Lucie Drevet
Inès Nguyen

DIA 1

Advanced Data Structure and Algorithms
Mini-Problem Report

ESILV LÉONARD DE VINCI
ÉCOLE D'INGÉNIEURS PARIS-LA DÉFENSE

Summary

A- To organize the tournament

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

In this project, we were asked to organize the next "Among Us" tournament for the next ZLAN. We had a total of 100 players and 10 players per game.

To represent a player and its score created a class called Node. Each node contains the player's number, his score during the current party, his total score (the sum of all his scores divided by the number of games played), his left node, his right node and his height.

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

We were asked to use a tree with a log complexity to reach an element. For this reason, we decided to choose an AVL Tree.

An AVL is a self-balancing Binary Search Tree. In this tree the difference between heights of left and right subtrees is equal to -1, O or +1.

In our function "*Database_game*", we filled in our AVL Tree with the players number, his current score and his total score. At the beginning, we created a value *score* that will contain the random score of the player and appended the list *saving* that we will use later to keep track of all the scores the player had in the tournament.

Then, if we were not on the finals, we filled the node of our AVL Tree with the player number, his random score and his total score (mean of all his scores). At every game played, we recreated an AVL Tree thanks to this function, but the previous scores were saved in "*saving*".

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

The players score can be between 0 and 12, so we simply decided to write a function *"Score*" that returns a random value between 0 and 12. Thanks to that, we only need to call this function whenever we want to attribute a current score during the party.

Because we were asked to calculate the total score based on the mean of all his game, we created a list *saving* that saved all the previous scores associated with the player and it was implemented in "*Database_game*".

We created a function called "*Mean*" that returns the total score of a player (the mean of all his scores). In order to do so, we created a value *scoreTotal* and we went through our list *saving* and looked for all the scores associated with a player. Everytime the scores were associated with the player number, we increased *scoreTotal* by the current score.

So, at the end, *scoreTotal* was equal to all the previous scores of a player.

Finally, we calculated the mean score of a player by dividing *scoreTotal* by the total number of games played.

It is important to note that in the following chapters, the total score represent the mean score.

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

We decided to create multiple AVL Trees. First, we created an AVL tree with 100 nodes for the 100 players and every variable contained in the node were equal to 0.

Then, we used a function called "*Database_game*" that filled in the nodes with the player number, its current score and its total score. In the class node, we created a function "*insert_player"* that allows us to insert those value in the node.

Plus, this function balances our tree. To do so, it updates the height, checks if the total score (called mean in our code) of the current node is superior than the total score in the left node and if the total score of the current node is inferior than the total score in the right node.

If those conditions weren't respected, we did a right rotation when the balance factor is superior than 1 and a left rotation if it is inferior than -1.

Every time we want to update the scores, we call the function "*Database_game"* that will then use the function insert_player and therefore balance the tree.

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

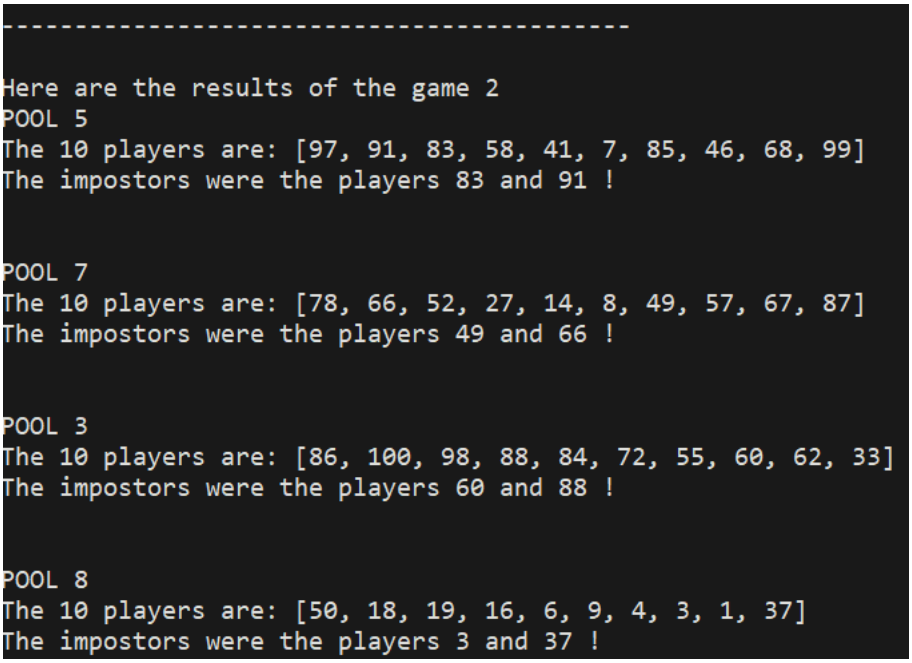
To create random games based on the database, we wrote different functions that would attribute a role and pools for each player:

