Advanced Data Structures and Algorithms : Among Us mini-problem

Kévin CELIE
Mariyam CHEICK ISMAIL
ESILV A4 - DIA 2

Table of contents

For your information :

This project is made by Kévin CELIE and Mariyam CHEICK ISMAIL.
We decided to do it in **python**.

You can find a version of this project on github by clicking in the following link :
https://github.com/m-cheicki/Among_us_ADSA

You will find the report and all the code commented and documented.
To see the output, run files named

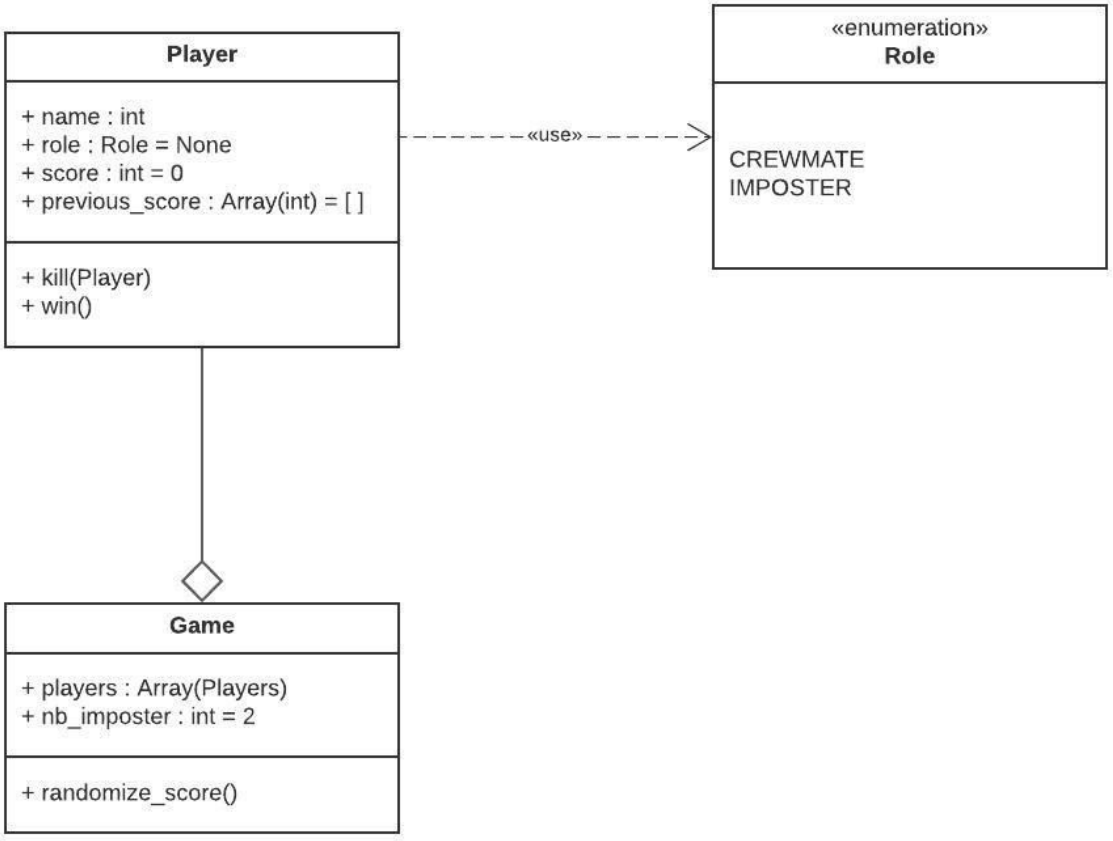
- For step 1 - step1.py
- For step 2 - step2.py
- For step 3 - step3.py
- For step 4 - step4.py

The other files were created in order to refactor the code and present a clean project.

Step 1 : Organize the tournament

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

To represent the Player, we choose to create an object Player, with his name to identify him, his role in the game he is currently playing, his score in the current game, and also an array with all his previous scores in the previous game.
Each game should contain 10 players.
We assume that there are two imposters by default.



2. **Propose a most optimized data structures for the tournament (called database in the following questions)**

For the tournament, we think that the AVL tree will be the most optimized.

AVL tree is a binary search tree (BST) in which the difference of heights of left and right subtrees of any node is less than or equal to one. The technique of balancing the height of binary trees was developed by Adelson, Velskii, and Landi and hence given the short form as AVL tree or Balanced Binary Tree.

