ADSA mini problem report : Among Us

Step 1

In the first part of the project, our goal is to create a tournament for 100 players with random games and scores attributed to each player. We have to argue about the best possible data structure to store the players and their scores.

To represent a player and its score we will simply create a Python object whose attributes are the player's score and name. We can later add static attributes or class methods in order to fit other needs.

Database

The data structure used to store the players is a bigger decision to make. The 2 obvious choices here would be a list or an AVL tree. Because one of our constraints is to be able to reach a player using its score with a $O(log(n))$ complexity at any time, the list would have to always be sorted (in order to apply a dichotomic search). An AVL tree is by definition always sorted.

When updating the scores for each player, we would need to reorganize the AVL tree, either by making rotations, or by deleting and readding the player with a new score. For 1 player, this operation has a $O(log(n))$ worst case complexity (which corresponds to the height of the tree), so for n players, the complexity of this operation would be $O(n\ log(n))$. This operation of $O(n\ log(n))$ complexity would have to be executed after each game session.

If we used a list, we would have to sort it after each game session, using an efficient sorting algorithm, such as Quick Sort or a Fusion Sort, the complexity would then also be $O(n\ log(n))$

However we need to be careful as when accessing a player's score to update it, we might encounter a higher complexity than expected. After looking at the documentation of the Python lists, we saw that deleting an element with its index is $0(n)$, however lists implement the `pop()` function like stacks, with $O(1)$ complexity.

We also need to consider how we will implement other methods, such as choosing 10 sets of 10 random players to play the 3 first games. This method would at least have a complexity of $O(n)$, since we need to choose the players who will play the game one by one. For an AVL tree we would use a recursive inorder traversal, and using a counter we could change the game for every 10 player.

This implementation is more difficult than one with lists, since we would just need to do 1 shuffle (with $O(n)$ complexity)

After considering it a lot, we found that in terms of pure complexity, it can be equivalent to use a list rather than an AVL (which is as we know a very good choice and one made by almost every student) by redefining some functions and using certain tricks. This also

enables us to rewrite a divide-and-conquer algorithm (fusion sort) and apply dichotomic search and stack-like operations.

Which is why we chose to use a list for our database.

Randomizing players' score

For randomizing the players' scores for each game, we decided to create a Game class, in which we implemented a method which elaborates a game scheme. We looked at some empirical statistics on the internet to know how much games are won by impostors and crewmates, and it appears to be pretty well balanced. The probability to win is actually 1/2 ! Then in each game we created a scenario (random number of crewmates killed in each round, random votes for each players during emergency meetings…) making rich and coherent possibilities of scores for the players ! Look it up in our code.

Update players' score

Since we created a whole class for games and players, when we use the previous randomization method, we can directly update each player's score according to what our function tells us, however that implies that the database will now be shuffled… Which is when our fusion sort comes handy, because we always want to be able to access a player with his score, so with each round of game played we sort the database.

Random games

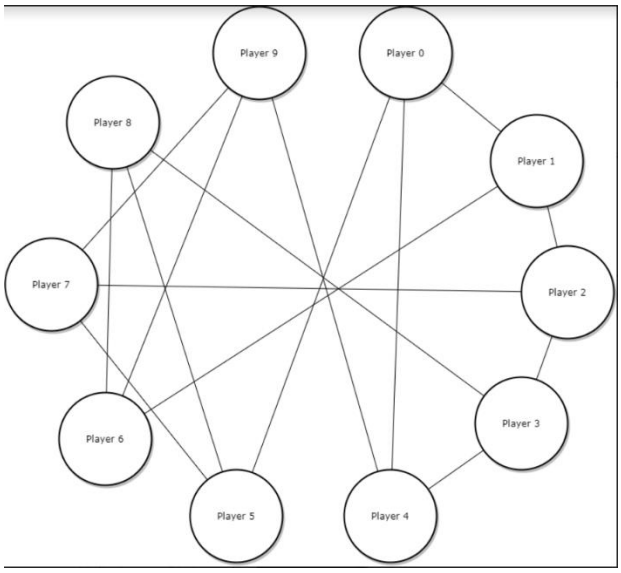
This part would also have been way more difficult if we used an AVL tree, however here we simply can shuffle our list, and take players ten by ten to play a game. After this is done we sort the list again.

