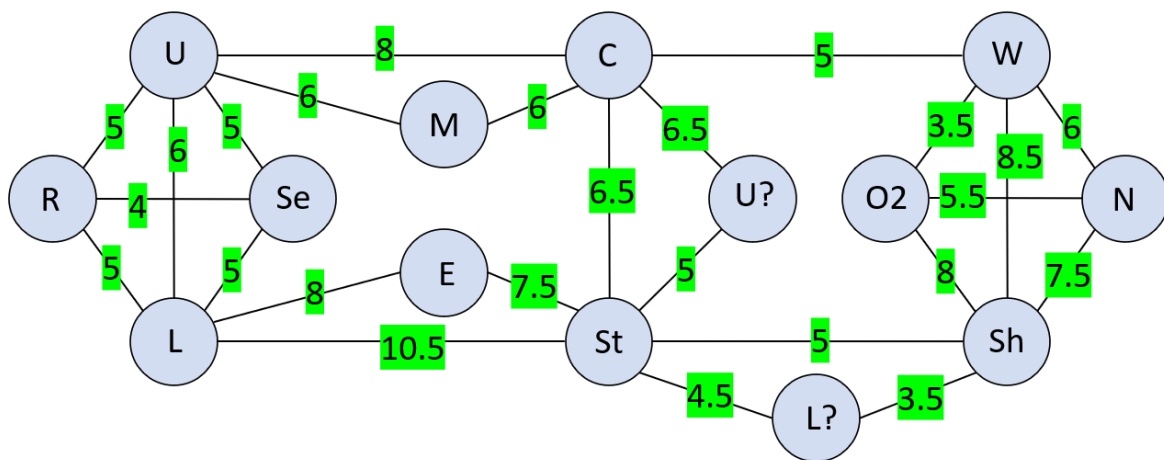
STEP 4: Secure the last tasks

QUESTION 1: Present and argue about the model of the map.

During this part, we form a pack with all remaining players. As a consequence, impostors cannot kill anyone or cannot take any vent otherwise they would be unmasked.

That's why we decided to represent the map with the crewmate's graph we designed in STEP3 - Part1. In this graph, we don't consider the vents.

We want to finish every task. As there are tasks in every room, we keep exactly the same graph (crewmate's graph).



QUESTION 2: Thanks to a graph theory problem, present how to find a route passing through each room only one time.

We want to find a route passing through each room only one time. As our graph is undirected, this problem is the same as finding the Hamiltonian paths of our graph.

The crewmate's graph is built thanks to our Graph class. After that, we can build the adjacency matrix (we obtain the same as for STEP3 – Part1).

In this matrix, we have “inf” values when it’s not possible to connect directly 2 rooms, and we have a finite number (=the weight) which represents the distance between 2 rooms when it’s possible to connect them directly.

The purpose is to pass by every room exactly one time, the quickest possible.

So first, we need to find all the possible routes passing by every room exactly once.

Then, we need to compare the costs of these routes (i.e How much time we need to pass every room of the route), by doing the sum of all the weights.

At the end, we choose the route which has the minimum cost because we want to travel the quickest possible.

QUESTION 3: Argue about an algorithm solving your problem.

First of all, we need to find all the possible routes (the Hamiltonian paths). We created a function called *Hamilton* which takes the adjacency matrix of our graph as parameter and will return all the Hamilton paths of our graph in a list called *complete_route*.

We loop on our rooms: for each room, we try to find a route(s) which starts with this room. To do so, we call a recursive function *HamiltonRecurisif*. This function takes several parameters: the adjacency matrix, a route (the actual route we are trying to build), a list of impossible routes (the ones which can't pass by every room exactly one time), and the list of complete routes (the Hamiltonian paths).

First, we check if the actual route is complete (passes by every room). If it is the case, we add it to our list of complete routes.

If not, we check the last room visited in the actual route in order to see which rooms we can visit next (rooms that are directly connected so a value different to "inf" in the adjacency matrix, and rooms that are not already in the route because each room should be visited only once). We add these future rooms in a queue called *possible_next_rooms*.

