Vermeulen Barthélemy
Villeneuve Matthis

ADSA Project:

Table of Contents:

Step 1 : Data structures

Step 2 : Graph colouring

Step 3 : Floyd-Warshall

Step 4 : Hamiltonian Path

Note: On each step, we wrote the name of the python file containing the code related to the aforementioned step.

Vermeulen Barthélemy
Villeneuve Matthis

Step 1: Data Structure.py

Hypothesis:

The subject was unclear on the way the games were supposed to happen, 3 games would not be enough to objectively sort the players, so we took the liberty to organize it as 3 games PER player, meaning 30 games for the first round, then removing the 10 lowest ranked players, leaving 90 players to compete then 27 games and so on until only 10 players remained, after which 5 games are playing, and then the final leader board is printed.

1.

Player: A player is represented by a name, his role, which changes in each game (for the randomization, role won't be used, but still be created to respect the fidelity of a normal among us game).

A player also has a gamecount and a score.

The score is updated each game a player plays in: the score is the average, ranging from 0 to 12, in the game initialization, the score is calculated this way:

Score=average score based on the gamecount

Temporary score=Score*Gamecount <- TemporaryScore is now a number way above 12, the total score.

NewGameCount=GameCount+1 <- This update the game count with the newest game.

Score=(TemporaryScore+NewScore)/NewGamecount <- This updates Score, by making it the new average with the new game counted as well.

2.

Tournament: A tournament is a class with a name, and a playerlist, a list composed of all the players attending the tournament, in our representation, the playerlist consists of 100 players.

The following methods are in the tournament class:

printPlayers() printing the player list with each player's attributes: Name, Score and GameCount.

dropPlayers(): Drops the 10 worst players regarding their score, useful for the tournament.

resetPlayers(): Is used to reset the gamecount and score of the players, between each phases of the tournament.

orderRank(): ranks the list depending on the score of each player.

printLeaderBoard(): Prints the list of the top 10, is used at the end of the final phase of a tournament.

3.

Randomize a game: A game is represented as a class, and has a playerlist, it is supposed to contain 10 players and we have two methods:

Randomize(): A method that, for each player in the list, updates its total score by adding one game and a random score, between 0 and 12.

4.

Update player score and the database: The player score is updated in each game following the formula detailed in the (1.), and thus the database is updated after each game, but in order to order the database, it is necessary to call the tournament.orderRank() method, this is done in our "main".

5.

Create Random games: This can be found in the GamePhase(tournament) method:

We create an empty list which will the contain the players that are yet to play, so in the very first iteration, this contains 100 players, and at the end of the phases, it should only consist of 10 players. Then we select random players to compete (using random.choices, with k=10, the number of player per game). And in the method, we do so to randomize it and so that each player plays three games, with random opponents.

6.

Create Rank Based games: This can be found in the GamePhaseByRanking(tournament) method:

Similar to the GamePhase(tournament) method, except that to play the games, we first select the players by their ranking (so for 100 players, the games are: 1-10, 11-20 and so on). This means that the first game is somewhat random since there is no previous score, but from the second game, each group of 10 player are matched by their ranking as we refresh the rank each time every player played the same number of games (1, then 2,then 3) after which we drop the 10 lowest ranked players and restart the method, until there are only 10 players left (the 10 iterations are to be coded in the main, as you can see under the comment "#Resolving the tournament" but replacing GamePhase by GamePhaseByRanking.

7.

Drop players and play the game until only 10 players remain: This is implemented in the main part of the code: The drop players method is explained in the database (tournament class) explanation.

As for the game playing: This can be seen in the main and goes this way:

Playing of one gamephase, which means every player has played 3 games;

Printing the players and scores to see what happened;

Dropping the 10 lowest ranked players;

Resetting each remaining player score and gamecount to 0;

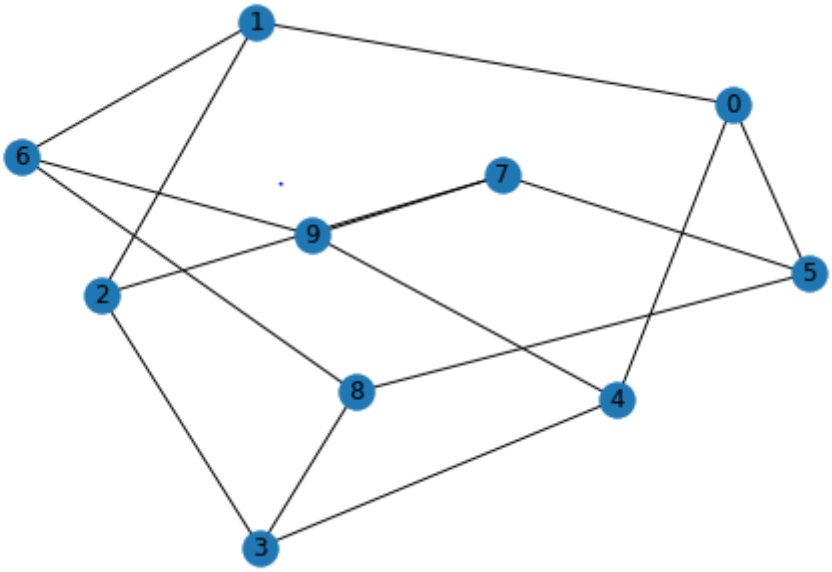
Then repeating the process, a total of 9 times, which means 9*10=90 players are dropped in total.