First, we began by separating all the players into 10 unique batch.

Then, we wanted to attribute to each players a completely random pool. For the first 3 games, the pools were not attributed based on a ranking. So, we thought that creating a function "*RandomGame*" would help us. *"RandomGame"* simply returns a pool that has a random value between 1 and 10. To assure ourselves that all the pools were unique, we created a list composed of all the previous pool values. If the current pool was contained in this list, we reimplemented the function thanks to the recursivity.

To continue, we needed to attribute roles to the players. To do that we wrote the function "Impostors". It creates two variables: "impostor1" and "impostor2". Then, with the exact same principles of the function "RandomGame" explained previously, it checked if those two values were different.

Finally, we displayed the pool, the players associated to this and who were the impostors with "Games". All the players were displayed sorted by their total score using the *"inOrder"* that we will explain later.



An example of a random game 2

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

We were asked to create 10 games based on ranking. At every game, the 10 last players were eliminated.

We began by sorting all the players in function of their total scores. For that we used an In-Order Traversal with our function *"inOrder".* The In-order Traversal visit the left node first, then the root and after that the right node. This means that it will return the values in an ascending order. In our function "*inOrder*", we returned tree lists (players, scores, total scores) sorted by the total score of each player.

Then we reused "*Games*". In "*Games*", we separate the players into n group of 10. Afterwards, we displayed the number of the pool for the 10 following players.

If we were in the 3 first games the pools were attributed randomly like we saw previously.

Otherwise, we used tree lists that took the players, scores and total score of the current pool sorted by the total score thanks to our "*inOrder"*. Next, we displayed the pools, the players associated to this pool and who were the impostors. To know in which pool the players were in, we simply created the variable *nbPool* which was equal to the number of players divided by ten plus one to have a decreasing number between 1 and the number of players.

The best players were in the pool 1 and the worst players were in the pool with the highest number.

```
POOL 1.0
The 10 players are: [74, 51, 48, 37, 29, 20, 10, 11, 5, 43]
The impostors were the players 37 and 5 !
```

An example of a 1st pool

```
----------------------------------------
Here are the results of the game 11
with a number of 30 players:

POOL 3.0
The 10 players are: [6, 1, 2, 28, 25, 47, 40, 35, 22, 31]
The impostors were the players 40 and 2 !
```

An example of the lowest pool in the game 11

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

In our main function, we created the variable *n* that represented the number of players at the beginning of the tournament.

Next, we made a list *excluded* that will contain all the players eliminated from the tournament and a list *players_number* that will be fill in by the players number for the current game.

Then, until the number of players was 10, we checked if the player was excluded and if it was not the case, we appended our *players_number* list. We implemented an AVL tree with the values of *player_number* and our game was created.

Afterwards, with "*inOrder*" we sorted the players, their scores and their total scores based on the total score and we appended *excluded* with the last 10 players.