Based on ranking the method is the same but without shuffling. To drop players from the database we use the `pop()` function.

Step 2

1. The second step of the project is to describe as a graph the relation between the players. We chose to represent the players as vertices and the relation of seeing each other as edges.

The edges are not weighted, as the players do see each other or they do not. They are neither directed, as a player seeing another implies the other seeing him.



The valuable information here is contained in the edges because we define that an impostor cannot see the other one, and if someone died, one of the players he has seen before dying is an impostor.

This is why we chose to represent the graph as an incidence matrix.

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

2. In the exemple, player 0 is reported dead, having seen player 1, 4 and 5, they may be impostors.

Assuming 1, 4 and 5 have the same probability of being the first imposter, we can define the following events :

I_1: *the event of being the first impostor*

I_2: *the event of being the second impostor*

$I = I_1 \cup I_2$

$P(I = x)$: *the probability of a player x being an impostor*

And the following probabilities :

- $P(I_1 = 1) = 1/3$
- $P(I_1 = 4) = 1/3$
- $P(I_1 = 5) = 1/3$

Therefore, we consider the other impostor has not seen those players.

- If 1 is impostor, 2 and 6 cannot be, which can be translated as :
  - $P(I_2 = 2|I_1 = 1) = 0$ and $P(I_2 = 6|I_1 = 1) = 0$
- If 4 is impostor, 3 and 9 cannot be :
  - $P(I_2 = 3|I_1 = 4) = 0$ and $P(I_2 = 9|I_1 = 4) = 0$
- If 5 is impostor, 7 and 8 cannot be
  - $P(I_2 = 7|I_1 = 5) = 0$ and $P(I_2 = 8|I_1 = 5) = 0$

Let's define sets of possible impostors for each scenario :

- 1 is the first impostor implies that {3,4,5,7,8,9} is the set of probable second impostor
- 4 is the first impostor implies that {1,2,5,6,7,8} is the set of probable second impostor
- 5 is the first impostor implies that {1,2,3,4,6,9} is the set of probable second impostor

Translated in probabilities :

$P(I_2 = x \in \{3,4,5,7,8,9\}|I_1 = 1) = 1$ or $P(I_2 = 3|I_1 = 1) = 1/6$ and $P(I_2 = 4|I_1 = 1) = 1/6$ … $P(I_2 = 9|I_1 = 1) = 1/6$

Let's now focus on the probability of player 1 being an impostor :

$P(I = 1) = P(I_1 = 1 \cup I_2 = 1)$

However player 1 cannot be both impostor 1 and impostor 2, thus the events are incompatible. This means that $P(I_1 = 1 \cup I_2 = 1) = P(I_1 = 1) + P(I_2 = 1)$

We said earlier that $P(I_1 = 1) = 1/3$ because player 1 has seen player 0 before he dies.

Let's now compute $P(I_2 = 1)$:

We know that being impostor 2 implies that impostor 1 is either player 4 or player 5 (which have also seen player 0) but it also implies that player 1 is impostor among the set of probable impostors for each of them :

$P(I_2 = 1) = P(I_1 = 4 \cap I_2 = 1) + P(I_1 = 5 \cap I_2 = 1)$

If player 4 is impostor, which has a $1/3$ probability of happening, player 1 has $1/6$ chance of being the impostor, same for player 5. Mathematially speaking :

$$P(I_2 = 1) = P(I_1 = 4)P(I_2 = 1|I_1 = 4) + P(I_1 = 5)P(I_2 = 1|I_1 = 5) = \frac{1}{3}\frac{1}{6} + \frac{1}{3}\frac{1}{6}$$

Thus $P(I_2 = 1) = \frac{1}{9}$

So the probability of player 1 being an impostor is :

$$P(I = 1) = \frac{1}{3} + \frac{1}{9} = \frac{4}{9}$$

We can have the same reasoning for all the other players, and obtain :

$$P(I = 4) = \frac{4}{9} P(I = 5) = \frac{4}{9} P(I = 2) = \frac{1}{9} P(I = 3) = \frac{1}{9} P(I = 6) = \frac{1}{9} P(I = 7) = \frac{1}{9} P(I = 8) = \frac{1}{9} P(I = 9) = \frac{1}{9}$$

We notice that $P(\cup_{x=1}^{9} I = x) = \sum_{x=1}^{9} P\,(I = x) = 2$ which makes sense as there are 2 impostors.

