Among Us Project: report of theorical study

Océane Görke & Michelle Hatoum DIA3

Step 1: To organize the tournament

1. Data structure to represent a Player and its Score
   Each player is going to be represented by a node that contains the following attributes (self, player, score=None). The left and right nodes are going to be initialized to None, as well as the height to 1.

2. The most optimized data structures for the tournament
   The database will be an AVL-Tree. This choice has been made because every operation on the tree runs with a **log complexity**. Moreover, the inorder traversal of an AVL-Tree allows to get the scores in ascending order.

3. Method that randomize player score
   In order to test each method, a random assignment of scores will be needed. This is going to be done by the method ***randScore****(liste)* outside the AVLTree class. This method takes every node from the input list and generates a random number between 0 and 12 in order to add it to the previous existing score. If the score was None, addition isn't supported between NoneObjects and integers so the score will be changed to 0 before being randomized. The method returns the list with the new scores.

4. Method to update Players score and the database
   Since there is no real game, the update of scores will be done by the ***randScore*** method that will add a score between 0 and 12 to each player's score after the false game is over.
   To update the database, the list returned by ***randScore*** will be transformed into an AVLTree with the use of method ***avlFromList****(self, liste, node=None)* . This method sets *self* to a new and void AVLTree and uses the method ***insert****(self, root, key, player=None)* to insert every node to *self* . It returns *self*, an updated and balanced AVLTree.

5. Method to create random games based on the database
   The first game will be based on the database because there is no ranking yet. Then, after that the ranking will have an importance. This is where the method ***createGames****(listplayers, rand=False)* is necessary. If rand is True, it means we need a random game, the method will create a new list, a copy of the players list and will choose randomly groups of 10 players and affect them to a certain game.

6. Method to create games based on ranking
   Using the same method, if rand is False, we need the ranking, so the method just uses the sorted list (thanks to the inorder traversal of our tree) and chooses the players in order in groups of 10.

7. Method to drop the players and to play game until the last 10 players

   The method ***dropPlayers****(listplayers)* only takes a list and deletes the first 10 players.
   To play game until there is only 10 remaining players, this will be implemented with a while loop inside the ***game*** method that we will discuss in the 9th point.

8. Method which display the TOP10 players and the podium after the final game

   displayFinals*(liste)* makes sure there is only 10 players left in the list that we made from our AVLTree and prints the name of each one.
   For the podium, the method ***podium****(liste)* only uses 3 prints to print the name of the 3 players with the highest scores.

9. Playing the game

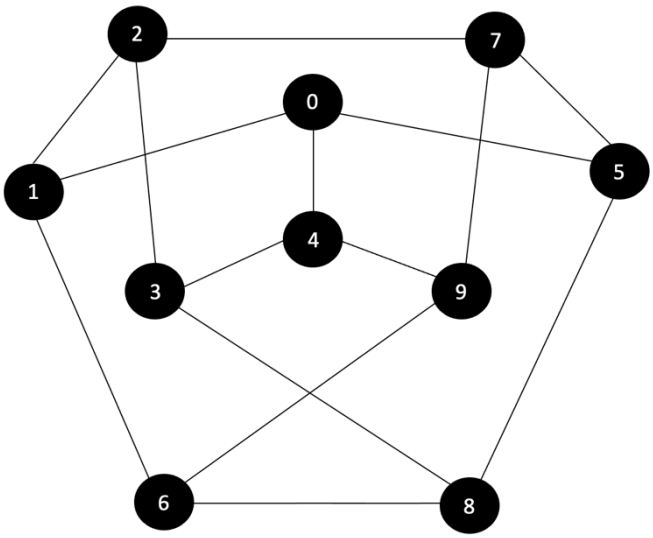
   The ***game****(nb, rand=True)* method takes the number of players in the tournament and a Boolean that tells if the player's names are going to be entered in the shell by the user or if the algorithm can generate random strings as names.
   Once there are nb players created and added to a players list, a void AVLTree (gameTree) is instantiated.
   The first part of the game will be in a while loop, checking at the end of every round if there are more than 10 remaining players.
   First, the number of players is checked so the method knows if this is the first round, thus, if it requires a random game or a ranking based game.
   Then, it prints the game groups and the players of every group.
   Once the games are "over", the players list is emptied and ***randScore*** is called so the players are attributed their first score if it is the first game, or their scores are randomly increased if not. At the same time, each player is appended to the players list.
   Of course, after that, gameTree is updated with the new scores and players list is filled all over again thanks to the ***fillList*** method. Now, gameTree and the list are updated and ranked, almost ready for the next round. To finish off the round, ***dropPlayers*** is called to eject the 10 worst players off the game.
   When there are 10 remaining players, first their names and scores are displayed. Then, the game is played, scores are increased, gameTree and list are updated and finally, ***podium*** is called to announce the best 3 players.