We decided to display the excluded players for the following parties with their total score.

```
The 10 players excluded from the following game are:
Player excluded 93  :  Total Score : 3
Player excluded 91  :  Total Score : 3
Player excluded 90  :  Total Score : 3
Player excluded 81  :  Total Score : 4
Player excluded 62  :  Total Score : 4
Player excluded 52  :  Total Score : 4
Player excluded 51  :  Total Score : 4
Player excluded 33  :  Total Score : 4
Player excluded 14  :  Total Score : 4
Player excluded 11  :  Total Score : 4
```

An example of all the players excluded for the game 12

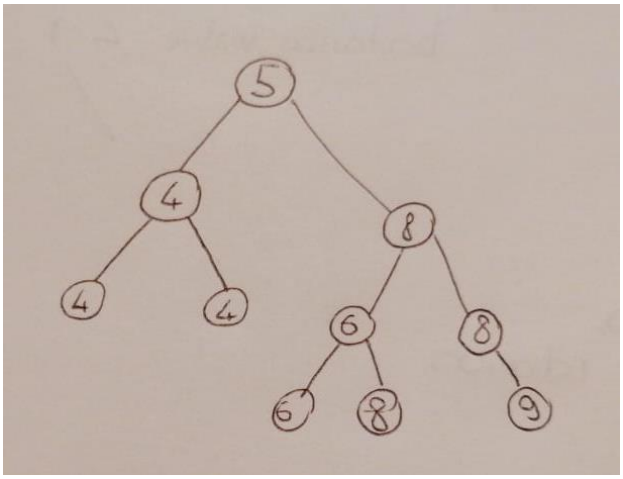
When they were only ten players lefts, we were in the finals.
For the finals, we decided to show the players, their total score and who were the impostors for the 5 finals games.

```
----------------------------------------
Here are the results of the game 5

Player  80  :  Total Score : 4
Player  18  :  Total Score : 4
Player  27  :  Total Score : 4
Player  71  :  Total Score : 5
Player  77  :  Total Score : 6
Player  50  :  Total Score : 6
Player  36  :  Total Score : 8
Player  8   :  Total Score : 8
Player  52  :  Total Score : 8
Player  20  :  Total Score : 9
The impostors were the players 18 and 80 !


Here are the 10 finalists :
Player  80
Player  18
```

An example of the game 5 in the finals

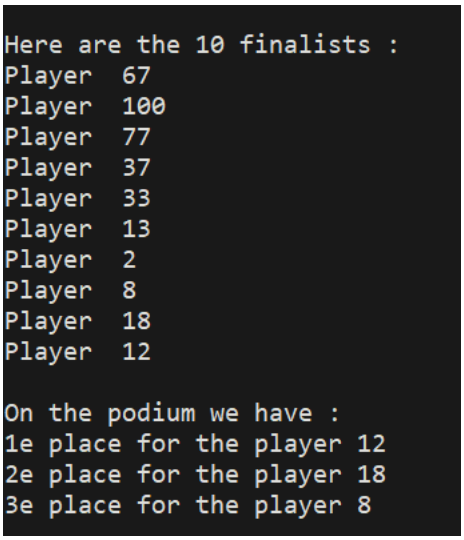


The AVL Tree corresponding to this game

8) Present and argue about a method which display the TOP10 players and the podium after the final game

For the finals, we reinitialized the scores to have total scores only based on the last five games. We made a function "*TOP10_and_podium*" that create tree lists: *finalists, finalScore, finalMean* sorted by the total score. Those lists contained all the values of the finals.

Next, we displayed all the finalists and their place by going through the *finalists* list and taking the 3 first players in decreasing order. The third player was the third last on our list, the second player, the second last on our list and finally the first player was the last on our list.



A screenshot of how we displayed the 10 finalists

B- Professor Layton < Guybrush Threepwood < You

1) Represent the relation (have seen) between players as a graph, argue about your model

To represent the relation 'have seen' between players, decided to use a dictionary. The number of keys is equal to the number of players who are still alive, and each represent a player.

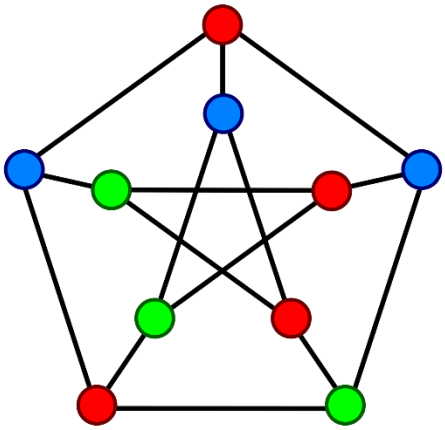
And then for each key, each value represents the have seen relationship of a player. Here each key is a node, and the values represent the vertices of the graph

2) Thanks to a graph theory problem, present how to find a set of probable impostors

To find a set of probable impostors, we decided to use the *graph coloring theorem*. In graph theory, graph coloring consists in assigning a color to each of its vertices so that two vertices connected by an edge have different colors.