8.

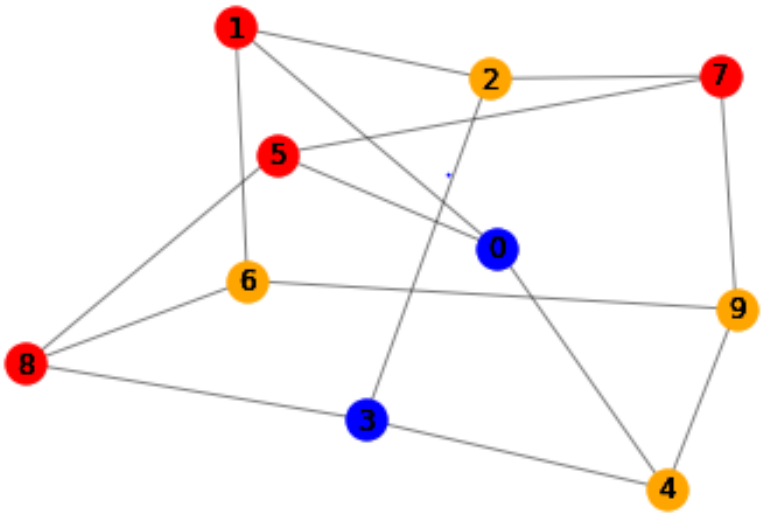
Final Phase: We then enter the final phase, calling the finalgame() method, which simulates the last 5 games with the remaining 10 players (the finalists) , we then order the finalists, and print the top 10, by emphasizing the top 3, with the printLeaderBoard() method.

Step 2: Impostors_Problem.py

1. We represent the relations with a simple, undirected graph because a player seeing another means both saw each other. Each information is equally important so weight would be of no use.



2. Graph colouring is a way to help us identify the probable set of impostor: it colours the players that have not seen each other, thus the probable second impostor, but this algorithm only gives us one possibility per iteration, so it is not sufficient to fully present every probable impostor duo.