Why did we use lists ?

As you can notice above, lists are vey present in this first step. We took this decision because it made our work so much easier and it helped with running time. Indeed, if we wanted to delete and update node's values directly from the tree it would have been much more time consuming. Please note that our main database is still an AVL Tree and that the trees are the reason why this whole step can work properly, list are only a support for our work.

Step 2: Professor Layton < Guybrush Threepwood < You

1. Represent the relation (have seen) between players as a graph, argue about your model.
   First of all, we modeled a graph on which we translate all the "have seen" relations. If two players are linked by an edge it means that they have seen each other. It looked like this:



   Nevertheless, using this model, we couldn't really see who the possible impostors were.

   The first idea that we had was supposing that 1, 4 or 5 was one impostor and so we supposed that all the players he didn't see was the second impostor. In this way, if we suppose that 1 is the impostor, the player who has killed 0, then 2 and 6 can't be the other impostor since they have seen 1 and both impostors never walk together. Moreover, we could clearly remark that 1, 4 and 5 didn't see each other. So if 1 is the impostor, 4 and 5 could be one too and likewise for the rest of the players.

   In a nutshell, 0 is dead. We suppose that one of the players he has seen is the impostor, which excludes the players this suspected impostor has seen. And all the other remaining players are suspected to be the other impostor. It didn't really fit the problem and neither give a solution. The crewmate players couldn't really know against who they had to vote.

2. Thanks to a graph theory problem, present how to find a set of probable impostors.
   In order to resolve the problem, we chose to use a graph coloration problem. Precisely, we used the Welsh-Powell algorithm. Nevertheless, since all the nodes have the same degree (they are all linked to 3 nodes), this algorithm didn't help to find a solution. Indeed, it sorts the nodes according to their degree but here there is no interest, so we chose to use the function shuffle() imported from the random module. It permitted to sort the representation of our graph in random ways.
   Then, we did some probabilities according to the color associated to each node.

3. Argue about an algorithm solving your problem.

As you could have seen in the first question, the first step of our resolution was to search for the neighbors of each node and eliminate some that couldn't be impostors. This is how we did it on python :

The list connections gathered tuples representing a player and the other player he has seen.

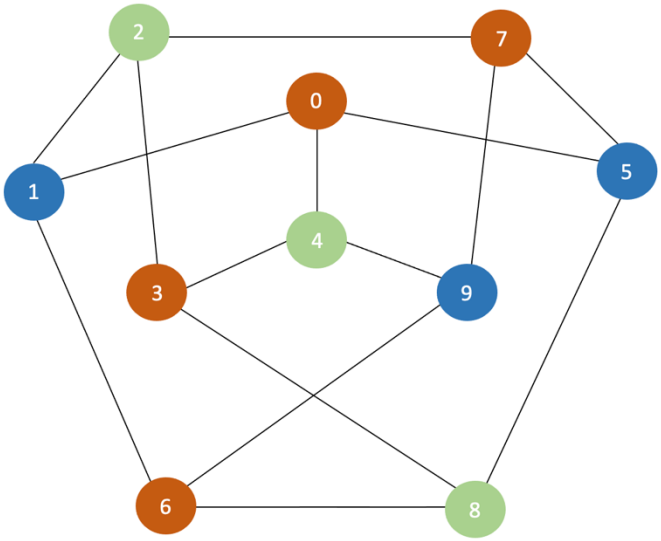
We define a class Graph which uses a dictionary to group the whole of the nodes. To each of them it is associated a list of its neighbors, meaning the players he has seen.