This graph was optimal to our problem because we know that impostors are likely to walk separately. This means that if a player has the same color as the node 1, 4 and 5 it is highly probable that he is also an impostor.

Often, we tried to use the minimum number of colors, called chromatic number. Then, thanks to the colors affected to the nodes, we were be able to find a set of probable impostors by looking to the node that had the same color as node 1, 4 and 5.



An example of a graph coloring diagram from Wikipedia

3) Argue about an algorithm solving your problem

For this type of graph theory problem, we used a *greedy algorithm*.

A greedy algorithm is a simple and intuitive algorithm that is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem.

The algorithm will create a dictionary that will define the color of each node. The keys represent the nodes and the values are the colors.

For that, it attributed a color to the first node and then for each node it looked if the previous node was a neighbor, and if yes it changed the color. Once the colors were attributed, we searched which nodes had the same color as the node 1, 4 and 5. If it has the same color, we made a list of probable impostors.

4) Implement the algorithm and show a solution

First, we run through the nodes with the dictionary *neighbors* and created a list *"l"* that contains all the colors of the nodes by exploring all the nodes in *colors_of_nodes.*

Then, we created a dictionary *imp* where the values of each keys were the possible partners of an impostor. Every time two colors were similar in the list *l*, we appended the values of *imp* for the given key.

So, we had all players that did not walked next to the player 1, 4 and 5.

```
Player  1  can be associated with player  3
Player  1  can be associated with player  5
Player  1  can be associated with player  9
Player  4  can be associated with player  2
Player  4  can be associated with player  6
Player  5  can be associated with player  1
Player  5  can be associated with player  3
Player  5  can be associated with player  9
```

A screenshot representing the probable impostors

C- I don't see him, but I can give proofs he vents!

1) Presents and argue about the two models of the map

In this step, our map will be represented by a non-directed graph with weights on vertices node represents a room, a vertex represents the link between two rooms and the weight of the vertices represent the length of this link.

We represented the two models of the map by creating two list of lists where the first two columns are two linked nodes and the third one is the weight of the vertices between those nodes. The first one, *model1*, was representing the map (distance between each room) without the vents and the second one, *model2,* the map with the vents. When they were a vent between two rooms, we put the distance between two rooms equals to 0.

To know the distances of the map, we draw it and took the measurement between all the rooms.

There is obvioulsy two different model of graph the same map.

The first graph contains 14 nodes and 25 vertices. One node is a room, and a vertex between two nodes represent the distance between the two rooms. The weight of each vertices is defined by the basic distance between the rooms that are associated. Those distances have been measured directly on the map.

The second graph is pretty close to the first one. The main difference is that for each nodes/room connected by a vent, we replace the weight of the vertices by 0, if the vertices already existed, and if not, we add a vertex (with a weight of 0 of course).

We represented these two graphs by matrix, where we doubled the first two columns of a line represent two node who are connected, and the third columns is the weight of their vertices. We doubled the number of lines to represent a non-directed graph.

2) Argue about a pathfinding algorithm to implement

For this problem we chose an algorithm to solve a shortest path problem. We decided to use the Floyd-Warshall's algorithm.

We decided to use this algorithm because contrary to his colleagues Djikstra's algorithm and Bellman-Ford's algorithm who solve single source shortest path problem, Floyd-Warshall's

algorithm solves all pairs shortest paths. Plus, this algorithm is working whether the edges weights are positive or negative.

When we execute the Floyd-Warshall's algorithm, it will return the length of shortest paths between every pairs of vertices.

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

We had fourteen rooms in our map, so we put fourteen vertices in the graph with the variable *V* equals to fourteen. Then we define the infinity (*INF* in our code) as the large enough value. INF will later be used for vertices that are not connected to each other.

In addition to that, with the function "*floydWarshall*", we solved all pair shortest path.