Then, we try to add a room of this queue in our actual route. If this leads to an impossible route, we remove the last room. If not, we call again *HamiltonRecurisif*: this time the actual route is longer than previously because we just added a room.

While we didn't try all the possibilities (while there are still non visited rooms in *possible_next_rooms* so while the queue is not empty), we repeat the process.

When a queue *possible_next_rooms* is empty, it means we tried all the possibilities for a route so we need to add it in our list of *impossible_routes*.

At the end, *Hamilton* find all the possible routes starting by each room.

To find the quickest route between all of them, we created a function called *FastestRoute* which computes the cost of each route and returns the fastest one with its cost. If several routes have the same minimum cost, it returns a list with all of them.

To compute the cost of a route, we need the adjacency matrix. There are 14 rooms in a complete route so 13 edges with an associated weight. The cost of the route is the sum of the weights of these 13 edges.

QUESTION 4: Implement the algorithm and show a solution.

We found in total 80 routes passing by every room exactly one time.

To be able to pass every room, we need to start with:

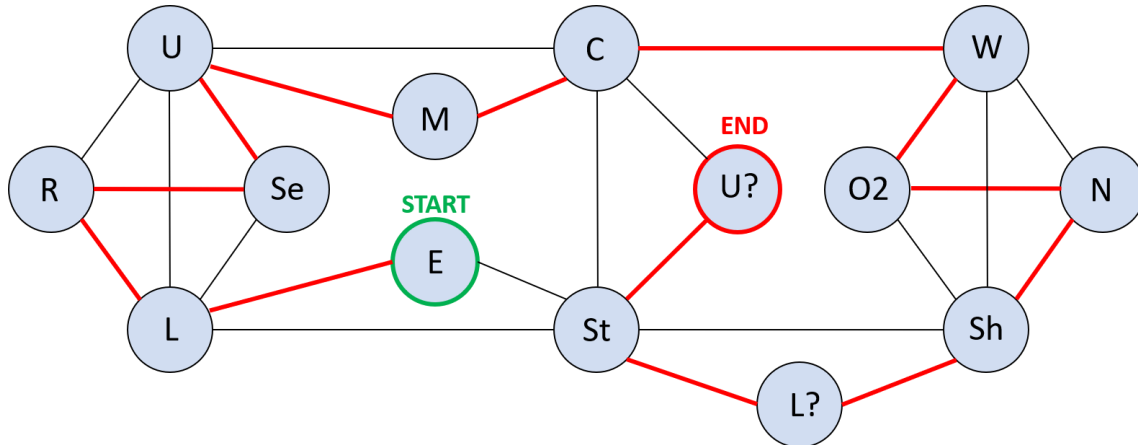
- Medbay (room 4 in our adjacency matrix): 8 possible ways
- Electrical (room 5): 16 possible ways
- Weapons (room 8): 8 possible ways
- O2 (room 9): 8 possible ways
- Navigations (room 10): 8 possible ways
- Upper? (room 12): 24 possible ways
- Lower? (room 13): 8 possible ways

Then we applied our algorithm to find the fastest route(s) between these 80 possible routes.

The fastest way to pass every room takes 68.5 seconds (it means the sum of all the weights is 68.5).
In total, 4 routes allow this:

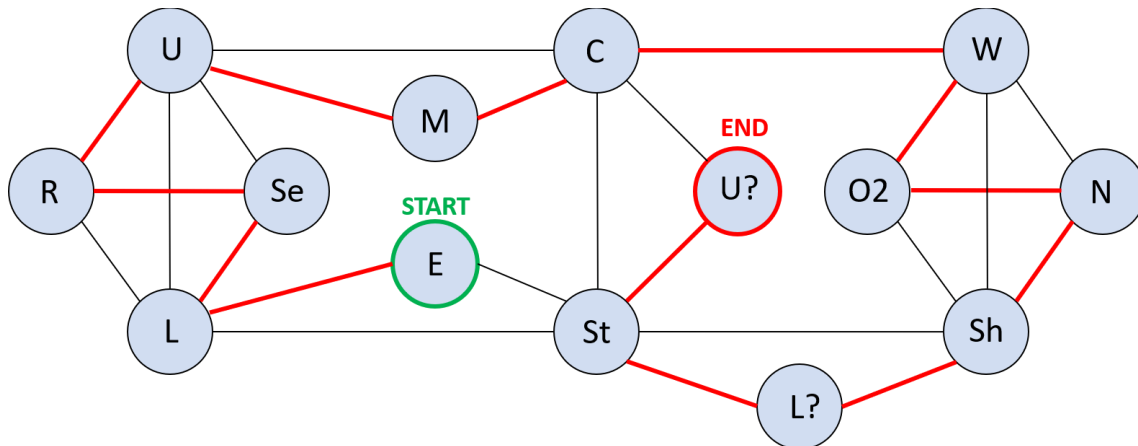
- [5, 1, 2, 3, 0, 4, 6, 8, 9, 10, 11, 13, 7, 12]

Electrical → Lower E → Reactor → Security → Upper E → Medbay → Cafeteria → Weapons → O2
→ Navigations → Shield → Lower? → Storage → Upper?



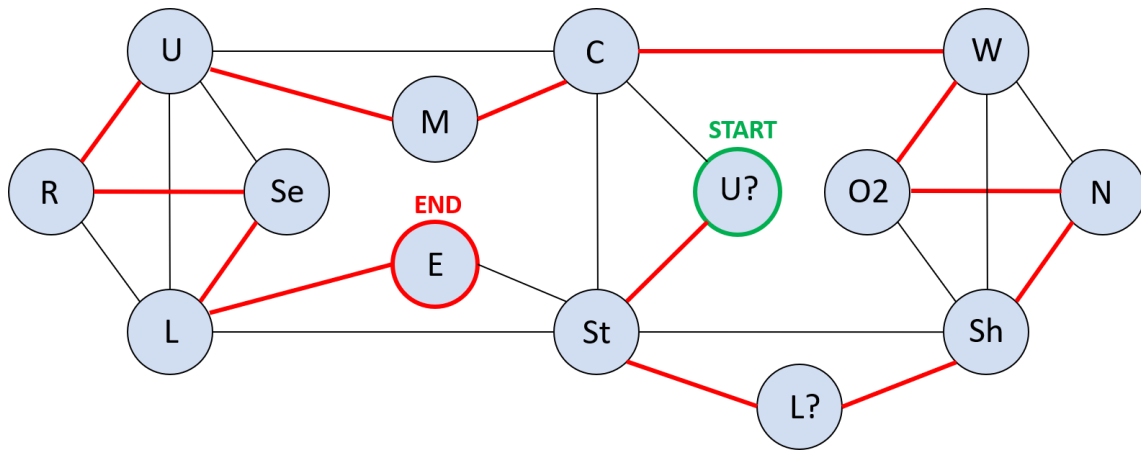
- [5, 1, 3, 2, 0, 4, 6, 8, 9, 10, 11, 13, 7, 12]

Electrical → Lower E → Security → Reactor → Upper E → Medbay → Cafeteria → Weapons → O2
→ Navigations → Shield → Lower? → Storage → Upper?



- [12, 7, 13, 11, 10, 9, 8, 6, 4, 0, 2, 3, 1, 5]

Upper? → Storage → Lower? → Shield → Navigations → O2 → Weapons → Cafeteria → Medbay →
Upper E → Reactor → Security → Lower E → Electrical



- [12, 7, 13, 11, 10, 9, 8, 6, 4, 0, 3, 2, 1, 5]

Upper? → Storage → Lower? → Shield → Navigations → O2 → Weapons → Cafeteria → Medbay →
 Upper E → Security → Reactor → Lower E → Electrical