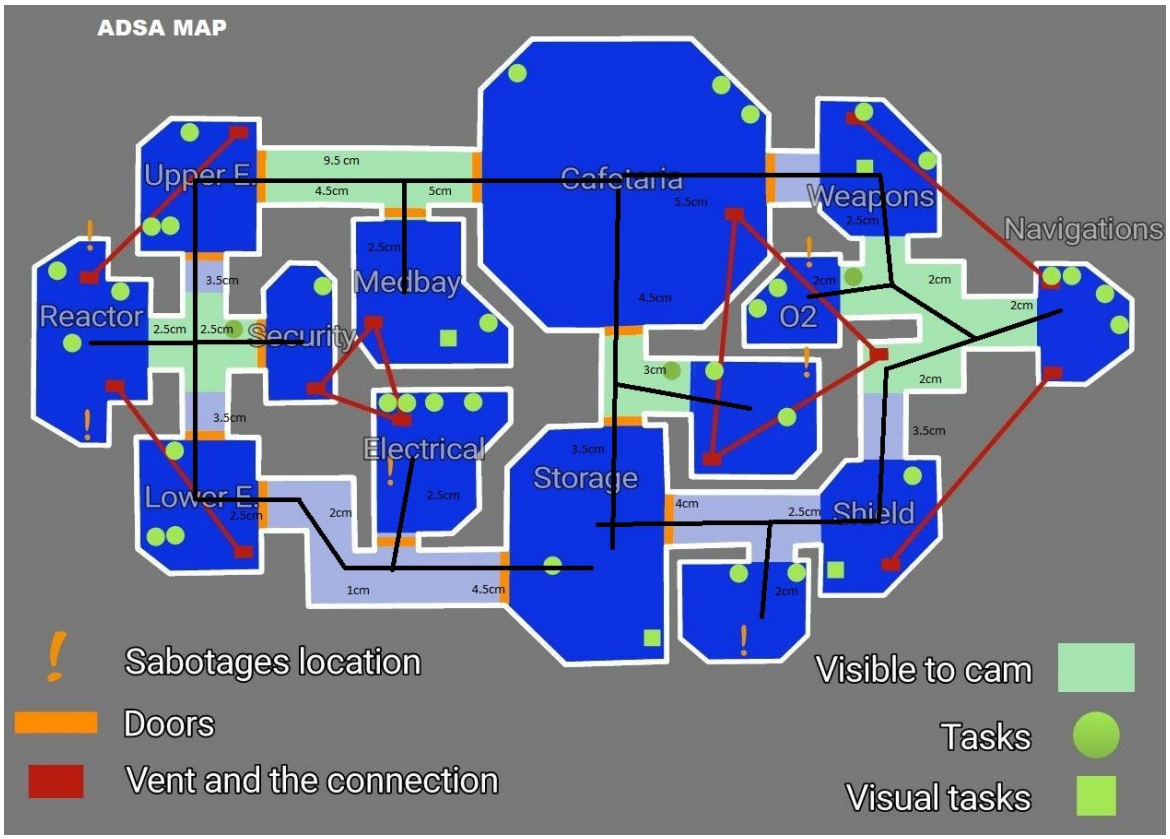
3. We decided to use the Ortools library to implement an algorithm solving our problem and display a set of probable impostors (see the function SecondImpostor) which displays the possible second impostor considering which of 1, 4 or 5 was the first one.
Here are the constraints we modelled:

```
def SecondImposter(x):
    #Creates model

    model=cp_model.CpModel()
    #Creates variables
    num_vals=10
    y=model.NewIntVar(1,num_vals-1,'y')


    #Creates the constraints
    model.Add(y!=x)

    if x==1:
        model.Add(y!=2)
        model.Add(y!=6)
    if x==4:
        model.Add(y!=3)
        model.Add(y!=9)
    if x==5:
        model.Add(y!=7)
        model.Add(y!=8)


    #Creates a solver and solves the model
    solver=cp_model.CpSolver()
    solution_printer = VarArraySolutionPrinter([y])
    status=solver.SearchForAllSolutions(model, solution_printer)
```

4. Finally, there is the 'PossibleImpostor()' method, displaying every possible set of impostors, along with the probability of the set being the real one.

```
def PossibleImposter():
    print("if 1 is imposter, the other imposter could be")
    print()
    SecondImposter(1)
    print()
    print("if 4 is imposter, the other imposter could be")
    print()
    SecondImposter(4)
    print()
    print("if 5 is imposter, the other imposter could be")
    print()
    SecondImposter(5).
    print()
    print("So there are 15 set of possible imposters:")
    print()
    print("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")
    print("Each set has 1 chance out of 18 of being the right set of imposters, exept  1:4  1:5  and  4:5  they have 1 chance ou
t of 9")
```

Step 3: Map_problem.py



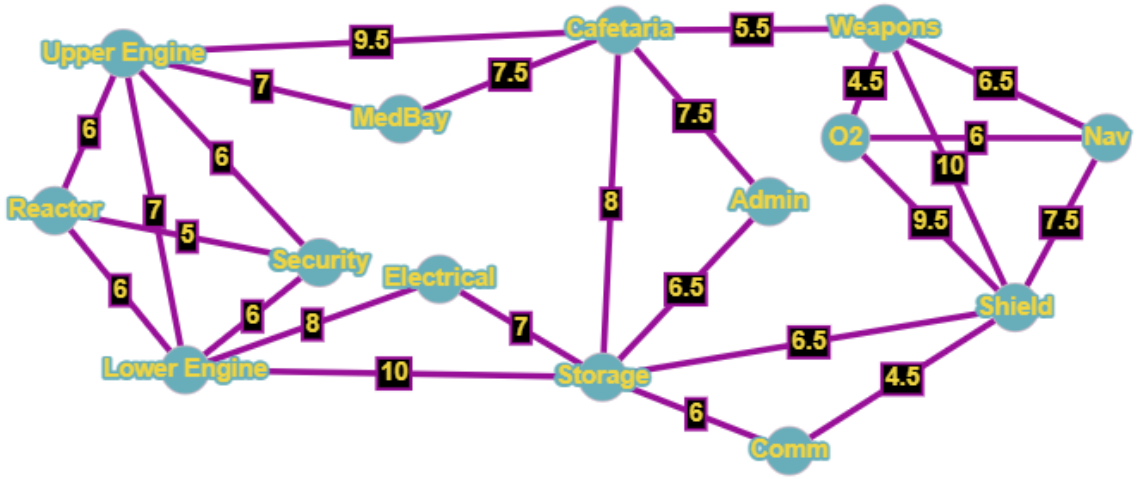
We measured the map roughly, with a ruler using the full screen image on a computer, here are the distances (in cm, Left: Crewmates, Right: Impostors):