Then we define a function ***get_neighbors***(self, node) allowing to get the neighbors of each node. Finally, the function ***imposteur***(self, impostor=None) permits to give a set of the possible impostors once we have supposed 1, 4 or 5 was one of them. Indeed, we are sure that at least one of them is an impostor because 0 has seen them before being killed.

Nonetheless, we weren't satisfied of this resolution, so we kept digging.

This is where we decided to use a graph coloration problem. As said in the previous question, we used random and probabilities. We are going to explain how.

First of all, we have the function ***fMatAdj***(n,LA) which creates the adjacent matrix corresponding to the model we discussed above. Then the ***WP***(n,LA) function is the Welsh-Powell algorithm we use to associate a color {0,1,2} to each node. For example, we obtain this 3-coloration graph :



The algorithm sometimes gave a 4-coloration graph but since a 3-coloration one is efficient in our problem we decided to not take the 4-coloration into consideration.

Once our graph is colored, we first created a ***freq_impostor***() in which the point was to isolate one of the first impostors: {1,4,5} thanks to one color and study the other nodes that have the same coloration. We thought about this because the two impostors don't walk together in a game so they haven't seen each other and so may have the same color.

First of all, we studied the case where the first impostors doesn't have the same coloration. Indeed if they have the same coloration, we can't conclude anything, so this is not interesting. We then implemented a frequency matrix regrouping the total number of times a color has been associated to each node. For each WP result, we only studied the color that was associated to only one of the first impostors. As we said before, the other impostor may have the same coloration as the isolated first impostor, so we didn't need to take the other colors into account.

We made this estimation to 1 000 games in order to have the probability the most accurate. We didn't try with more games because even 1 000 games took a few minutes. If you want to test the code and make it quick, you can change the number of games directly on line 129.

Secondly we implemented the function ***guess_impostor***() whom the aim is to find the two impostors. In order to do this, we proceeded in 3 steps. The first step consists in cleaning the freq matrix and keep only the maximum of the column 1, 4 and 5. We did this because we really wanted to focus on the most probable impostor and the other freq wouldn't be used in any cases. The second step is so to determine each of 1,4 or 5 will finally be retained to be one of the two final impostors. Again, we only kept the maximum of them, and we also kept the color associated. This is where comes the third and last step : apart from player we designated at the previous step, we looked for the maximum of the line. This is where there is the most of chances to find the second impostor.

And finally, we created an ***estimate_impostor***() function in order to strengthen our estimation. We realize the ***guess_impostor()*** 50 times and increase the number of appearances of each designated impostor in the dedicated list. Taking the two maxima in this list, this is how we find the two most probable impostors.

4. Implement the algorithm and show a solution.

According to our algorithm's result, we found that the first impostor was 4 and the second one may be 8. But we can see on the returned list that 8 is the player that has been the most designated by ***guess_impostor***(). Nonetheless, we should take note that it has just one more appearance than player 2 and only 2 more than player 6.

