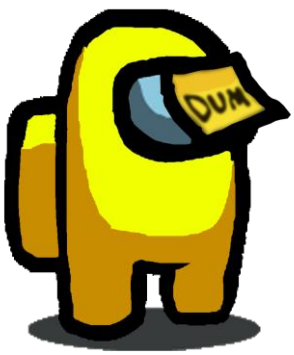
AMONG-US

Mini-project

Tristan Darrigol

Ulysse Berthet



DIA-2

AMONG-US :

Mini-project

Step 1:

Organizing the tournament

Step 2:

Professor Layton < Guybrush Threepwood < You

Step 3:

I don’t see him, but I can give proofs he vents !

Step 4:

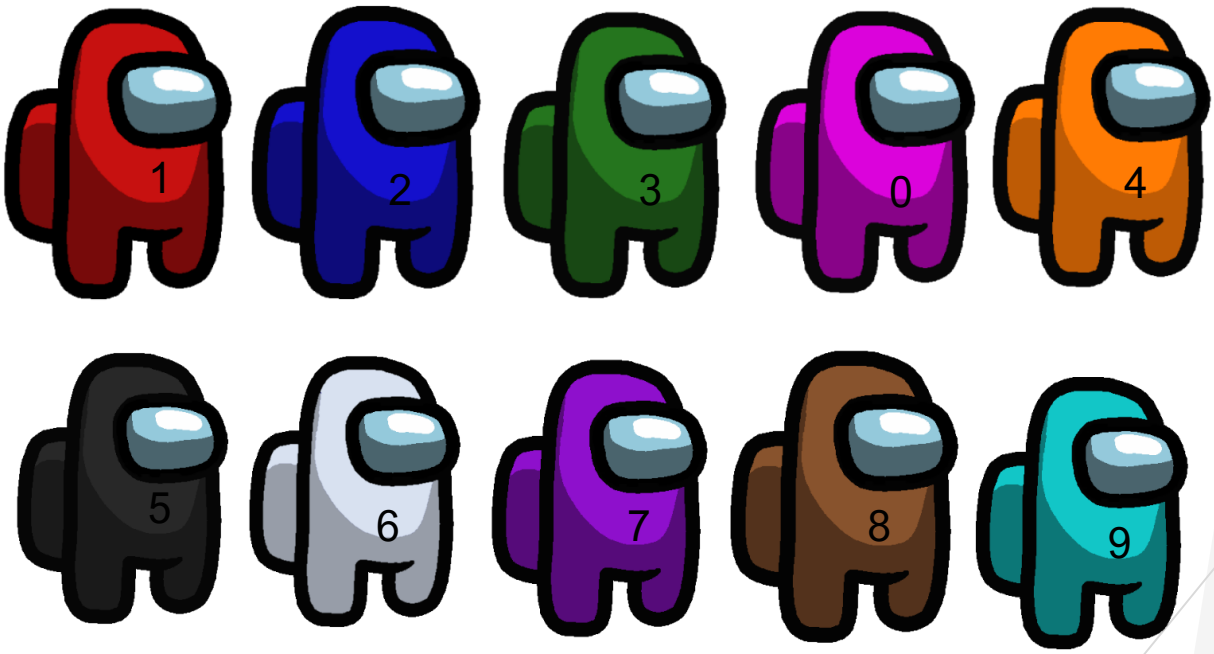
Secure the last tasks

AMONG-US :

Mini-project

Step 1:

Organize the tournament



Representing a player and its score :

To represent a player and its score we create a class Player containing the following parameters :

- **id** : to track the player throuout the tournament
- **score** : the average score of the player for all of its games
- **nb_game :** the number of games the player played. It is used to calculate the average score.
- **left, right and height :** used to store the player in an AVL tree
- It also has some functions :
- **__str__ :** used to print a description of the player in the following format :
- *[player id] : [average score] x [number of games played]*
- **add_score :** used to add a new game score to the player and re calculate his average using the following formula :
- new_average_score =

$$\frac{\text{previous\_average\_score} \times \text{previous\_number\_of games} + \text{game\_score}}{\text{new\_number\_of\_games}}$$

- **reset_score :** reset the player number of game and average score to play the final 5 games.
- **reset_childrens :** reset the player children to insert it into a new tree
- **display** : display the tree, showing the score of the player at each node
- **_display_aux :** to create the display tree line by line

Example of the display using 1 players with score ranging from 0 to 9 :

```
Insertion Order :
Player : 0 : 0.0 x 1
Player : 1 : 1.0 x 1
Player : 2 : 2.0 x 1
Player : 3 : 3.0 x 1
Player : 4 : 4.0 x 1
Player : 5 : 5.0 x 1
Player : 6 : 6.0 x 1
Player : 7 : 7.0 x 1
Player : 8 : 8.0 x 1
Player : 9 : 9.0 x 1

Tree display :

      ____3.0_________
     /                \
  _1.0_            ____7.0_
 /     \          /        \
0.0   2.0      _5.0_      8.0_
              /     \          \
            4.0     6.0        9.0
```

Data Structure for the tournament :

We decided to use an AVL tree because it is the most optimized data structure when we need to insert and delete nodes in variable places in the data and keep the nodes in order :

Indeed, we need to perform insertion, search and deletion.

- In an AVL tree the average and worst-case complexity for searching, insertion and deletion is O(log n).
- Compared to O(n) for an array (worst and average case) and a BST (only for worst case).
- We can’t use stack, queue, singly or doubly linked list because we need to insert the data not just at the beginning or end of the list.

Our AVL tree stores Players (described in question 1) and is ordered by average player score.

Data Structure for the tournament :

The AVL tree was slightly modified to allow the use of node with the same score.

During the insertion it use the following algorithm :

```
if new.score < current.score :
    insert to the left
elif new.score > current.score :
    insert to the right
elif new.score = current.score :
    if height right > height left :
        insert to the left
    elif height right < height left :
        insert to the right
    elif height right = height left :
        insert to the right or the left randomly
```

As for depletion :

```
if search.score < current.score :
    go to the left
elif search.score > current.score :
    go to the right
elif search.score = current.score
    if search.id != search.id:
        go to the left
        go to the right
    else :
        delete the node like a classic AVL
```

Randomizing a player Score at each game :

In order to randomize the player's score, we use the following rules to simulate a game :

Here is the ranking model:

- Impostor: 1pts per kill, 3pts per undiscovered murder, 10pts if win
- Crewmate: 3pts if the argument unmasks an imposter, 1pts if all solo tasks are made, 5pts if win

We are then able to generate 10 scores from the game and apply it to the 10 players using the following algorithm :