We cannot deduce the impostors from this example, however if 2 players who have seen player 0 had also seen each other, we could have deduced that they could not both be impostors, and then obtain different probabilities. Also if 2 players who have seen the dead player had seen the same players, the probability for those to be impostor 2 would have been less.

3. We implemented the algorithm computing these probabilities, we also implemented an algorithm to generate random has-seen graphs to test the algorithm in numerous combinations of player relations (see the code for implementation details)

4. The results showed are the same as displayed above

Step 3

1. First, we need to represent the map for the impostors and the crewmates. We will use weighted graphs, with the rooms' center represented by vertices, and the corridors with undirected edges (because it takes the same time to go from one to another and vice versa). The time to travel between 2 rooms is represented by the weight of the edge between the 2 nodes. This works fine for crewmates, but for impostors, we also have to take the vents into accout.

Most vents are located in a room, so we can simply add an edge between the centers of the rooms connected by a vent. There is one exception however, with the vent located in a corridor. The first option is to add a node representing the vent to the graph, with all the connexions that implies (edges between the vent and all rooms next to that corridor). The other is to directly change the edges values to take that vent into account. By doing that, we lose some information, but as we are trying to compute the shortest path, the end result will be the same (and it is much faster for us to write)

We wrote the graphs' adjacency matrixes in .csv files that we import into our program. We made 2 versions, one using the pandas library and one without. It was easier to visualize with pandas (with titles for rows and columns), but since we did not make the library, we also wanted a version without it. In the end we only kept the version without pandas, in case the user of the program does not have it installed.

2. Now that we have our graphs, we need to compute the shortest paths. We want to find out all the shortest paths between each pair of vertices, so we chose to use the Floyd Warshall algorithm. We modify the adjacency matrix of the graph to fill it with all shortest paths.

3. The method for Floyd Warshall is as follows :

```
def Floyd_warshall(mat: List[List[float]]):
    """
    This method modifies the adjacency matrix, replacing the edges with the
shortest path for every
    pair of vertices.

    Parameters
    ----------
    mat : List[List[float]]
        Adjacency matrix.

    Returns
    -------
    None.

    """
    for k in range(14):
        for i in range(14):
            for j in range(14):
                if mat[i][j] > mat[i][k] + mat[k][j]:
                    mat[i][j] = mat[i][k] + mat[k][j]
```

4. The display method is the following :

```
def print_all_paths_fw():
    mat_cm, mat_im = import_graph()
    Floyd_warshall(mat_cm)
    Floyd_warshall(mat_im)

    for i in range(14):
        for j in range(14):
            print(rooms[i],rooms[j],'\nfor crewmates :',
                  mat_cm[i][j],'seconds\nfor impostors :',
mat_im[i][j],'seconds\n')
```