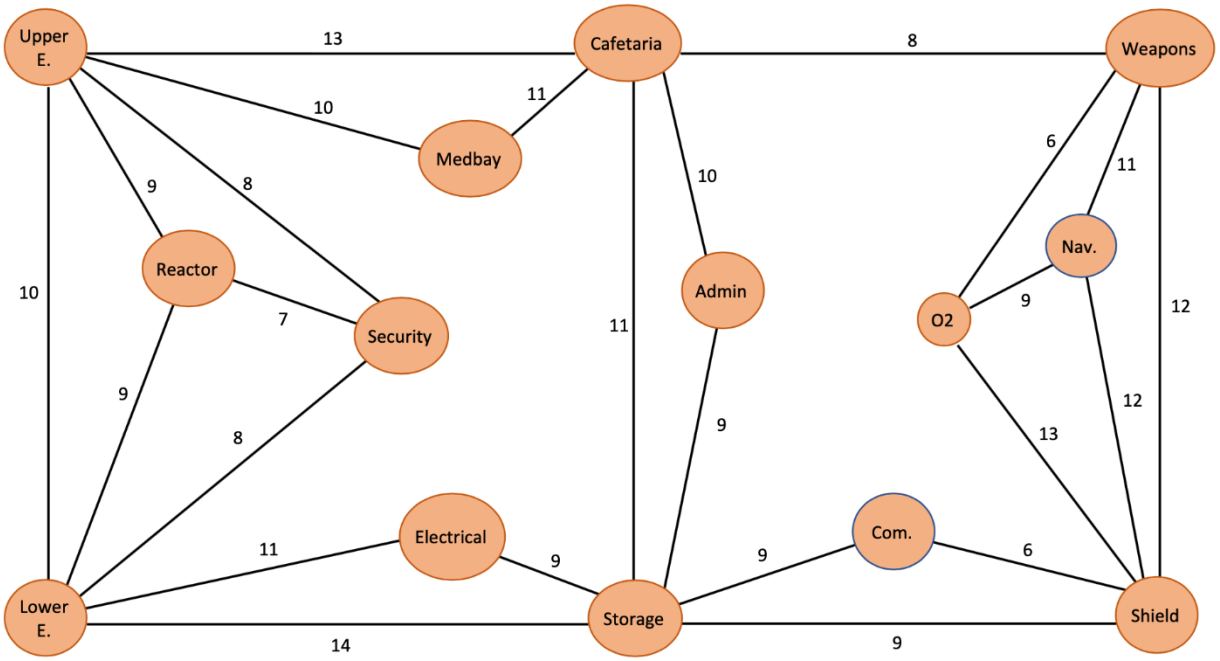
This resolution is based on probabilities but there is nothing totally sure.

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

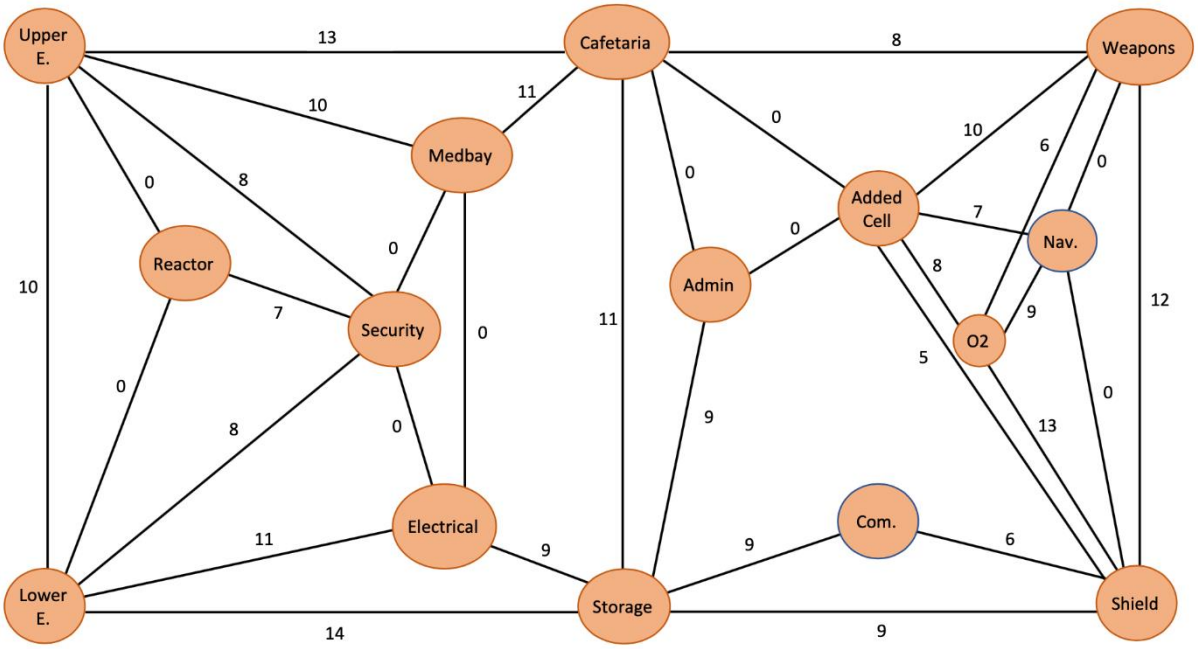
1. The two models of the map

   The map will be created in two ways, the first one is the crewmates one, and the second is the one with vents and modified itineraries that only impostors can use.

   Crewmates map:



   Impostors map:



2. Pathfinding algorithm

   The best algorithm to use in this case is Floyd-Warshall's. This algorithm allows to see the shortest distance between every point which is exactly what is needed.

The initial board for crewmates is:

	Upper E	Lower E	Reactor	Security	Medbay	Electrical	Cafetaria	Storage	Admin	O2	Com	Weapons	Navigation	Shield
Upper E	0	10	9	8	10		13							
Lower E	10	0	9	8		11		14						
Reactor	9	9	0	7										
Security	8	8	7	0										
Medbay	10				0		11							
Electrical		11				0		9						
Cafetaria	13				11		0	11	10			8		
Storage		14				9	11	0	9		9			9
Admin							10	9	0					
O2										0		6	9	13
Com								9			0			6
Weapons							8			6		0	11	12
Navigation										9		11	0	12
Shield								9		13	6	12	12	0

For impostors, the orange cells are the ones that have been modified or created compared to the table above.

	Upper E	Lower E	Reactor	Security	Medbay	Electrical	Cafetaria	Storage	Admin	O2	Com	Weapons	Navigation	Shield	Added Cel
Upper E	0	10	0	8	10		13								
Lower E	10	0	0	8		11		14							
Reactor	0	0	0	7											
Security	8	8	7	0	0	0									
Medbay	10			0	0	0	11								
Electrical		11		0	0	0		9							
Cafetaria	13				11		0	11	0			8			0
Storage		14				9	11	0	9		9			9	
Admin							0	9	0						0
O2										0		6	9	13	8
Com								9			0			6	
Weapons							8			6		0	0	12	10
Navigation										9		0	0	0	7
Shield								9		13	6	12	0	0	5
Added Cel							0		0	8		10	7	5	0

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

   The method we use is called ***shortestroute(***graph). It takes a list of lists and returns the completed list of lists thanks to Floyd-Warshall's algorithm.

   Indeed, it returns the full table of time between every pair of room but the only problem is that for readability purpose, we used a Dataframe to display our result table but it is too large for the shell so there are the completed tables including the columns that are not visible on python:

Crewmate's traveling time:

	Upper E	Lower E	Reactor	Security	Medbay	Electrical	Cafetaria	Storage	Admin	O2	Com	Weapons	Navigation	Shield
Upper E	0	10	9	8	10	21	13	24	23	27	33	21	32	33
Lower E	10	0	9	8	20	11	23	14	23	36	23	31	35	23
Reactor	9	9	0	7	19	20	22	23	32	36	32	30	41	32
Security	8	8	7	0	18	19	21	22	31	35	31	29	40	31
Medbay	10	20	19	18	0	31	11	22	21	25	31	19	20	31
Electrical	21	11	20	19	31	0	20	9	18	31	18	28	30	18
Cafetaria	13	23	22	21	11	20	0	11	10	14	20	8	19	20
Storage	24	14	23	22	22	9	11	0	9	22	9	19	21	9
Admin	23	23	32	31	21	18	10	9	0	24	18	18	29	18
O2	27	36	36	35	25	31	14	22	24	0	19	6	9	13
Com	33	23	32	31	31	18	20	9	18	19	0	18	18	6
Weapons	21	31	30	29	19	28	8	19	18	6	18	0	11	12
Navigation	32	35	41	40	30	30	19	21	29	9	18	11	0	12
Shield	33	23	32	31	31	18	20	9	18	13	6	12	12	0

Impostor's traveling time:

	Upper E	Lower E	Reactor	Security	Medbay	Electrical	Cafetaria	Storage	Admin	O2	Com	Weapons	Navigation	Shield	Added Cel
Upper E	0	0	0	7	7	7	13	14	13	21	23	18	18	18	13
Lower E	0	0	0	7	7	7	13	14	13	21	23	18	18	18	13
Reactor	0	0	0	7	7	7	13	14	13	21	23	18	18	18	13
Security	7	7	7	0	0	0	11	9	11	19	18	16	16	16	11
Medbay	7	7	7	0	0	0	11	9	11	19	18	16	16	16	11
Electrical	7	7	7	0	0	0	11	9	11	19	18	16	16	16	11
Cafetaria	13	13	13	11	11	11	0	9	0	8	11	5	5	5	0
Storage	14	14	14	9	9	9	9	0	9	15	9	9	9	9	9
Admin	13	13	13	11	11	11	0	9	0	8	11	5	5	5	0
O2	21	21	21	19	19	19	8	15	8	0	12	6	6	6	8
Com	23	23	23	18	18	18	11	9	11	12	0	6	6	6	11
Weapons	18	18	18	16	16	16	5	9	5	6	6	0	0	0	5
Navigation	18	18	18	16	16	16	5	9	5	6	6	0	0	0	5
Shield	18	18	18	16	16	16	5	9	5	6	6	0	0	0	5
Added Cel	13	13	13	11	11	11	0	9	0	8	11	5	5	5	0

4. <u>Display the interval of time for each pair of room where the traveler is an impostor</u>

We created the method ***compare***(graph1,graph2) to answer this question. This method takes 2 (completed) graphs and prints a sentence comparing the traveling time from a room to another for crewmates and impostors.

We decided to use this method in a while loop for readability purpose. Indeed, if we wanted to print a sentence for every itinerary possible it would be hard to read because there will be a lot of lines. The main method ***interface*** will ask the user if they want to compare an itinerary and will stop calling ***compare*** when the user answers "no".

		Upper E	Lower E	Reactor	Security	Medbay	Electrical	Cafetaria	Storage	Admin	O2	Com	Weapons	Navigation	Shield
		4	5	0	7	2	0	8	5	1	3	0	6	0	3
Upper E	4	INF	4.5	2	5.5	3		6							
Lower E	5	4.5	INF	2.5	6		2.5		5						
Reactor	0	2	2.5	INF	3.5										
Security	7	5.5	6	3.5	INF										
Medbay	2	3				INF		5							
Electrical	0		2.5				INF		2.5						
Cafetaria	8	6				5		INF	6.5	4.5			7		
Storage	5		5				2.5	6.5	INF	3		2.5			4
Admin	1							4.5	3	INF					
O2	3										INF		4.5	1.5	3
Com	0								2.5			INF			1.5
Weapons	6							7			4.5		INF	3	4.5
Navigation	0										1.5		3	INF	1.5
Shield	3								4		3	1.5	4.5	1.5	INF