In an AVL tree, all operations such as acces, search, insertion or deletion, have a logarithmic time complexity (which gives us O(log(n)) with the Big-O notation), in the best case as well in the worst case. Additionally, it is very helpful when we want sorted datas, as we want to rank players by their score.

It is also the most optimized data structure seen in the Advanced Data Structures and Algorithms course.

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

As we decided to make Player as an object with his score, we just randomize his score by using the random package of python to return an integer between 0 and 12.

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

As our Player is an object, we compute his score value according to his actions, by adding his score before the action to the score gained by doing the action. And, then, we re-create the AVL tree to update the database.

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

In order to create random games, we decide to first create a list of 100 Players. Then, we shuffle the order of that list. We group them into 10 Players to create 10-Player games. And we randomize the score of each player.
We repeat these steps twice more, because we want three random games, in total.

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

As we have made an AVL tree, we can easily have the ranking. Indeed, the AVL tree is quite practical when we want sorted datas, as explained above.

To create games based on ranking, we do an in-order depth traversal of our AVL tree, in order to have increasing order of all the scores. We, then, take ten players by ten to create 10-player games, based on ranking.

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

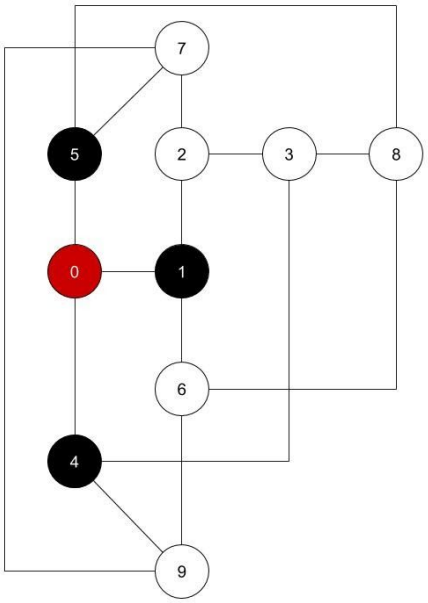
One of the main advantages of the AVL tree is that it is adapted when we want sorted datas.
As, we want to eliminate the ten last players in the ranking, we delete the ten firsts of that in-order traversal. And, we recreate games and rebuild the AVL tree with the remaining players, until there are only ten players left.

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

Once there are only ten players left, we show the in-order traversal of that tree. We can invert the order of that list, as that traversal gives us an array with increasing values. With decreasing values, we can have our TOP 10 players. To show the podium, we select only the three-first players.

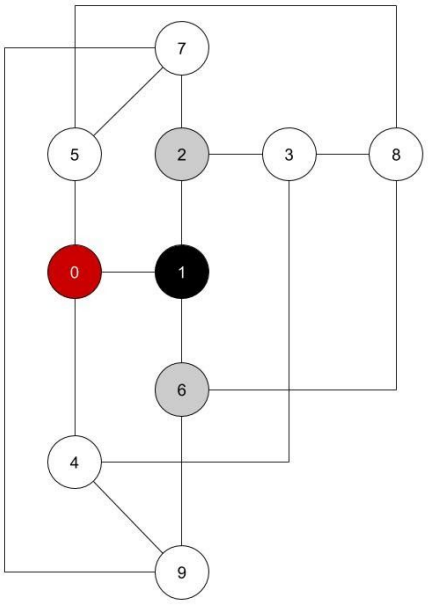
Step 2 : Professeur Layton < Guybrush Threepwood < You

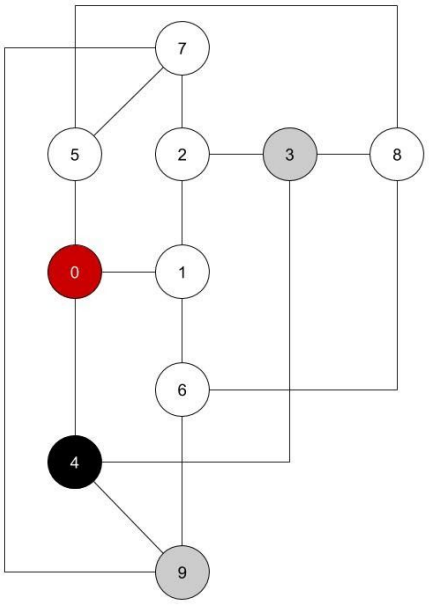
1. Represent the relation between players as a graph, argue about your model