In order to do so, we created a list of lists (or matrix) *dist* that will be the output matrix. At the end, this matrix will contain the **shortest distances between every pair of vertices** initializing the solution matrix same as input graph matrix. We can say that the initial values of shortest distances are based on shortest paths considering no intermediate vertices

After that, we added all vertices one by one to the set of intermediate vertices.

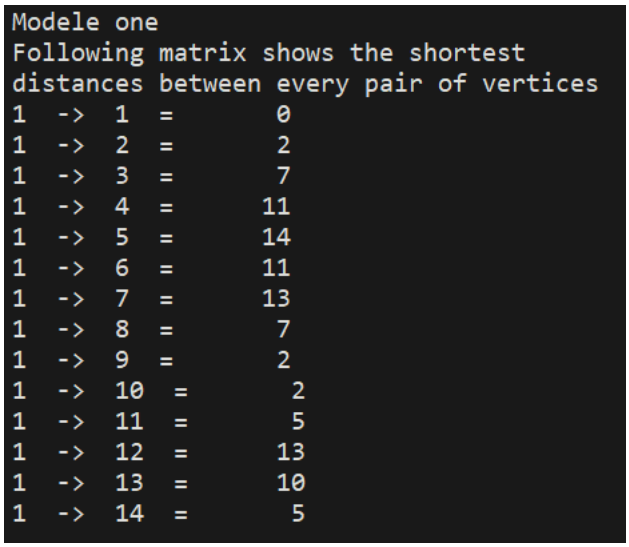
Before the start of an iteration, we have shortest distances between all pairs of vertices such that the shortest distances consider only the vertices in the set {0, 1, 2, ... k-1} as intermediate vertices.

After the end of an iteration, vertex no. k is added to the set of intermediate vertices and the set becomes {0, 1, 2, ... k}.

We picked all vertices as destination for the above picked source. As we were running through the matrix *dist,* we checked if the current vertex was on the shorted path from a room to another. If it was the case, then update the value of the output matrix *dist* for those rooms.

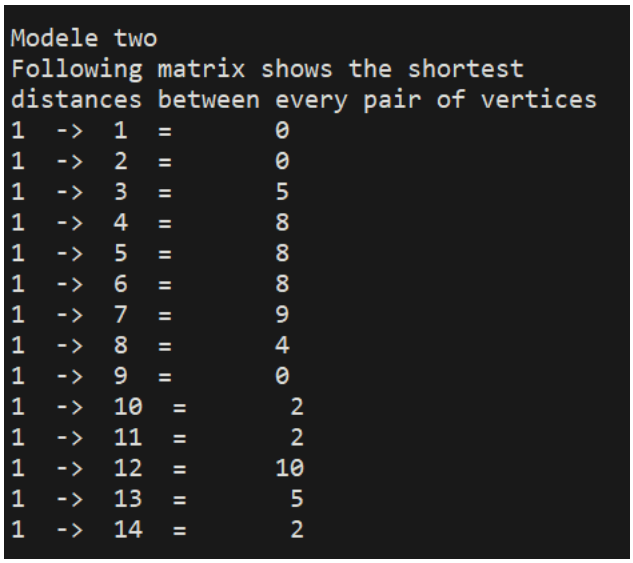
Next, we had to show the time travel for any pair of rooms for both models.

We simply wrote "printSolution", a function that shows the shortest distances between two rooms. Then, in the function "*find_impostor*", we first run through the model one and two and printed our solution as bellow.



An example of how we showed the time travel for any pair of rooms for the model two

Here we can see that the only distance equals to 0 is between the room 1 and 1, which is quite logical. Otherwise, we see that all the shortest distances are different than 0. For example, in line 2, the distance between the room 1 and 2 is 2.

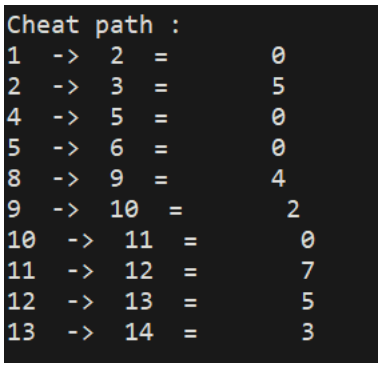


An example of how we displayed the time travel for any pair of rooms for the model two

In this screenshot, we can see that the shorted path between the room 1 and the room 2 is 0 which means that there is a vent between those two rooms.