Cafet -> Upper E: 9.5 Cafet -> Weapons: 5.5 Cafet -> Admin: 7.5 Cafet -> Storage: 8 Cafet -> MedBay: 7.5 Storage -> Admin: 6.5 Upper Engine -> MedBay: 7 Weapons -> O2: 4.5 Weapons -> Nav: 6.5 Weapons -> Shield: 10 O2 -> Nav: 6 O2 -> Shield: 9.5 Nav -> Shield: 7.5 Shield -> Comm: 4.5 Shield-> Storage: 6.5 Comm -> Storage: 6 Storage -> Electrical: 7 Storage -> Lower Engine: 10 Lower Engine -> Electrical: 8 Upper Engine -> Lower Engine: 7 Lower Engine/Upper Engine -> Security/Reactor: 6 Security -> Reactor: 5	Cafet -> Upper E: 9.5 Cafet -> Weapons: 5.5 Cafet -> Admin: 0 Cafet -> Storage: 8 Cafet -> MedBay: 7.5 Upper Engine -> MedBay: 7 Weapons -> O2: 4.5 Weapons -> Nav: 0 Weapons -> Shield: 10 O2 -> Nav: 6 O2 -> Shield: 9.5 Nav -> Shield: 0 Shield -> Comm: 4.5 Shield-> Storage: 6.5 Comm -> Storage: 6 Storage -> Electrical: 7 Storage -> Lower Engine: 10 Lower Engine -> Electrical: 8 Upper Engine -> Lower Engine: 7 Lower/Upper Engine -> Reactor: 0 Lower/Upper Engine -> Security: 6 Security -> Reactor: 5 Security -> MedBay: 0 Security -> Electrical: 0 MedBay -> Electrical: 0 Cafet/Admin -> Nav: 4 Cafet/Admin -> O2: 6 Cafet/Admin -> Shields: 3.5

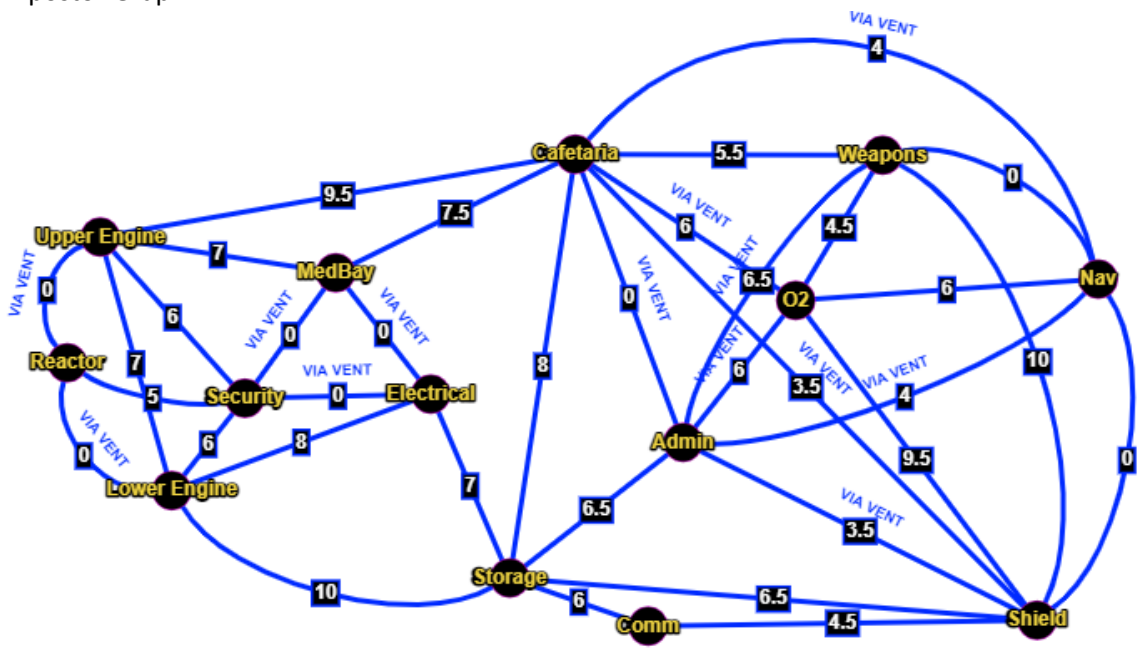
Vermeulen Barthélemy
Villeneuve Matthis

1. **The Two graphs:**

Crewmate Graph:



Impostor Graph:



2. **Algorithm: Floyd-Warshall**

In order to get every time between each combination of room, we need to use an algorithm that solves all pair of the shortest paths, which is exactly what Floyd-Warshall's algorithm does. We implemented this graph with the Floyd-Warshall algorithm we already coded in class, the code will be included, here are the two matrices for both impostors and Crewmates: (the values are rounded to the closest int)

3. Travel time:

```
Impostor's floyd:


Shortest paths beween each nodes/vertices:
      0      9      9      7      9      7      7      0      6      8      3      6      3      3
      9      0      0      5      0      5      5      9     10     16     13     15     13     13
      9      0      0      5      0      5      5      9     10     16     13     15     13     13
      7      5      5      0      5      0      0      7      7     13     11     13     11     11
      9      0      0      5      0      5      5      9     10     16     13     15     13     13
      7      5      5      0      5      0      0      7      7     13     11     13     11     11
      7      5      5      0      5      0      0      7      7     13     11     13     11     11
      0      9      9      7      9      7      7      0      6      8      3      6      3      3
      6     10     10      7     10      7      7      6      0      6      6     11      6      6
      8     16     16     13     16     13     13      8      6      0      4      9      4      4
      3     13     13     11     13     11     11      3      6      4      0      4      0      0
      6     15     15     13     15     13     13      6     11      9      4      0      4      4
      3     13     13     11     13     11     11      3      6      4      0      4      0      0
      3     13     13     11     13     11     11      3      6      4      0      4      0      0


\Crewmate's floyd:


Shortest paths beween each nodes/vertices:
      0      9     16     15     15      7     15      7      8     14     14     10      5     12
      9      0      7      6      6      7     15     17     17     23     23     19     15     21
     16      7      0      6      6     14      8     16     10     16     16     26     22     24
     15      6      6      0      5     13     14     22     16     22     22     25     21     27
     15      6      6      5      0     13     14     22     16     22     22     25     21     27
      7      7     14     13     13      0     22     15     15     21     22     17     13     19
     15     15      8     14     14     22      0     13      7     13     13     23     20     21
      7     17     16     22     22     15     13      0      6     12     13     17     13     19
      8     17     10     16     16     15      7      6      0      6      6     16     13     14
     14     23     16     22     22     21     13     12      6      0      4     14     14     12
     14     23     16     22     22     22     13     13      6      4      0      9     10      7
     10     19     26     25     25     17     23     17     16     14      9      0      4      6
      5     15     22     21     21     13     20     13     13     14     10      4      0      6
     12     21     24     27     27     19     21     19     14     12      7      6      6      0
```

Let’s write both matrices and highlight the paths that are similar for both.

CREW	Cafeteria	Upper E	Lower E	Security	Reactor	Medbay	Electrical	Admin	Storage	Comm	Shield	O2	Weapons	Nav
Caf	0	9	16	15	15	7	15	7	8	14	14	10	5	12
Up E	9	0	7	6	6	7	15	17	17	23	23	19	15	21
Low E	16	7	0	6	6	14	8	16	10	16	16	26	22	24
Sec	15	6	6	0	5	13	14	22	16	22	22	25	21	27
React	15	6	6	5	0	13	14	22	16	22	22	25	21	27
Med	7	7	14	13	13	0	22	15	15	21	22	17	13	19
Elec	15	15	8	14	14	22	0	13	7	13	13	23	20	21
Admin	7	17	16	22	22	15	13	0	6	12	13	17	13	19
Storage	8	17	10	16	16	15	7	6	0	6	6	16	13	14
Comm	14	23	16	22	22	21	13	12	6	0	4	14	14	12
Shield	14	23	16	22	22	22	13	13	6	4	0	9	10	7
O2	10	19	26	25	25	17	23	17	16	14	9	0	4	6
Weapons	5	15	22	21	21	13	20	13	13	14	10	4	0	6
Nav	12	21	24	27	27	19	21	19	14	12	7	6	6	0