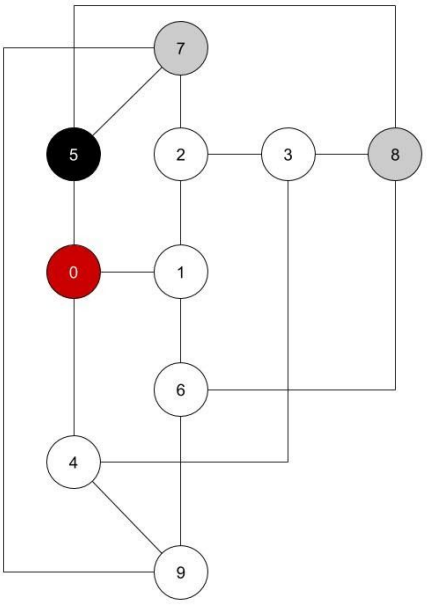


In red, the Player 0 is dead. It was a crewmate.
It has been killed by one of the two imposters in the game.
So, we can conclude that either Player 1, or Player 4 or Player 5 was an imposter.

In this part, we have to find the second imposter. In order to find him, we know that he didn't see Player 1, or Player 4, or Player 5.







If Player 1 is the imposter	If Player 4 is the imposter	If Player 5 is the imposter
P1 has seen P0, P2 and P6. P0 is dead and we know the 2 imposters haven't seen each other. **In white, possible imposters.**	P4 has seen P0, P3 and P9. P0 is dead and we know the 2 imposters haven't seen each other. **In white, possible imposters.**	P5 has seen P0, P7 and P8. P0 is dead and we know the 2 imposters haven't seen each other. **In white, possible imposters.**

We can solve our problem with the graph representation. In our case, our graph will be

unweighted and undirected.

To represent a graph model, we can either do it with adjacency lists or adjacency matrices.

An adjacency list is an array of lists. The length of this array will be equal to the number of vertices, so 10 in our case.
The pros of this representations are :

- In terms of time complexity, it is fast to :
    - Add a new edge : O(1)
    - Add or delete a node
    - Iterate over all edges because you can access any node neighbors directly
- In terms of space complexity, the lesser the number of edges are, the lesser the memory space will be used.

But this representation presents disadvantages like finding the presence or absence of specific edges between any two nodes, which is slightly slower than with the matrix.

An adjacency matrix is a two-dimensional array of VxV size where V is the number of vertices. In our case, it will be a 10x10 matrix. For i and j, two vertices of the graph, if there is an edge between them, the value of matrix[i][j] will be equal to 1. If there is no edge between these two vertices, the value will be 0.
As we build an undirected graph, the matrix will be symmetric.
This representation have advantages like :

- Easy implementation
- Fast deletion of an edge : O(1)
- Fast detection of edges between two vertices : O(1)

But has also its disadvantages :

- Consumes a lot of space however is the number of vertices : $O(V^2)$
- Adding a vertex is time costly : $O(V^2)$

To solve our problem we would rather take the second option. Indeed, even if it consumes a lot of space memory, it is easier to implement. Moreover, we want easy and fast detection of edges between two vertices, which is one of the advantages of this type of representation.

2. **Thanks to a graph theory problem, present how to find a set of probable imposters.**

To solve our problem, we decided to use the principle of the graph coloring approach.
The graph coloring approach is a simple way to label graph components such as regions, vertices and edges under specific constraints.
In theory, two adjacent vertices, adjacent edges or adjacent regions, cannot share the same color. Moreover, we have to find the minimum of different colors that respects this constraint (as known as the chromatic number).

In our situation, we can adapt this graph theory approach. Indeed, in the previous question, you can see that we have colored the relations with the possible first imposter in order to say that they cannot be imposters.
If the first imposter has seen some players, they cannot be imposters. So, the others are possible imposters.

3. **Argue about an algorithm solving this problem**

The initial graph coloring problem is implemented following these steps :

- First, color the first vertex with one color
- For the remaining V - 1 vertices, do :
    - Consider the currently picked vertex V
    - Coloring it with the lowest numbered color that has not been used on any previous colored vertices adjacent to it
    - If all previously used colors appear on vertices adjacent to V, assign a new color

We have to adapt this pseudo code in our situation.
As explained in the first question of this step, to find a set of possible imposters, we first determine who had relations with the first imposter :

- If Player 1 was the imposter : Player 2 and Player 6 cannot be imposters.
- If Player 4 was the imposter : Player 3 and Player 9 cannot be imposters.
- If Player 5 was the imposter : Player 7 and Player 8 cannot be imposters.

The others are the possible imposters. So, we decide to color in the same color the possible first imposter and its direct relations. And we conclude that others are possible imposters.

To make it even simpler, we decide to use booleans :

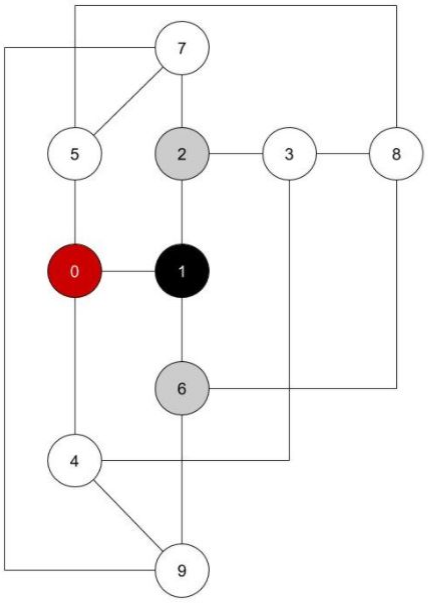
- If the first imposter saw a player there is a relation between them and he cannot be an imposter (0).
- If the first imposter didn't see a player, there is no relation between them, so he can be a imposter (1)

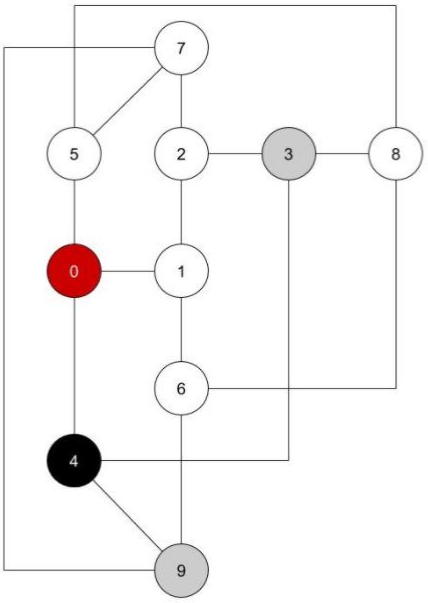
We then show all the players that doesn't have relation with the imposter

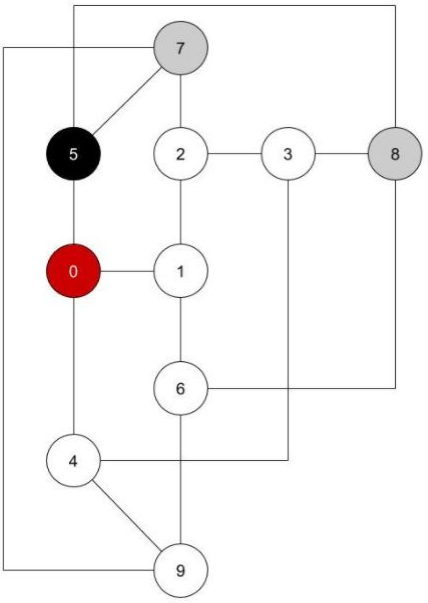
4. Implement the algorithm and show a solution

```
If the first imposter is PLAYER 1, the other imposter is one of these players :
3 4 5 7 8 9
If the first imposter is PLAYER 4, the other imposter is one of these players :
1 2 5 6 7 8
If the first imposter is PLAYER 5, the other imposter is one of these players :
1 2 3 4 6 9
```

We can see that it corresponds exactly to our schema (where we represent in white the set of possible imposters)







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

1. **Present and argue about the two models of the map.**

For this part of the mini-problem, we consider an undirected weighted graph.

An undirected graph is a graph where nodes (or vertices) are connected together and where all edges are bidirectional. This means that going from point A to point B is the same as going from point B to point A.

A weighted graph is a graph where each edge has a weight. A weight is a numerical value. In our case, the weight of each edge of our graph will be the distance between two nodes (and so the time it takes to go from the node to another one, as we consider that 1 cm is equal to 1 second).

We decide to represent our two models as two undirected weighted graphs by computing adjacency matrices.
Note that the two matrices will be very similar to each other. The difference is that the imposter can take vents so he can travel from a room to another one quicker than a crewmate. Only a few values will differ but they work exactly the same.

As there are 14 rooms and we have an undirected graph, so our both matrices will be a 14x14 symmetric matrix.

2. **Argue about a pathfinding algorithm to implement.**

A pathfinding algorithm is an algorithm that permits us to find the shortest path between two points. There are multiple pathfinding algorithms. Here are these seen during our course :

- **Bellman-Ford Algorithm** : solves the single-source problem if edge weights may be negative
- **Dijkstra's Algorithm** : solves the single-source shortest path problem with non-negative edge weight
- **Floyd-Warshall** : solves all pairs shortest paths

Time and space complexities of these algorithms :

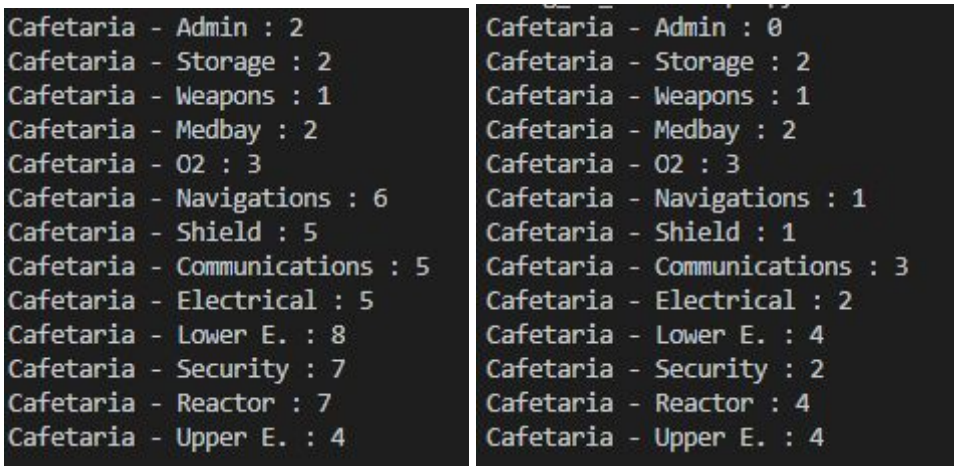
	Dijkstra	Bellman-Ford	Floyd-Warshall
Space complexity	O(M)	O(M)	$O(N^2)$
Time complexity	$O(N^2)$	O(MN)	$O(N^3)$
When edge weights are negative	No	Yes	Yes

Where N is the number of nodes and M the number of edges.

For our problem, we want to find all shortest paths between every node. The only algorithm we studied that can do this trick is the Floyd-Warshall algorithm (but its time and space complexity aren't really great though).

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

Here is an extract of the output that we have.



On the left, there is the time to travel from the cafeteria door to each room, for a crewmate.
On the right, there is the time to travel from the cafeteria door to each room, for an imposter.

4. **Display the interval time for each pair of rooms where the traveler is an imposter.**

In order to do so, you just have to run the Floyd-Warshall algorithm in the imposter's map. You will see the full output. Here is another extract of the output.

```
Cafetaria - Admin : 0
Cafetaria - Storage : 2
Cafetaria - Weapons : 1
Cafetaria - Medbay : 2
Cafetaria - O2 : 3
Cafetaria - Navigations : 1
Cafetaria - Shield : 1
Cafetaria - Communications : 3
Cafetaria - Electrical : 2
Cafetaria - Lower E. : 4
Cafetaria - Security : 2
Cafetaria - Reactor : 4
Cafetaria - Upper E. : 4

Admin - Storage : 2
Admin - Weapons : 1
Admin - Medbay : 2
Admin - O2 : 3
Admin - Navigations : 1
Admin - Shield : 1
Admin - Communications : 3
Admin - Electrical : 2
Admin - Lower E. : 4
Admin - Security : 2
Admin - Reactor : 4
Admin - Upper E. : 4

Storage - Weapons : 3
Storage - Medbay : 3
Storage - O2 : 5
Storage - Navigations : 3
Storage - Shield : 3
Storage - Communications : 3
Storage - Electrical : 3
Storage - Lower E. : 5
Storage - Security : 3
Storage - Reactor : 5
Storage - Upper E. : 5

Weapons - Medbay : 3
Weapons - O2 : 2
Weapons - Navigations : 0
Weapons - Shield : 0
Weapons - Communications : 2
Weapons - Electrical : 3
Weapons - Lower E. : 5
Weapons - Security : 3
```

Step 4 : Secure the last tasks

1. **Present and argue about the model of the map.**

The model of the map is an undirected and unweighted graph that we represent by an adjacency matrix.

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

To find a route in a graph there is several algorithms like :

- the Kruskal algorithm
- the Prim's algorithm
- the Hamiltonian path or Hamiltonian cycle

The two firsts algorithms quoted above are for MST (which stands for Minimum Spanning Tree).

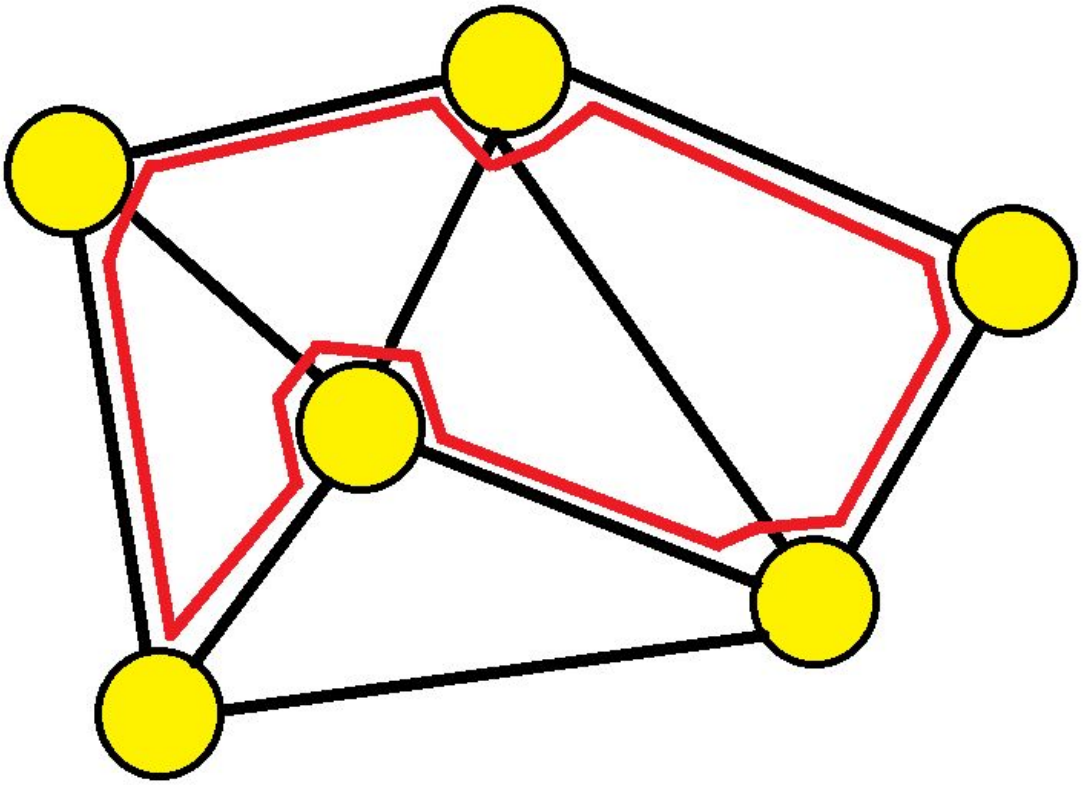
Given a connected and undirected graph, a **spanning tree** of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A **minimum spanning tree** (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.
Shortly, a Minimum Spanning Tree (MST) problem is : Given connected graph G with positive edge weights, **find a minimum weight set of edges that connects all of the vertices**.

Kruskal algorithm :

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If a cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

Prim's algorithm :
It finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

The **Hamiltonian path** is a path that visits exactly once each vertex of our graph.



3. **Argue about an algorithm solving your problem.**

To solve our problem, we choose to implement the Hamiltonian path. Indeed, we want a path where each room is visited exactly once.

4. **Implement the algorithm and show a solution.**

If we start from the cafeteria, the crewmate won't be able to visit exactly once each room of the map. There is no solution.
But if he decides to start from the admin room for instance, he can visit every room once.

```
Solution does not exist

Solution Exists: Following is one possible Hamiltonian path
Admin  Cafetaria  Weapons  O2  Navigations  Shield  Communications  Storage  Electrical  Lower E.  Security  Reactor  Upper E.  Medbay
```