4) Display the interval of time for each pair of room where the traveler is an impostor

To display the interval of time for each pair of room where the traveler is an impostor, still in the function *"find_impostor"*, we run through all the vertices. Then, when a distance between two rooms of the model two was different in comparison from the model one, we printed the cheated path.



An example of a cheat path

D- Secure the last tasks

1) Presents and argue about the model of the map

For this last step, our map will be represented by a non-directed graph with weights on vertices. Like in the step 3, a node represents a room, a vertex represents the link between two rooms and the weight of the vertex represent the length of this link.

Furthermore, similarly to the step 3, we decided to represent the graph by a matrix where the first two columns are two linked nodes and the third one is the weight of the vertex between those nodes.

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

For this problem, we will use the Hamiltonian graph theory problem and more precisely, the Hamiltonian path problem.

In graph theory, the Hamiltonian path problem is a problem of determining whether a Hamiltonian path (a path in an undirected or directed graph that visits each vertex exactly one) exists in a given graph.

In our situation, this path problem is ideal because we want to visit every room to complete the left missions without passing through the same room twice.

3) Argue about an algorithm solving your problem

The Hamiltonian path's algorithm is the first step to solve our problem. In fact, as we explained earlier, it will find paths where we go through every room without passing by a room twice.

Once we find all the Hamiltonian path that exists in our graph, we must decide which one is the shortest.

For that we will simply calculate the length of each path thanks to the weight of each vertices and we will then return the shortest one.

5) Implement the algorithm and show a solution

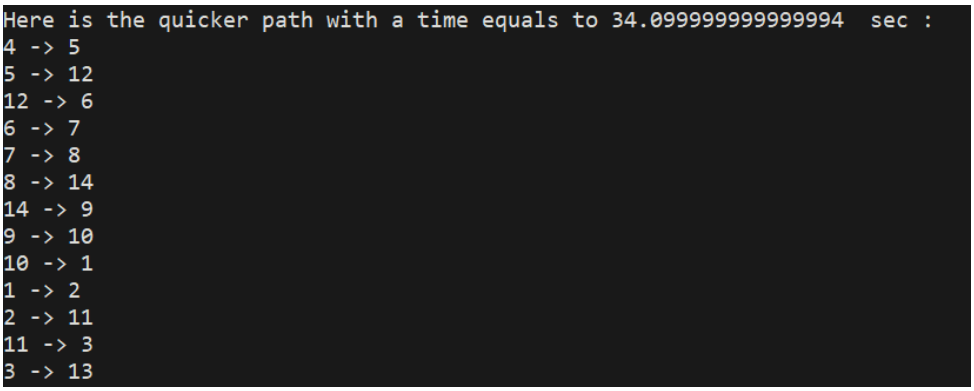
We began by creating a function "*hamilton*". In this function, we have a value *size* that will be equals to the length of our graph. This will help us to know if we visited all the rooms.

We created a dictionary G were the keys are the rooms and the values are all the rooms connected to this room. To have all the possible pahts, we implemented the "*hamilton*" function by beginning with all the different rooms.

So, in "*hamilton*", we began by a room. The current room we were in was represented by the value *v* Then, we created a list *to_visit* that we appended with the value none following by a connected room until there are no more connected rooms left.
Next, we chose a room connected to our room and we did this until v is equal to none, meaning that the room we were in was only connected to rooms we visited before.
If the size of our path was different than the size of our graph, it meant that the path we decided to take could not pass through each room only one time and it was not the path we were looking for.

Then, we popped the rooms (went back) until we found a room with connected rooms we did not visited.
We did those operation until *v* was equal to none and the size of our path was equal to the size of our graph. It meant that all the rooms were visited and only once. We had our final path.

After we had all our possible path by beginning by different rooms, we calculated the distance of each Hamiltonian's paths. We looked for the path with the minimal distance and printed it.



A screenshot of best the Hamiltonian's path

We can see that we passed through every room once in a total of 34.1 seconds.