Vermeulen Barthélemy
Villeneuve Matthis

IMPOSTOR	Cafet	Upper E	Lower E	Security	Reactor	Medbay	Electrical	Admin	Storage	Comm	Shield	O2	Weapons	Nav
Cafet	0	9	9	7	9	7	7	0	6	8	3	6	3	3
Up E	9	0	0	5	0	5	5	9	10	16	13	15	13	13
Low E	9	0	0	5	0	5	5	9	10	16	13	15	13	13
Sec	7	5	5	0	5	0	0	7	7	13	11	13	11	11
React	9	0	0	5	0	5	5	9	10	16	13	15	13	13
Med	7	5	5	0	5	0	0	7	7	13	11	13	11	11
Elec	7	5	5	0	5	0	0	7	7	13	11	13	11	11
Admin	0	9	9	7	9	7	7	0	6	8	3	6	3	3
Storage	6	10	10	7	10	7	7	6	0	6	6	11	6	6
Comm	8	16	16	13	16	13	13	8	6	0	4	9	4	4
Shield	3	13	13	11	13	11	11	3	6	4	0	4	0	0
O2	6	15	15	13	15	13	13	6	11	9	4	0	4	4
Weapons	3	13	13	11	13	11	11	3	6	4	0	4	0	0
Nav	3	13	13	11	13	11	11	3	6	4	0	4	0	0

The yellow numbers are the distances that are the same for both Impostors and Crewmates. Any other movement will not take the same amount of time depending on the role of the player.

So here are the pairs in which we wouldn't be able to tell if a player is a crewmate or an impostor:

- Cafeteria/Upper Engine
- Cafeteria/Med Bay
- Lower Engine/Storage
- Lower Engine/Comm
- Security/Reactor
- Electrical/Storage
- Electrical/Comm
- Storage/Admin
- Storage/Comm
- Storage/Shield
- Comm/Shield
- O2/Weapons

Apart from those, any other movement will show differences between Crewmates and Impostors.

4. Time interval where it is clear the player is an impostor:

We will consider that for a movement between rooms to be suspicious, the actual travel time must be at least 10% faster than the expected time. For example, from Cafeteria to Lower Engine, we expect 16s to do the movement, but it takes 9s for an impostor, making it clear he is one if it moves between those rooms.
As you can see, apart from Upper Engine to Weapons (and the difference is 13 and 15, so the fact we didn't count this one is due to some simplifications used in the python program, the variation is 13% so could still be counted)
Here is a written summary of all the combinations, along with the difference of time between a crewmate vs Impostor's time to travel between the two rooms. The interval will be displayed like this: [Min Time For Impostor ; CrewMate Time-1] If the duration of a movement is in this range, the player is likely to be an impostor, if the duration of a movement is superior of the given range, the player is probably a crewmate.

- Cafeteria/Lower Engine [9;15]
- Cafeteria/Security [7;14]
- Cafeteria/Reactor [9;14]
- Cafeteria/Electrical [7;14]
- Cafeteria/Admin [0;6]
- Cafeteria/Storage [6;7]
- Cafeteria/Comm [8;13]
- Cafeteria/Shield [3;13]
- Cafeteria/O2 [6;9]
- Cafeteria/Weapons [3;4]
- Cafeteria/Nav [3;11]
- Upper Engine/Lower Engine [0;6]
- Upper Engine/Security [5;5] = 5
- Upper Engine/Reactor [0;5]
- Upper Engine/Med Bay [5;6]
- Upper Engine/Electrical [5;14]
- Upper Engine/Admin [9;16]
- Upper Engine/Storage [10;16]
- Upper Engine/Comm [16;22]
- Upper Engine/Shield [13;22]
- Upper Engine/O2 [15;18]
- Upper Engine/Weapons [13;14]
- Upper Engine/Nav [13;20]
- Lower Engine/Security [5;5] = 5
- Lower Engine/Reactor [0;5]
- Lower Engine/MedBay [5;13]
- Lower Engine/Electrical [5;7]
- Lower Engine/Admin [9;15]
- Lower Engine/Shield [13;14]
- Lower Engine/O2 [15;25]
- Lower Engine/Weapons [13;21]