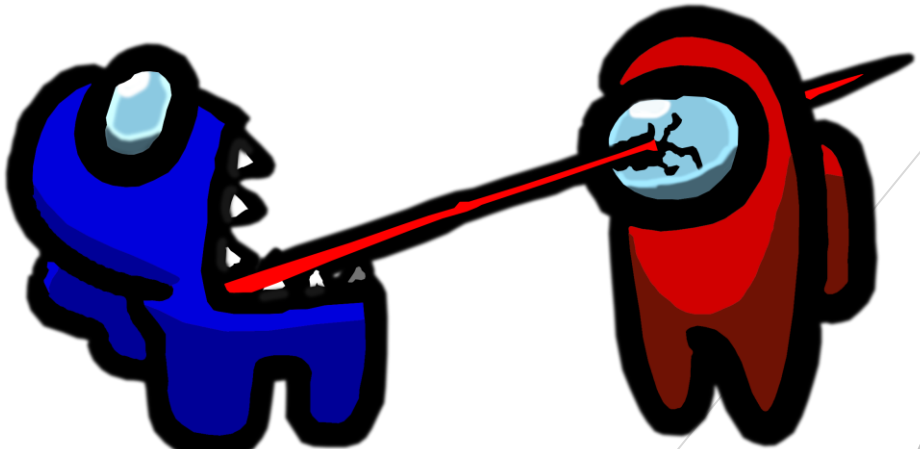
```
random_game(players)
    impostors <- create list of length 2 filled with 0
    crewmates <- create list of length 8 filled with 0

    impostorsWin <- randomly set to True or False

    if impostorsWin:
        every impostors get 10 points for winning the game
        first impostor gets between 0 and 8 points for his kills
        second impostor impostors gets between 0 and the number of kills of the first impostors for his kills
        first impostor gets between 0 and 3 times his number of kills (in steps of 3) for his undiscovered murders
        second impostor gets between 0 and 3 times his number of kills (in steps of 3) for his undiscovered murders

        unmasked_impostors <- randomly set to True or False
        if unmasked_impostors:
            one crewmate gets 3 points
        all crewmates except at least one can randomly get 1 point if all their tasks where made

    else :
        first impostor gets between 0 and 7 points for his kills
        second impostor impostors gets between 0 and the number of kills remaining
        first impostor gets between 0 and 3 times his number of kills (in steps of 3) for his undiscovered murders
        second impostor gets between 0 and 3 times his number of kills (in steps of 3) for his undiscovered murders

        win_task  <- randomly set to True or False
        if win_task:
            unmasked_impostors <- randomly set to True or False
                if unmasked_impostors:
                    one crewmate gets 3 points
            every crewmate get 1 point
        else :
            one crewmates gets 3 points
            one crewmates gets 3 points
            all crewmates except at least one can randomly get 1 point if all their tasks where made

    liste_score <- concatenation of impostors and crewmates
    every score over 12 in liste_score is remplaced by 12
    a score from liste_score gets randomly assigned to a player from players
```

Randomizing a player Score at each game :

- To win the impostors need to kill all the crewmates.
  Their kill can get the bonus points if its undiscovered.
  If the impostors win, the crewmates can still gain points by completing theirs task (but not all for every crewmate) or discovering an impostor.

- To win the crewmates needs to unmask the impostors or complete all their tasks.
  If they complete all their task, they can also discover an impostor. If they discover all impostors, some crewmates can complete all their tasks (but not all the crewmates). In both cases the impostors can still kill up to 7 of the crewmates.

- The score need to be between 0 and 12 so if its higher so we replace every score higher than 12 by 12



Updating Player Scores and the Database :

We update the player score and their ranking in the following way:

First, we delete the last 10 players of the tree.

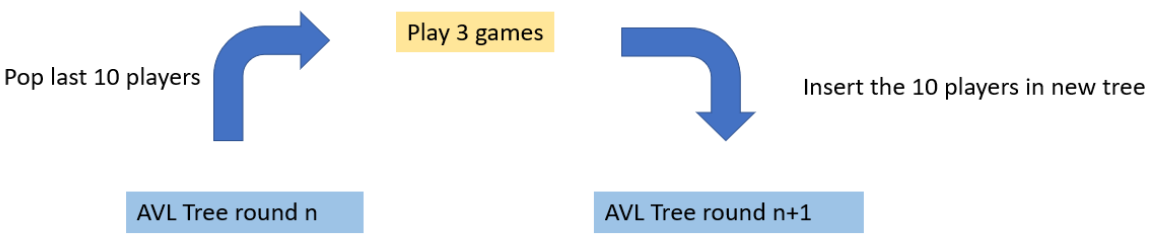
Then we pop the following last 10 players of the tree, play the 3 games and insert them in a new tree until the first tree is empty (see following graph).

Search, insertion and deletion have complexity of O(log n).

The game have a complexity of O(n) to apply the new score to players.

To do that we use the function manche1, wich generate the player at the beginning, plays the first 3 games and insert them in the first tree.

Then we use the manche function wich plays every games until the end.



Creating and Playing games:

Creating a random game :

To create random games, we apply the function random game described beforehand as much times as needed (3 during the first phases and 5 at the end). To generate multiples games, we use the function nGames which apply the function random game n times.

Creating a game based on ranking :

To create the games, we use the manche function which select the last 10 players, play a game and insert them in a new tree.

Droping the players :

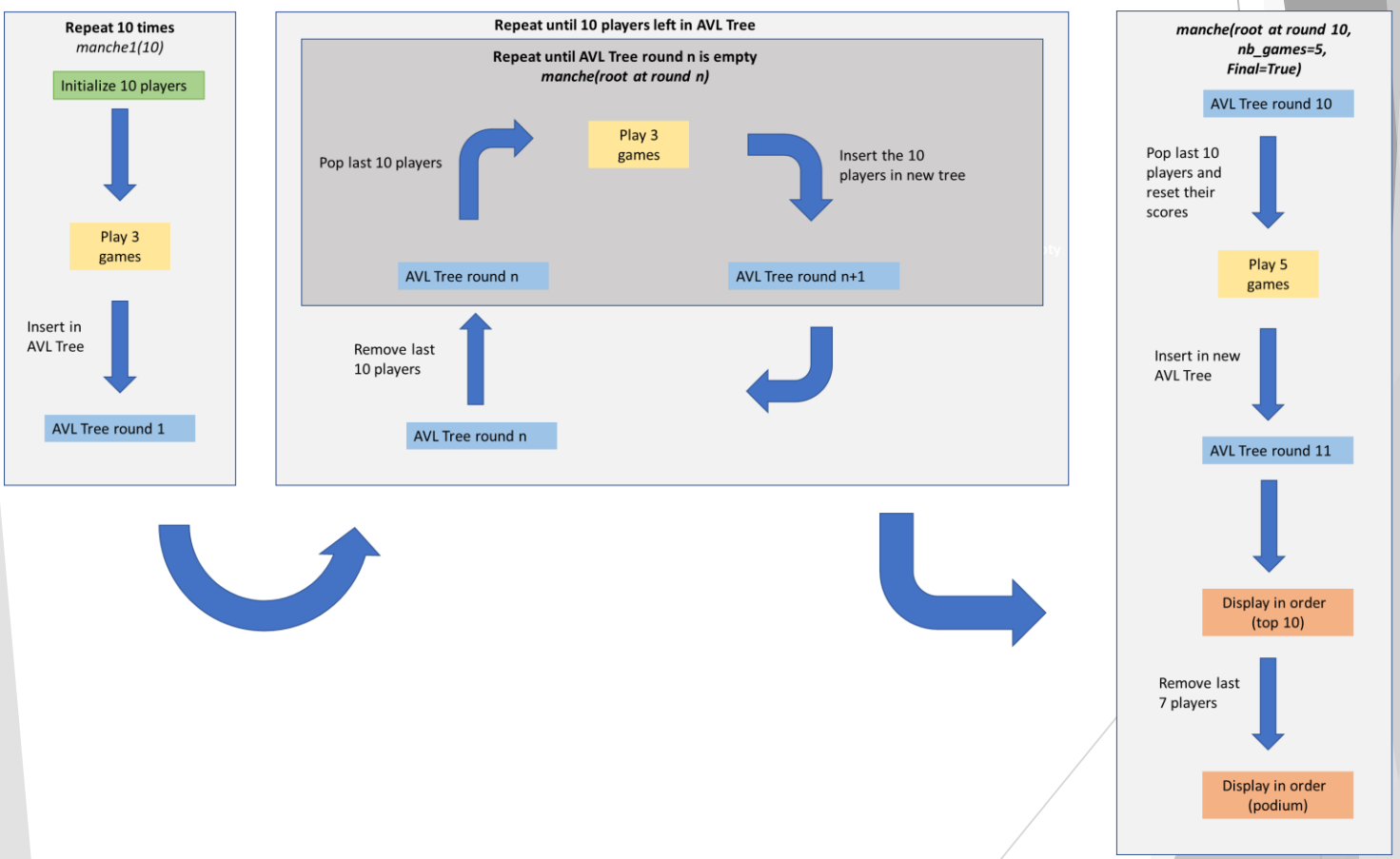
The manche function first drop the last 10 players based on ranking before playing the games with the remaining players .