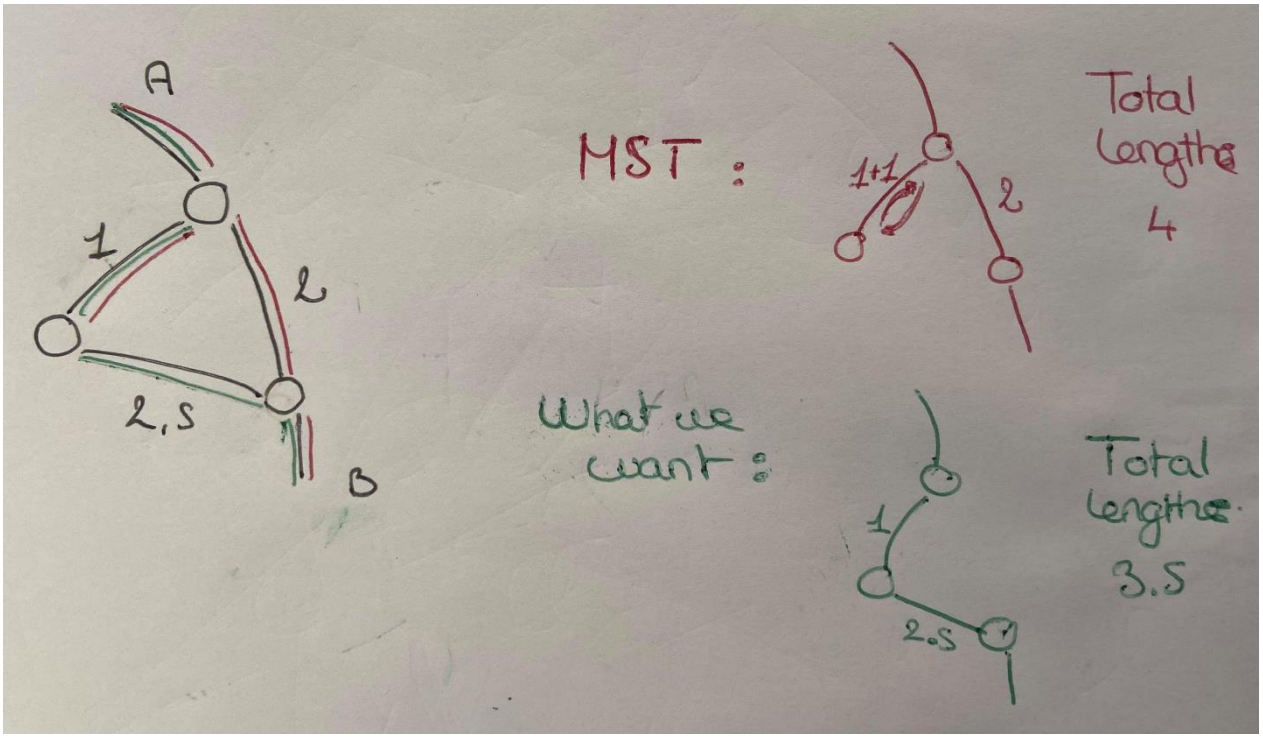
Step 4

First thoughts on the problem

In the last step, the players all stay together to finish the tasks without anyone being killed by the impostor. They have to go through all rooms as fast as possible, we then wish to find the shortest path linking all rooms together. For this purpose we can use the crewmates graph we created previously for step 3.

This looks a lot like a MST problem, but we figured out that there may be a slight difference.

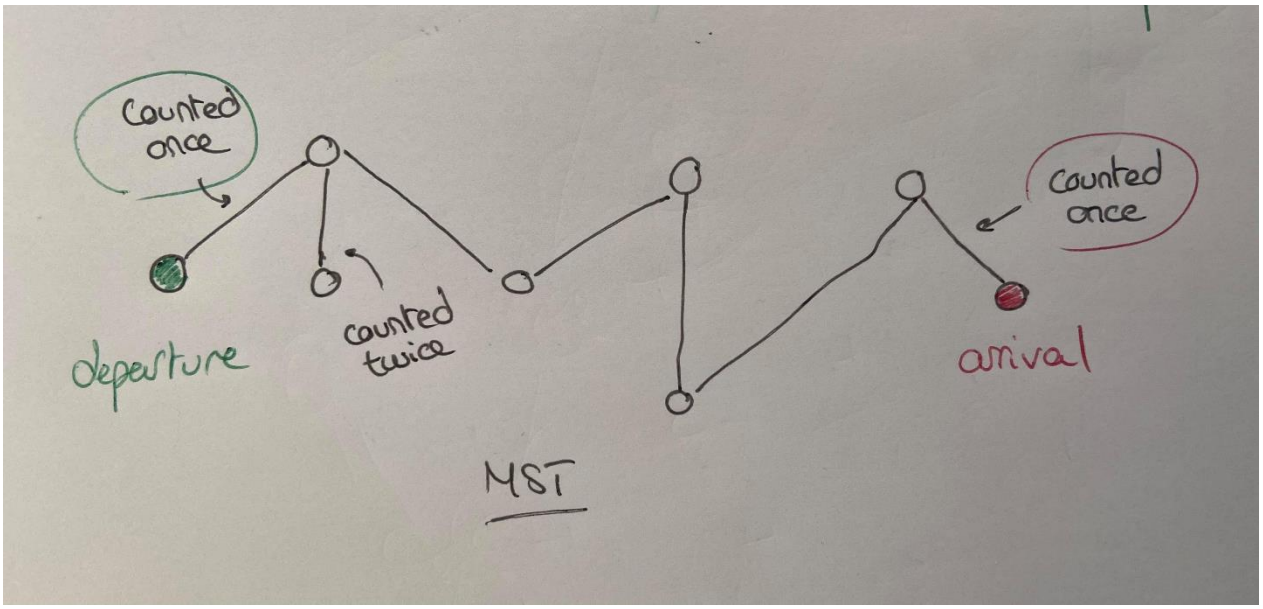
In a MST algorithm, we are trying to find the path which links all nodes together with a minimum total length. However here, it is not actually the total length we are trying to minimize… Let us look at a quick example to illustrate :



In this image, we see 3 nodes (rooms) and their respective distance to another. In a minimum spanning tree, as we seek to minimize total length, we would link the three rooms with the 2 shortest edges (1 and 2) resulting in a total length of 3. However if we were crewmates trying to go from A to B passing through each room as fast as possible, we would have to turn around after visiting the bottom-left room, thus traversing the 1-length edge twice. The total length would then become 4 ! If we rather had used the 2.5-length edge, we could have gone through all rooms with a total distance of 3.5, which is better.

Here the problem is that each time a node is linked to the rest of the MST by only 1 edge, visiting it requires to go back and forth, thus counting the edge twice. In some cases this has to be done anyway, but in others like in the previous example, there may be a better solution.

The 2 exceptions to this rule are the starting and arrival node of the crewmates. In our concrete case, the starting point will always be the cafeteria, but we cannot know for sure about the arrival before computing the algorithm, see the figure below.



What we might actually want, if it exists, is closer to a Hamiltonian path. However, in our graph it is impossible to go in each room only once, and even if it were, we could not know for sure that this is the best possible path (without computing all such paths' lengths).

EDIT : After re-reading the problem carefully, we noticed that what is actually asked, is to go through each room **only once**. We thought at first that this was impossible since in order to decide such manoeuvre, the players would need to talk about it during an emergency meeting, hence *starting from the cafeteria*. However, if the players were to start in the room they wanted, a Hamiltonian path does exist. Furthermore, it is not explicitly written that we need to find the shortest path of this kind, we just have to show one that exists. (However because the final goal is to go as fast as possible through all rooms, we considered relevant to compute the shortest Hamiltonian path anyway)

What we decided

1. After examining those new elements, it becomes clear that we actually need to find a Hamiltonian path. Since we do not need to take into account the travel time between rooms, we could use an unweighted graph representing all rooms and connexions for crewmates. However we already have a weighted graph from step 3, and because we might want to compute which of the Hamiltonian paths is the shortest, we decide to use it to represent the map.

2. The graph theory problem here is to find a Hamiltonian path in the graph, a path passing through all nodes only **once**.

3. To solve this problem, our first instinct is to start from a node, and go from one to another using depth-first search through the connecting edges, adding each node in a stack. When a node is in the stack, it cannot be added again, if there is no more node linked to the last in stack by an edge, we backtrack by popping it and trying with another one. If the stack gets to the actual number of nodes, it means we have found a Hamiltonian path, and can retrieve the stack.

This method is very intuitive, as it is probably the closest to the way a human would do (even though the human would have some heuristic to choose the best next node) however for a computer it represents a very high complexity. We would have to try starting from all nodes, and in the worst case, the algorithm has a $O(n!)$ complexity. Which means overall it has a $O(n * n!)$ complexity.

Another algorithm for solving this problem that we looked for is the Held-Karp algorithm, which uses dynamic programming and is more difficult to understand. It has a time complexity of $O(2^n n^2)$ but also a higher space complexity. Since the last is not very intuitive, and that in our case we do not have that much nodes, we will opt for the DFS with backtracking algorithm.

4. The implemented algorithm showed us that there are 5 equivalent shortest Hamiltonian paths, for example [5, 7, 6, 4, 3, 2, 1, 0, 13, 12, 10, 11, 9, 8] which corresponds to : 1. mid_room 2. storage 3. right_room 4. shield 5. navigation 6. O2 7. weapons 8. cafetaria 9. medbay 10. upper_e 11. security 12. reactor 13. lower_e 14. electrical

Of course, this is relative to the measures we did for each edge during step 3.

Authors : Paul JOUËT and Aladin HOMSY

Group : DIA3