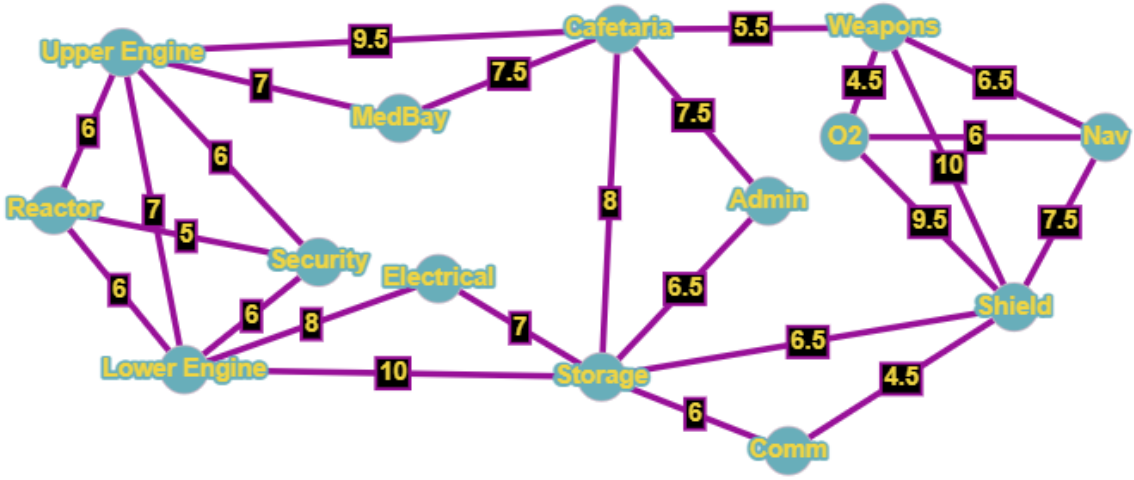
- Lower Engine/Nav [13;23]
- Security/MedBay [0;12]
- Security/ Electrical [0;13]
- Security/Admin [7;21]
- Security/Storage [7;15]
- Security/Comm [13;21]
- Security/Shield [11;21]
- Security/O2 [13;24]
- Security/Weapons [11;20]
- Security/Nav [11;26]
- Reactor/MedBay [5;12]
- Reactor/Electrical [5;13]
- Reactor/Admin [9;21]
- Reactor/Storage [10;15]
- Reactor/Comm [16;21]
- Reactor/Shield [13;21]
- Reactor/O2 [15;24]
- Reactor/Weapons [13;20]
- Reactor/Nav [13;26]
- MedBay/Electrical [0;21]
- MedBay/Admin [7;14]
- MedBay/Storage [7;14]
- MedBay/Comm [13;20]
- MedBay/Shield [11;21]
- MedBay/O2 [13;16]
- MedBay/Weapons [11;12]
- MedBay/Nav [11;18]
- Electrical/ Admin [7;12]
- Electrical/Shield [11;12]
- Electrical/O2 [13;22]
- Electrical/Weapons [11;19]
- Electrical/Storage [11;20]
- Admin/Comm [8;11]
- Admin/Shield [3;12]
- Admin/O2 [6;16]
- Admin/Weapons [3;12]
- Admin/Nav [3;18]
- Storage/O2 [11;15]
- Storage/Weapons [6;12]
- Storage/Nav [6;13]
- Comm/O2 [9;13]
- Comm/Weapons [4;13]
- Comm/Nav [4;11]
- Shield/O2 [4;8]
- Shield/Weapons [0;9]
- Shield/Nav [0;6]
- O2/Nav [4;5]
- Weapons/Nav [0;5]

Step 4: Hamiltonian_Path.py

Hypothesis:

The way this part is explained, in this last phase of the game, so we have to find the shortest possible route that passes through each room. We will use the graph for the Crewmates we have already designed in Step 3; the graph being undirected and having only non-negative weight, this is a mix between the travelling Salesman problem and a Hamiltonian Path: We have to find the shortest Hamiltonian Path, without the need to end the path in the same room we started from.

1. **Map model:**

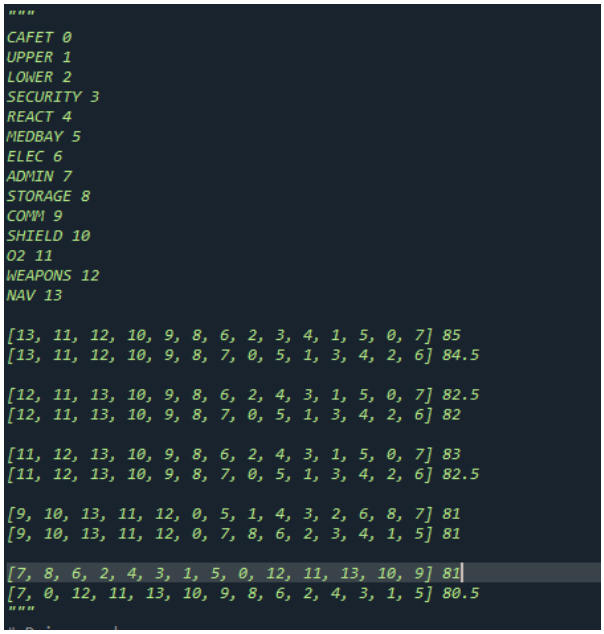


2. **The solution to our problem is a Hamiltonian Path: A route passing through each room only once, and without any constraint as to whether you need to return to the first room.**

3. **We coded an algorithm that displays every possible Hamiltonian Path, we will then calculate each path's total weight and choose the shortest one. From our code, just modify the value of the "Start" variable to match every possible combination (5,6,7,9,11,12,13) and see every Hamiltonian Path.**