Displaying the top players:

In order to display the top 10, we display the last tree in order. To display the podium (top 3) we delete the last 7 players and display the tree in order.

Creating and Playing games:

To summarize the different steps, we made the following graph :



AMONG-US :

Mini-project

Step 2:

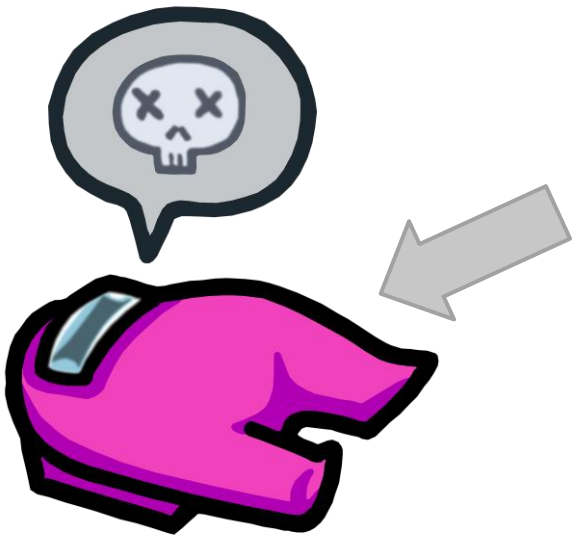
Professor Layton < Guybrush Threepwood < You

Setup :

As we are not only the tournament organizer, but also a player, we need to find a way to win the tournament.

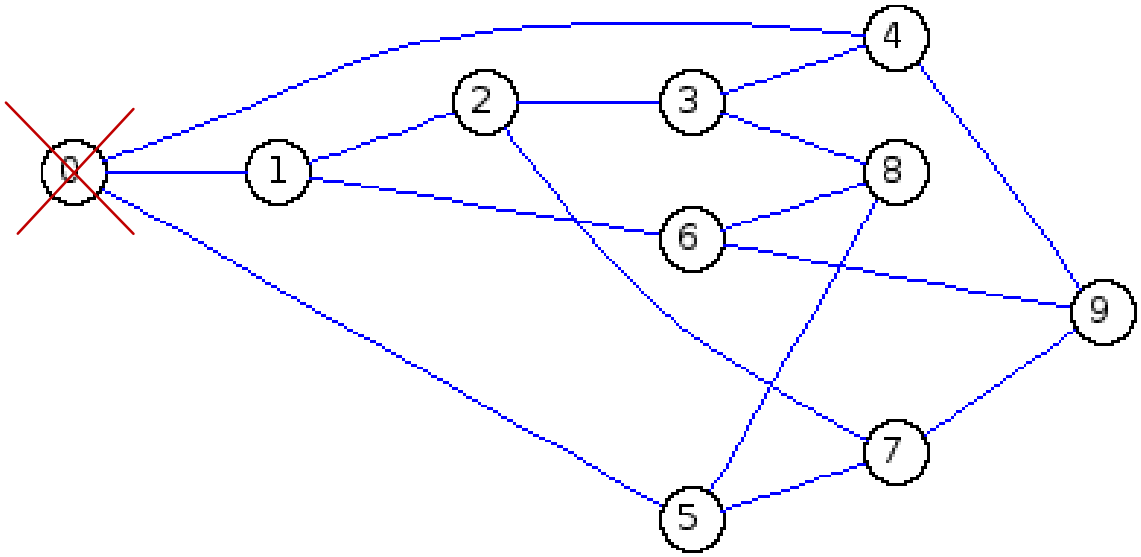
We'd like to exploit the informations about players which have been seen together. After the first kill, we have the following informations :

- Player 0 has seen player 1, 4 and 5
- Player 1 has seen player 0, 2 and 6
- Player 2 has seen player 1, 3 and 7
- Player 3 has seen player 2, 4 and 8
- Player 4 has seen player 0, 3 and 9
- Player 5 has seen player 0, 7 and 8
- Player 6 has seen player 1, 8 and 9
- Player 7 has seen player 2, 5 and 9
- Player 8 has seen player 3, 5 and 6
- Player 9 has seen player 4, 6 and 7.



Player 0 has been reported dead

Representing the «have seen» relation :



First of all, we do not take player 0 into account.

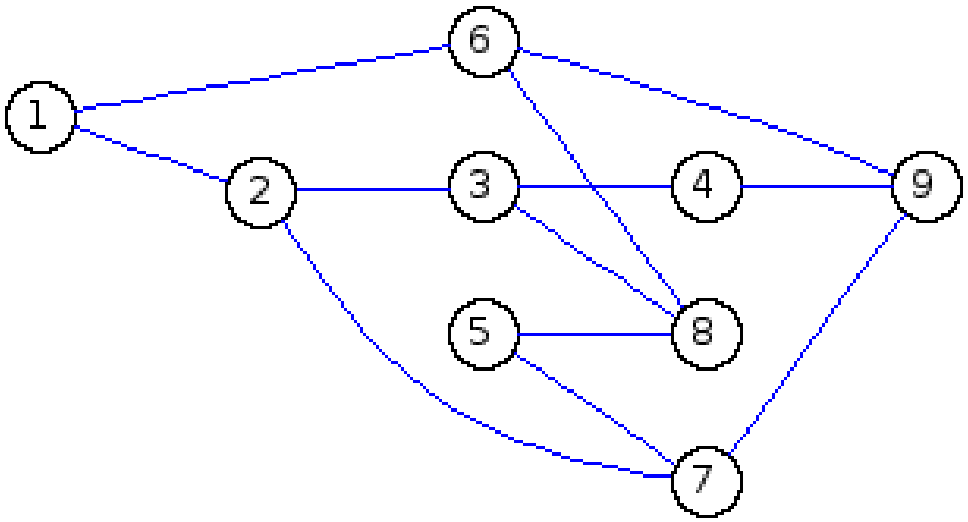
As a matter of fact he is dead, and a crewmate. And he has seen player 1,4 and 5.

As we use this information as a « seed » information : each of the three players that 0 has seen are used as « seed impostors » in order to create the impostors sets.

Thus, incorporating player 0 in the graph does not make any sense for our problem, and do not help us finding the potentials imposters set, as we are using the informaton given by 0's death beforehand.

Representing the «have seen» relation :

Thus, we choose to represent the « have seen » relation by the following graph :



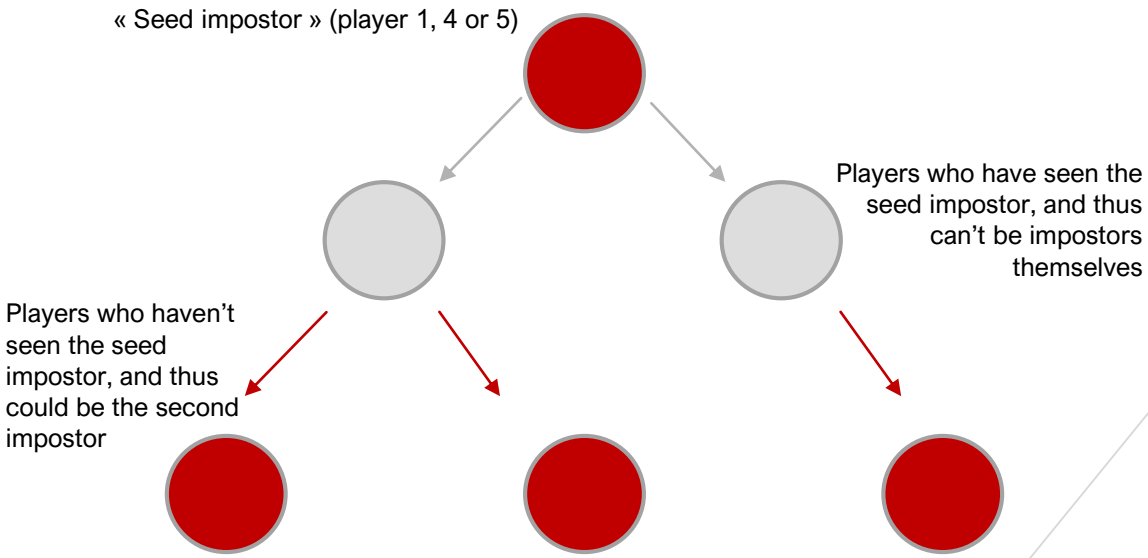
- Vertices represent players, from player 1 to player 9, and the edges represent the fact that two players have already seen each others at the time 0 is reported dead.
- The graph is undirected, the «have seen » relation is mutual, if player A has seen player B, player B has seen player A.
- The graph is unweighted, all « have seen » relations have the same value

Finding a set of probable impostors using a graph theory problem :

First of all, what do we know about the players ?

- The only players who saw 0 before he was killed are player 1,4 and 5.
- We know that the second imposter hasn't seen the first one

So, we will use a simple colorization problem in order to find the imposters set :

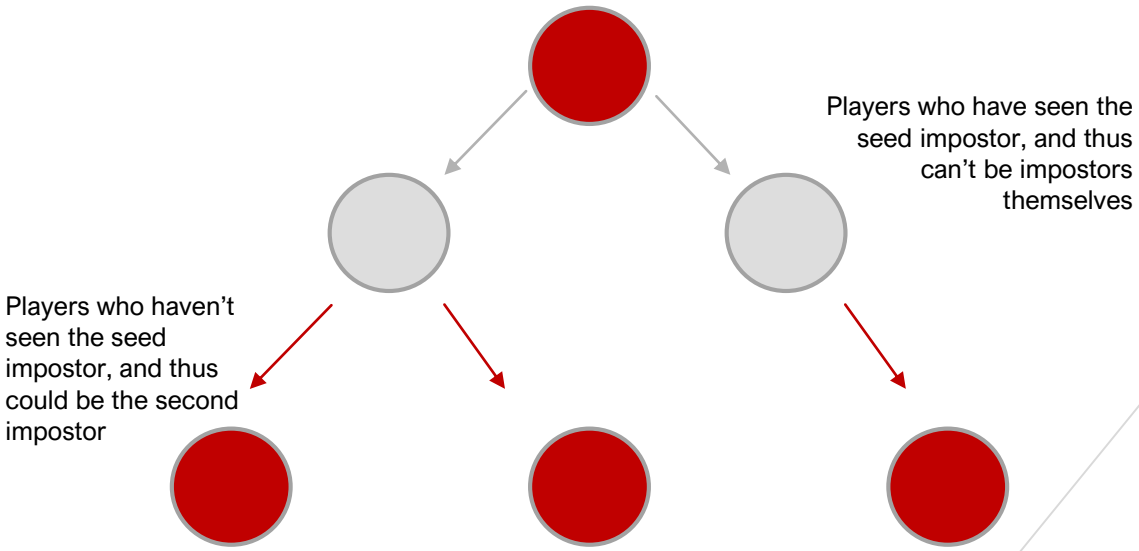


Finding a set of probable impostors using a graph theory problem :

We will use the following rules for the colorization problem :

- The edge representing the seed impostor will get a color.
- The direct neihgbours of the seed vertices are colorized with a different color than the first one as they can't be impostors
- All others vertices are colorized the same way as the first vertice, in order not to forget any potential impostor.

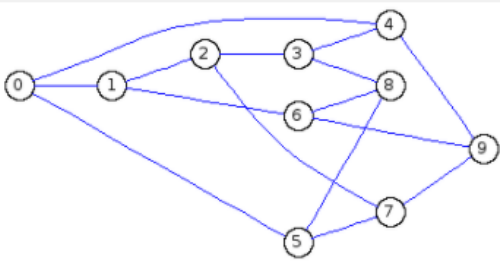
« Seed impostor » (player 1, 4 or 5)



An algorithm to solve the problem :

Representing the graph using its adjacency matrix :

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



We have to implement an algorithm that « colorize » only the neighbourings vertices of the seed vertice

An algorithm to solve the problem :

A simple way to solve this problem is to first colorize all vertices the same coloras the seed vertice, and then change the color of all the directly-neighbouring vertices :

Pseudo-Code Colorization Algorithm :

Input : Adjacency Matrix of a graphe, list of potential impostors

1. Initialize the list corresponding to the color of each player (all 1 for example)
2. Create an empty list
3. For every item in the list of potential impostors do :
    1. For every player (column) in the adjacency matrix do :
        1. If the value corresponding to the relation « have seen » between the potential impostor and the player in the adjacency matrix is equal to 1 then :
           Change the color of said player
    2. Add every impostors tuple to the empty list representing the set of potential impostors

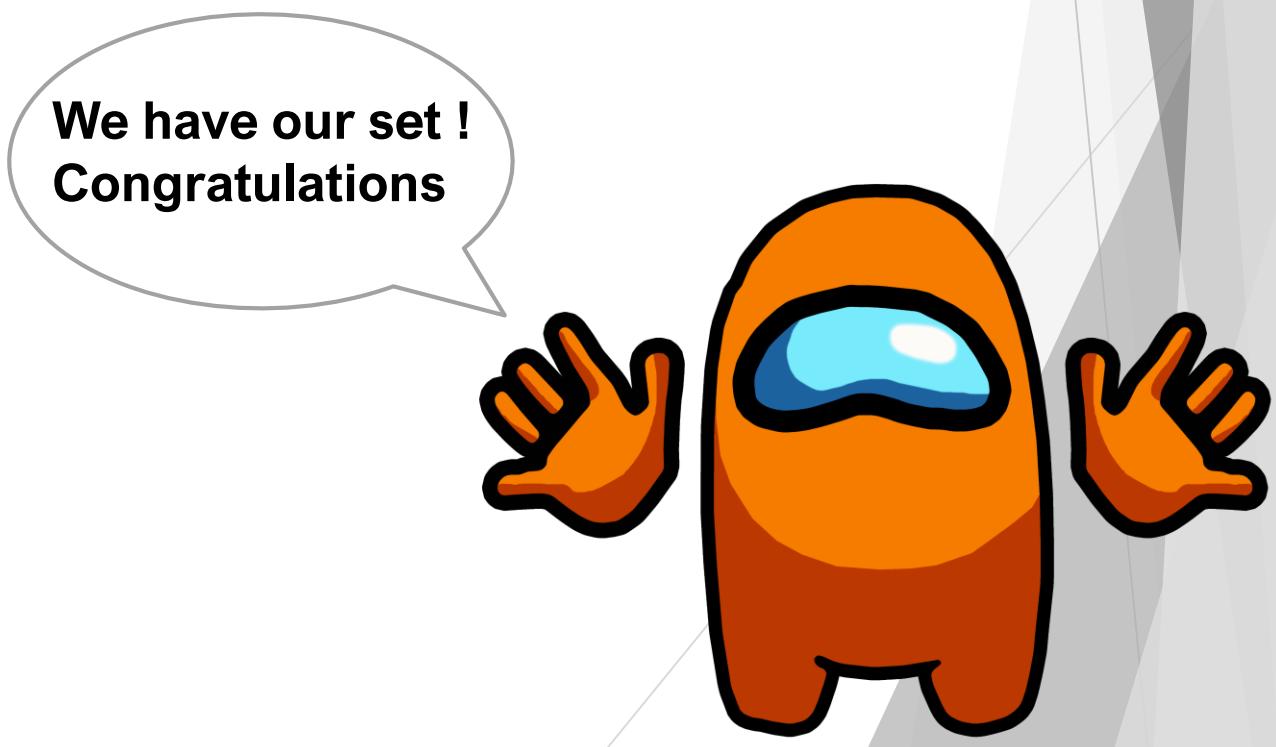
Solution :

The solution to our problem using the algorithm we discussed earlier is the following :

```
The possible impostor set if player 1 killed player 0 is S1 =  {[(1, 3), (1, 4), (1, 5),
(1, 7), (1, 8), (1, 9)]}
The possible impostor set if player 4 killed player 0 is S2 =  {[(4, 1), (4, 2), (4, 5),
(4, 6), (4, 7), (4, 8)]}
The possible impostor set if player 5 killed player 0 is S3 =  {[(5, 1), (5, 2), (5, 3),
(5, 4), (5, 6), (5, 9)]}
```

We obtain three sets, depending of the seed. The final set is then an union of those three subsets, minus the duplicates :

S_{final} = {(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)}

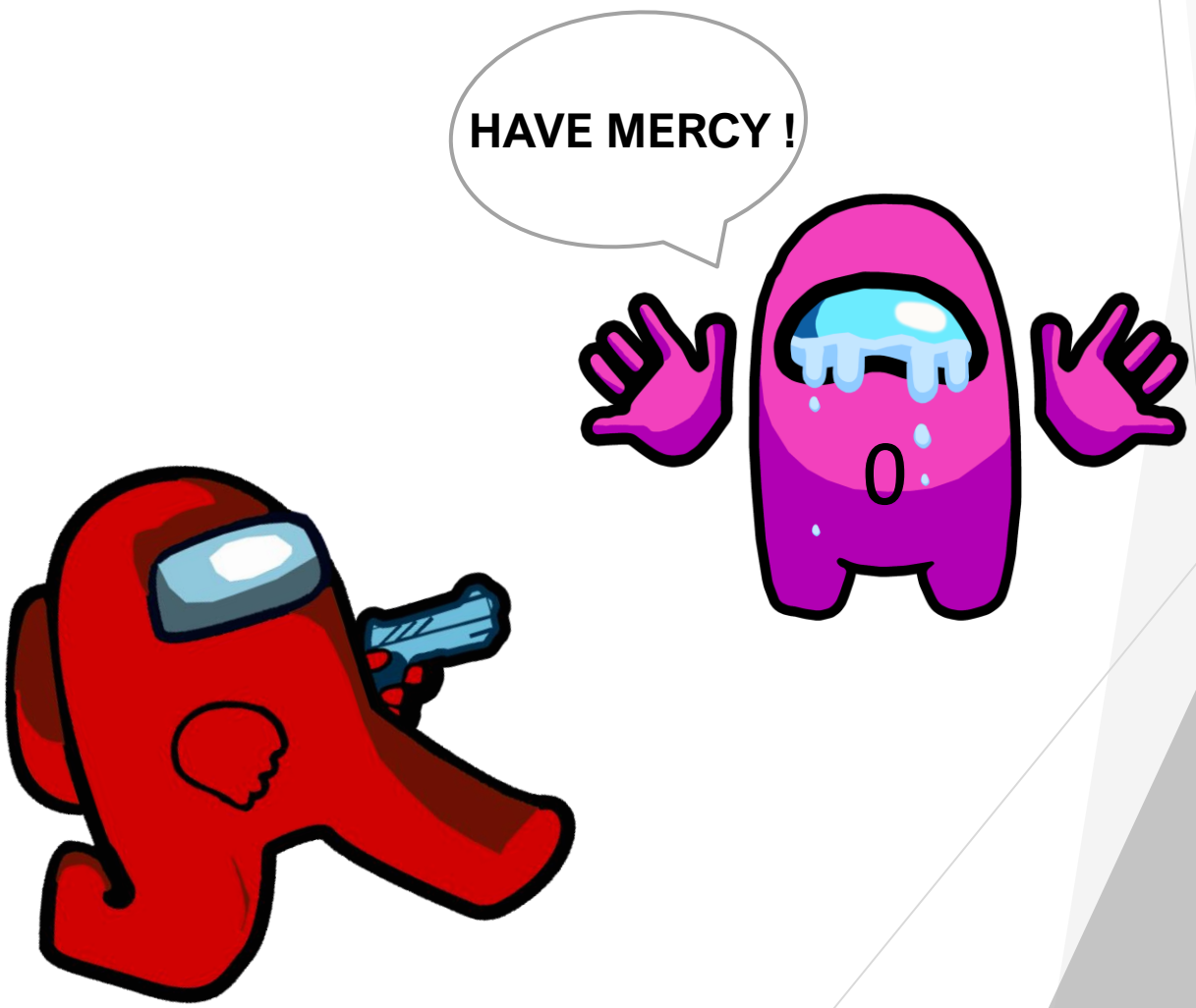


AMONG-US :

Mini-project

Step 3:

I don’t see him, but I can give proofs he vents !



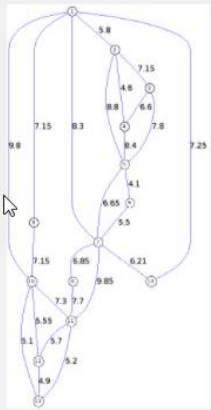
Two models of map :

Indeed, the crewmates map's representation won't be the same as the impostors one, considering the fact that they can't vent.

We'll first present the crewmates map using the following indexes :

1	2	3	4	5	6	7	8	9	10	11	12	13	14
Cafeteria	Weapons	Navigation	O2	Shield	Communication	Storage	Electrical	Mdedbay	Upper E	Lower E	Reactor	Security	Admin

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	5.8	0	0	0	0	8.3	0	7.15	9.8	0	0	0	7.25
2	5.8	0	7.15	4.6	8.8	0	0	0	0	0	0	0	0	0
3	0	7.15	0	6.6	7.8	0	0	0	0	0	0	0	0	0
4	0	4.6	6.6	0	8.4	0	0	0	0	0	0	0	0	0
5	0	8.8	7.8	8.4	0	4.1	6.65	0	0	0	0	0	0	0
6	0	0	0	0	4.1	0	5.5	0	0	0	0	0	0	0
7	8.3	0	0	0	6.65	5.5	0	6.85	0	0	9.85	0	0	6.21
8	0	0	0	0	0	0	6.85	0	0	0	7.7	0	0	0
9	7.15	0	0	0	0	0	0	0	0	7.15	0	0	0	0
10	9.8	0	0	0	0	0	0	0	7.15	0	7.3	5.55	5.1	0
11	0	0	0	0	0	0	9.85	7.7	0	7.3	0	5.7	5.2	0
12	0	0	0	0	0	0	0	0	0	5.55	5.7	0	4.9	0
13	0	0	0	0	0	0	0	0	0	5.1	5.2	4.9	0	0
14	7.25	0	0	0	0	0	6.21	0	0	0	0	0	0	0

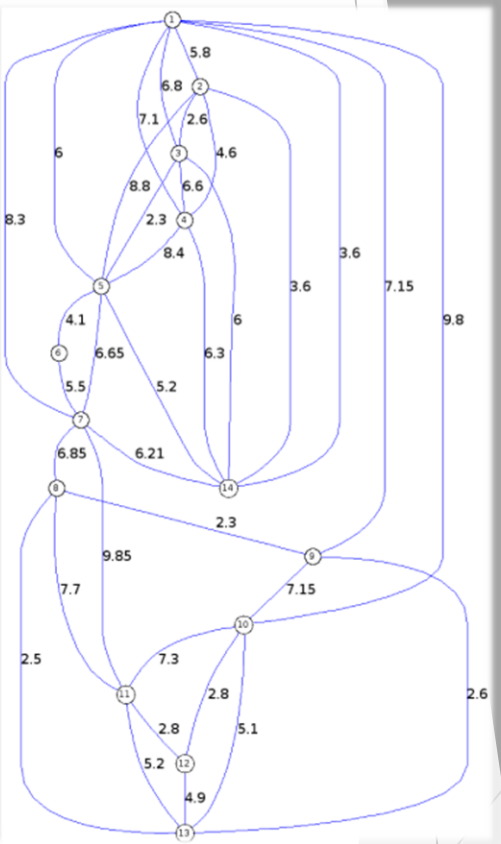


Every room is represented by a vertice, and a road from one room to another is represented by an edge, the weight is representing the distance between those two rooms. The graph is undirected, because any road go both ways.

Two models of map :

As the impostor can use the vents, we need to modify the representation of the map in order to add those possibilities. Starting from the crewmates mape, we measure the distance from a room's vent to it's center, and add / replace values in the matrix. Also, sometimes it's shorter not to go by the center of the room, and directly leave the room after taking the vent, we take that into account :

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	5.8	6.8	7.1	6	0	8.3	0	7.15	9.8	0	0	0	3.6
2	5.8	0	2.6	4.6	8.8	0	0	0	0	0	0	0	0	3.6
3	6.8	2.6	0	6.6	2.3	0	0	0	0	0	0	0	0	6
4	7.1	4.6	6.6	0	8.4	0	0	0	0	0	0	0	0	6.3
5	6	8.8	2.3	8.4	0	4.1	6.65	0	0	0	0	0	0	5.2
6	0	0	0	0	4.1	0	5.5	0	0	0	0	0	0	0
7	8.3	0	0	0	6.65	5.5	0	6.85	0	0	9.85	0	0	6.21
8	0	0	0	0	0	0	6.85	0	2.3	0	7.7	0	2.5	0
9	7.15	0	0	0	0	0	0	2.3	0	7.15	0	0	2.6	0
10	9.8	0	0	0	0	0	0	0	7.15	0	7.3	2.8	5.1	0
11	0	0	0	0	0	0	9.85	7.7	0	7.3	0	2.8	5.2	0
12	0	0	0	0	0	0	0	0	0	2.8	2.8	0	4.9	0
13	0	0	0	0	0	0	0	2.5	2.6	5.1	5.2	4.9	0	0
14	3.6	3.6	6	6.3	5.2	0	6.21	0	0	0	0	0	0	0



We note that this graph is way more complicated than the first one, as impostors have way more possibility of moves

Every room is represented by a vertice, and a road from one room to another is represented by an edge, the weight is representing the distance between those two rooms.

Pathfinding Algorithm :

There are two mains pathfinding algorithms :

- Dijkstra :

  Gives the shortest path from a vertice v to all other vertices. It returns distance, and vertices to visit, in a form of a dictionnary. The time complexity is

- Floyd-Warshall :

  Gives all « distances » of the shortest path from any vertice to any other vertices of a graph, in form of a distance matrix. The time complexity

Using adjacency list, Dijkstra's complexity D is :

D = O($Number\ of\ edges + number\ of\ vertices\ * \log(number\ of\ vertices)$)

D1 = $number\ of\ vertices *$ D

As for Floyd-Warshall, the complexity D2 is :

D2 = $(number\ of\ vertices)$^3

For Both graph, we have 14 vertices, the only difference is the number of edges, let's note n the number of vertices, and e the number of edges :

D1 = $e * n + n^2 * \log(n)$
D2 = $n * n^2$
We then have D1 > D2 if and only if : $e > n^2 - nlog(n)$
For n = 14 we have n² - nlog(n) ~ 180

Thus, as we never have 180 edges, we'll use dijkstra's

Pathfinding Algorithm :

We chose to use Dijkstra's algorithm, applied to all vertices.

Even though, we implemented both algorithms, in order to verify if, in the application, Dijkstra was actually faster.

We confirmed that it is.

Here is the pseudo-code for Dijkstra's pathfinding algorithm

Pseudo-Code Dijkstra's Algorithm :

Input : Adjacency List of a graphe, an edge's index, we'll note « s »

1. P = Empty ensemble
2. Q = Ensemble of all vertices
3. While Q ain't empty do :
    1. u = vertice of Q with min distance
    2. For every vertices connected to u do :
        1. d = distance to u + dist(u,v)
        2. If d< d(s,v), then:
            1. d(s,v) = d
    3. Add u's distance and path to P, and delete it from Q
4. Return(P)

Solutions :

As we do not really care about the path, but just about the travelling times, we'll build a table in order to easily compare the travelling times as crewmate and impostor :

Travelling Times for Crewmates

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1		5,8	12,95	10,4	14,6	13,8	8,3	15,15	7,15	9,8	17,1	15,35	14,9	7,25
2	5,8		7,15	4,6	8,8	12,9	14,1	20,95	12,95	15,6	22,9	21,15	20,7	13,05
3	6,8	2,6		6,6	7,8	11,9	14,45	21,3	20,1	22,75	24,3	28,3	27,85	20,2
4	7,1	4,6	6,6		8,4	12,5	15,05	21,9	17,55	20,2	24,9	25,75	25,3	17,65
5	6	4,9	2,8	8,4		4,1	6,65	13,5	21,75	23,8	16,5	22,2	21,7	12,86
6	10,1	9	6,4	12,5	4,1		5,5	12,35	20,95	22,65	15,35	21,05	20,55	11,71
7	8,3	9,81	8,95	12,51	6,65	5,5		6,85	15,45	17,15	9,85	15,55	15,05	6,21
8	9,45	15,25	15,8	16,55	13,5	12,35	6,85		22,15	15	7,7	13,4	12,	13,06
9	7,15	12,95	13,95	14,25	13,15	14,65	9,15	2,3		7,15	14,45	12,7	12,25	14,4
10	9,8	15,6	16,6	16,9	15,8	19,9	14,45	7,6	7,15		7,3	5,55	5,1	17,05
11	14,95	19,66	18,8	22,05	16,5	15,35	9,85	7,7	7,8	5,6		5,7	5,25	16,06
12	12,6	18,4	19,4	19,7	18,6	18,15	12,65	7,4	75	2,8	2,8		4,9	21,76
13	9,75	15,55	16,55	16,85	15,75	14,85	9,35	2,5	2,6	5,1	5,2	4,9		21,26
14	3,6	3,6	6	6,3	5,2	9,3	6,21	13,05	10,75	13,4	16,06	16,2	13,35	

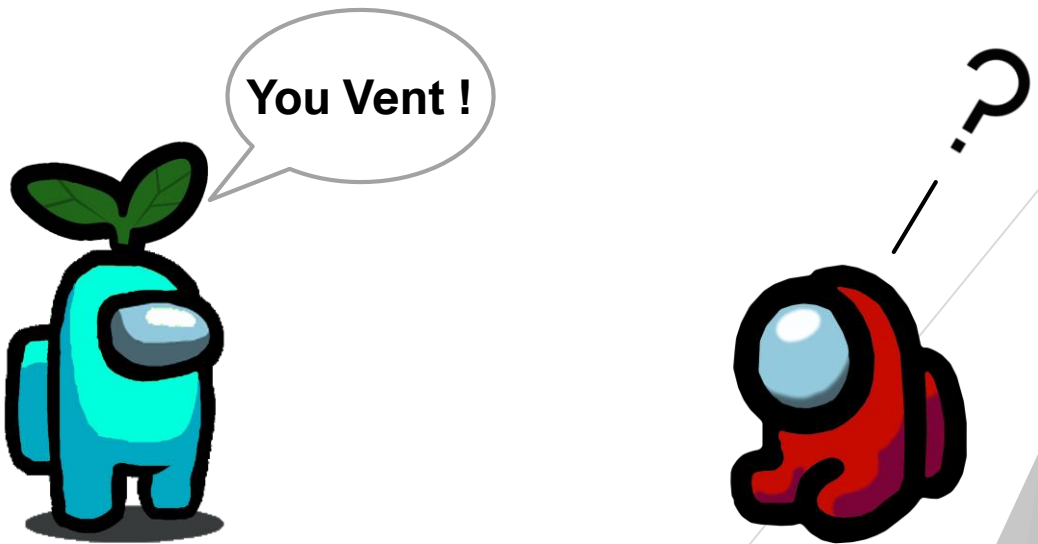
Travelling Times for Impostors

Solutions :

Using the previous table, we can easily see the gap of travelling time bewteen crewmates and impostors. But, in order to get a better idea of the differences, we implemented a quick function that list all travelling times differences between crewmates and impostors. Here is an extract from the display :

```
Impostors take 3.3 secs less than crewmates to reach O2 from Caféteria
Impostors take 8.6 secs less than crewmates to reach Shield from Caféteria
Impostors take 3.7 secs less than crewmates to reach Communication from Caféteria
Impostors take 5.7 secs less than crewmates to reach Electrical from Caféteria
Impostors take 2.15 secs less than crewmates to reach Lower E from Caféteria
Impostors take 2.75 secs less than crewmates to reach Reactor from Caféteria
Impostors take 5.15 secs less than crewmates to reach Security from Caféteria
Impostors take 3.65 secs less than crewmates to reach Admin from Caféteria
Impostors take 4.55 secs less than crewmates to reach Navigations from Weapons
Impostors take 3.9 secs less than crewmates to reach Shield from Weapons
Impostors take 3.9 secs less than crewmates to reach Communication from Weapons
Impostors take 4.29 secs less than crewmates to reach Storage from Weapons
Impostors take 5.7 secs less than crewmates to reach Electrical from Weapons
Impostors take 3.24 secs less than crewmates to reach Lower E from Weapons
Impostors take 2.75 secs less than crewmates to reach Reactor from Weapons
Impostors take 5.15 secs less than crewmates to reach Security from Weapons
Impostors take 9.45 secs less than crewmates to reach Admin from Weapons
Impostors take 5.5 secs less than crewmates to reach Shield from Navigations
Impostors take 5.5 secs less than crewmates to reach Communication from Navigations
Impostors take 5.5 secs less than crewmates to reach Storage from Navigations
```

We finally have a way to proove who’s venting and who’s not !

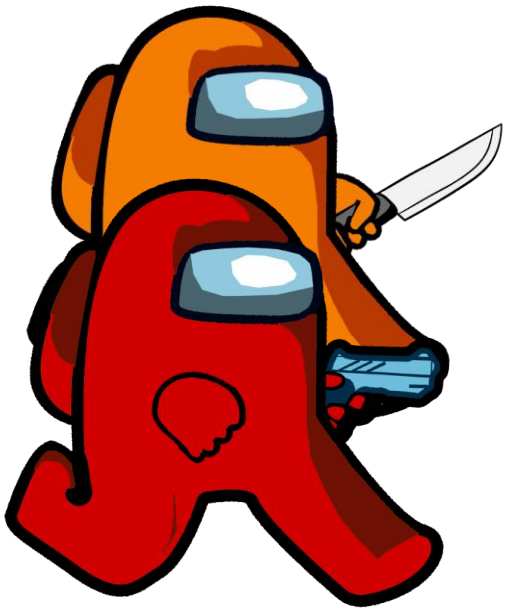


AMONG-US :

Mini-project

Step 4:

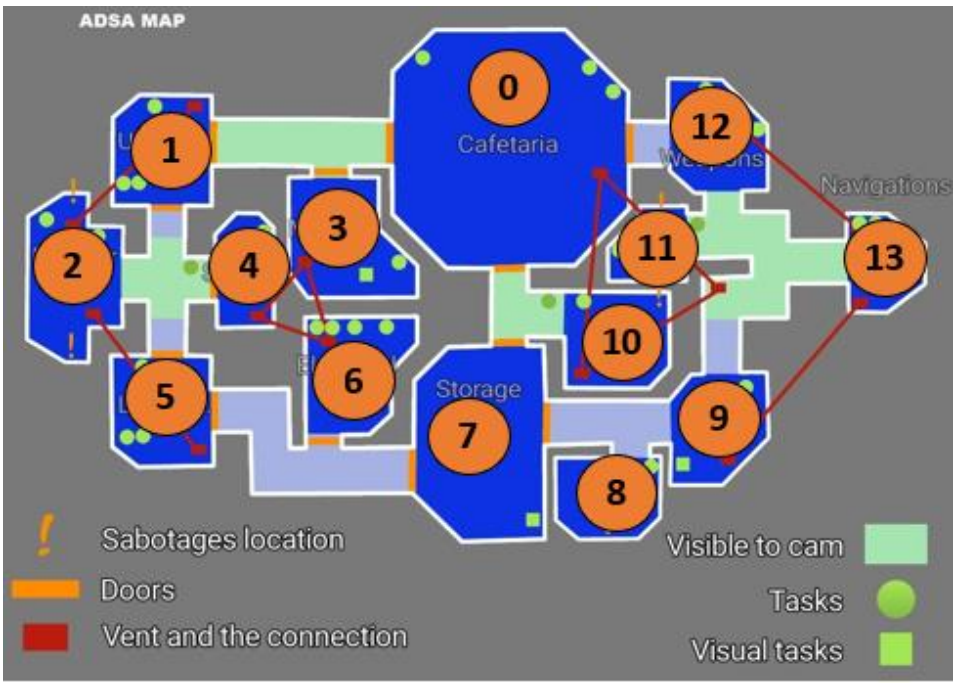
Secure the last tasks



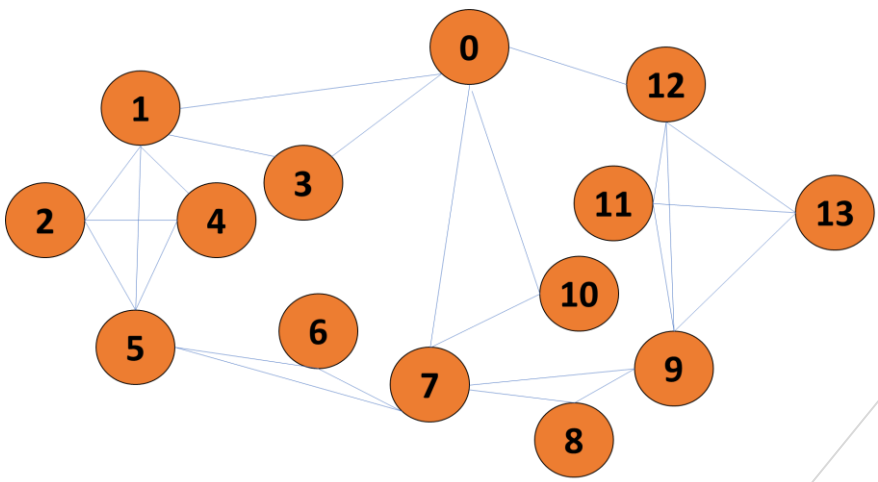
Model of the Map :

We'll use a model similar to the crewmates one in Step 3:

Every room is a vertice, they connected corridors are represented by the edges. The graph is undirected because we can go in both directions. The impostor can't use the vents otherwise he will be unmasked, so we don't put the vents on the graph.



Giving the following graph :



Model of the Map :

To represent the graph, we use and agency matrix : One line for each vertices, 1 it is connected to the vertices of this index, 0 if it isn't :

Room	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	1	0	1	0	0	0	1	0	0	1	0	1	0
1	1	0	1	1	1	1	0	0	0	0	0	0	0	0
2	0	1	0	0	1	1	0	0	0	0	0	0	0	0
3	1	1	0	0	0	0	0	0	0	0	0	0	0	0
4	0	1	1	0	0	1	0	0	0	0	0	0	0	0
5	0	1	1	0	1	0	1	1	0	0	0	0	0	0
6	0	0	0	0	0	1	0	1	0	0	0	0	0	0
7	1	0	0	0	0	1	1	0	1	1	1	0	0	0
8	0	0	0	0	0	0	0	1	0	1	0	0	0	0
9	0	0	0	0	0	0	0	1	1	0	0	1	1	1
10	1	0	0	0	0	0	0	1	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1	1
12	1	0	0	0	0	0	0	0	0	1	0	1	0	1
13	0	0	0	0	0	0	0	0	0	1	0	1	1	0

Graph Theory Problem :

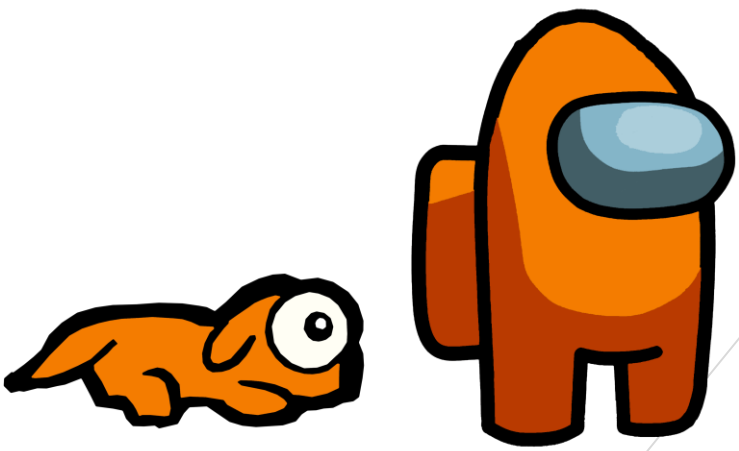
This is known as the “Traveling Salesman” problem.

We need to find a Hamiltonian path. A path that goes to every room one and only one time.

For every room it should go to all rooms it has not already been to, and that he can reach without going to a room he already went in.

Not every starting point has a solution, for example we can visually see that the Cafeteria (0) has no solution.

Furthermore, as we can see on the map, the order won’t significantly increase the distance, as we are working with distances and need to go to all rooms. Not all rooms are connected, and we can not take any shortcuts. Thus, all Hamiltonian path will have a close travelling time.



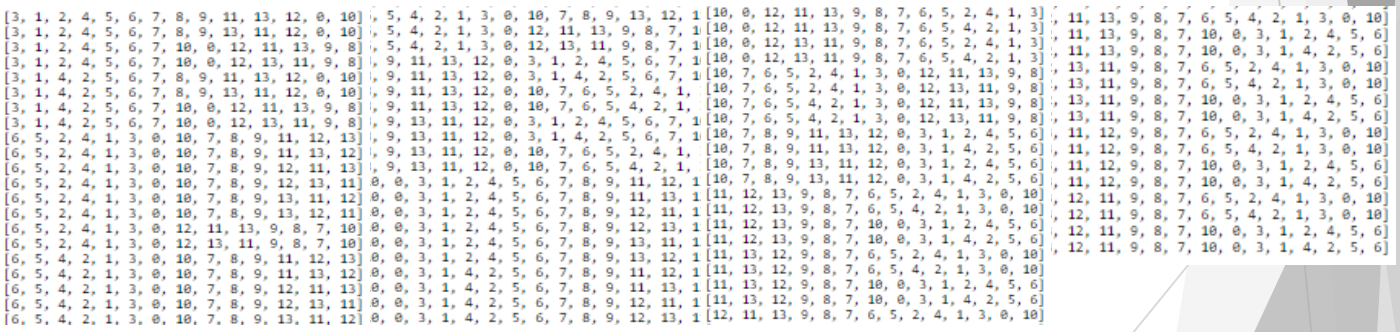
Algorithm and Results :

In order to find the Hamiltonian paths, we'll use the following recursive algorithm :

```
recSearch(graph,start,used):
    if have been to every vertices or every vertices connected to the current on have been visited allready :
        if have been to every vertices :
            display the path
    else :
        for every vertice :
            if the vertice was not already used and is connected to the curent vertice:
                add the vertice to the path
                recSearch(graph,new vertice,used)
for every vertice:
    recSearch(graph,first vertice,[first vertice])
```

For every nodes we try to go to every remaining nodes, so it has a complexity of O(n!)

We find the following 80 Hamiltonian Paths :



The End !

Thank you for Reading !