4. **With our algorithm, we managed to find an optimal solution: we calculated every Hamiltonian Path given by our algorithm and selected the shortest one. Here is a sample of the given Paths, and their weight: The output is printed that way**
   [room, room, etc] weight

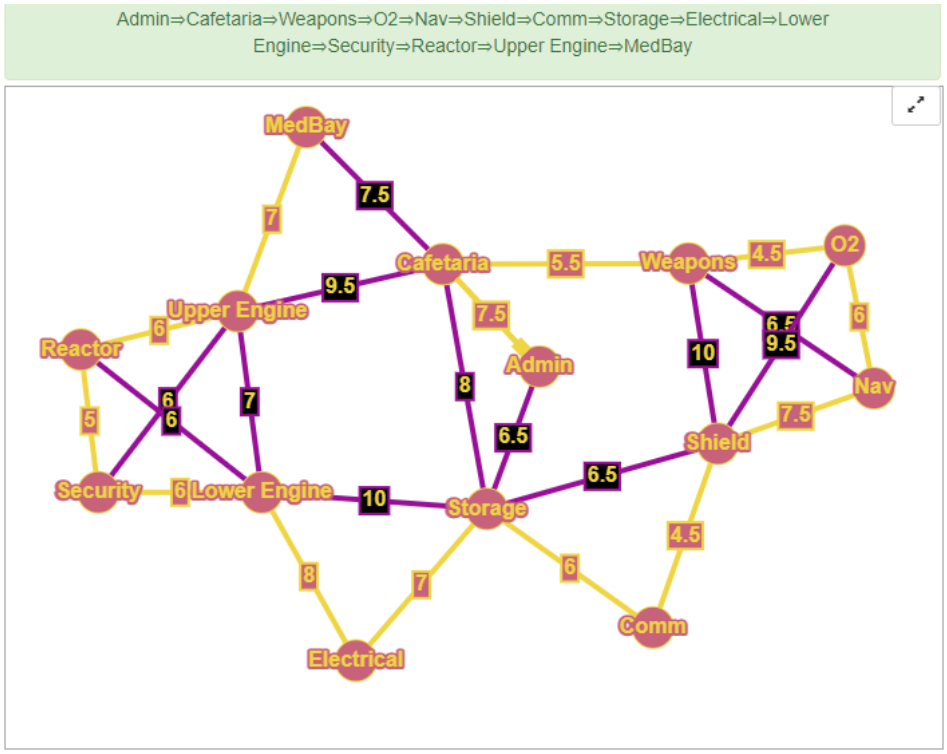
   The equivalent between each room and its number is also written in the screenshot.



```
"""
CAFET 0
UPPER 1
LOWER 2
SECURITY 3
REACT 4
MEDBAY 5
ELEC 6
ADMIN 7
STORAGE 8
COMM 9
SHIELD 10
O2 11
WEAPONS 12
NAV 13

[13, 11, 12, 10, 9, 8, 6, 2, 3, 4, 1, 5, 0, 7] 85
[13, 11, 12, 10, 9, 8, 7, 0, 5, 1, 3, 4, 2, 6] 84.5

[12, 11, 13, 10, 9, 8, 6, 2, 4, 3, 1, 5, 0, 7] 82.5
[12, 11, 13, 10, 9, 8, 7, 0, 5, 1, 3, 4, 2, 6] 82

[11, 12, 13, 10, 9, 8, 6, 2, 4, 3, 1, 5, 0, 7] 83
[11, 12, 13, 10, 9, 8, 7, 0, 5, 1, 3, 4, 2, 6] 82.5

[9, 10, 13, 11, 12, 0, 5, 1, 4, 3, 2, 6, 8, 7] 81
[9, 10, 13, 11, 12, 0, 7, 8, 6, 2, 3, 4, 1, 5] 81

[7, 8, 6, 2, 4, 3, 1, 5, 0, 12, 11, 13, 10, 9] 81
[7, 0, 12, 11, 13, 10, 9, 8, 6, 2, 4, 3, 1, 5] 80.5
"""
```

We can observe that the shortest Hamiltonian Path is the last one of our screenshots, with a total weight of 80.5. Here is the path:
ADMIN – CAFET – WEAPONS – O2 – NAV – SHIELD – COMM – STORAGE – ELECTRICAL – LOWER – SECURITY – REACTOR – UPPER – MEDBAY

In order to make sure our algorithm indeed gave us the shortest one, we checked thanks to an online graph solver, that gave us the following result:



This confirms our choice.

Shortest Path, Best path for the crewmates: