Intelligent systems lab project

Participants:

Alejandro Del Hoyo Abad
Adrián Gómez Del Moral Rodrígez-Madridejos
Sergio Pozuelo Martín-Consuegra

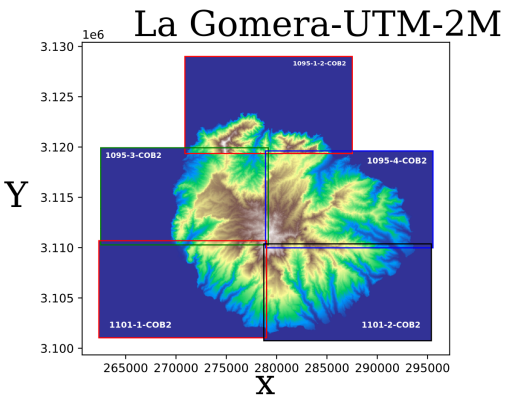
Course: 2023-2024
Group: A1-09

Index

Description of the problem

The goal of this project is to develop an intelligent route planning system for the island of La Gomera, belonging to the Canary Islands. The geographical information for this project can be obtained from the Modelo Digital del Terreno del PNOA.

First, we're asked to implement three basic functions, one to create a map object with all the required attributes from a given *.hdf5* file containing five .asc (ASCII Grid) files containing the information, a second one to return the height value of a set of given coordinates, and a final more complex one to create a scaled down resized copy of a given map to a certain factor and with a selected formula to resolve the conflict of height different when combining map cells (maximum value, mean value or minimum), creating then another .hdf5 file to save the map in.



Once the very basic operations are functional, the next step is to create the search problem artifact with the states, goal and successor functions that will later be part of the path-finding algorithm. For this, a state will be defined as a set of UMT YX coordinates belonging to a map, being 2 of these the beginning and goal states. The successor function will return a list of valid successors, showing the cardinal direction (N: North, E: East, S: South and W: West), the coordinates the successor state is in, the length traveled and the slope crossed.
On the other hand, the goal function's only purpose is to check if a given state satisfies the solution.

With these we can begin to create the search algorithm, we first need a data structure that allows the storing of ordered nodes to implement the frontier, as well as a tree structure created of nodes with an unique ID, parent, state, value (calculated depending on strategy), depth, cost, heuristic and action. The nodes will also have a path function to return the path from the root to that node. The expansion of this tree will be performed via a list of successors on the state, maintaining the order established and having the node value assigned depending on the strategy (BFS, DFS or UCS). There will also be a visited list that can check if a state is in it or add a new one, to complete the basic functions needed for our search algorithm to solve our previously defined problem.

As a final feature, the program will incorporate the A* and greedy strategies with two new heuristics (Manhattan and Euclidean) that each are calculated with a specific formula defined as:
Hmanhattan((y,x))=|Dx-x|+|Dy-y|
Heuclidean((y,x))=sqrt((Dx-x)^2+(Dy-y)^2)

All of this should result in a program capable of reading *.hfd5* files containing geographical data of the island, resizing it and finding a path from one given point to another following one of the 7 possible strategies: BFS, DFS, UCS, Greedy with Euclidean heuristic, Greedy with Manhattan heuristic, A* with Euclidean heuristic and A* with Manhattan heuristic taking into account that it does not have to surpass a maximum slope.

Solution

Task 1: Domain of the problem

For the first task, we implemented the map creation, resizing and coordinate height reading through the use of the following implementation divided in two classes (*Map* and *Dataset*), located in the map_island.py file.

The first thing to implement is the map creation, done through the **__init__()** function in the map class, that when given a file name, initializes the variables that define the map, and initializes the dataset objects that compose the *.hdf5* file. Storing their header information for later use. At the end, a Map object is returned.

```
for dataset_name in self.f:  # Search the datasets and set their values
    dataset = self.f[dataset_name]
    self.nodata_value = dataset.attrs["nodata_value"]
    self.size_cell = dataset.attrs["cellsize"]
    min_xinf = min(min_xinf, dataset.attrs["xinf"])
    min_yinf = min(min_yinf, dataset.attrs["yinf"])
    max_xsup = max(max_xsup, dataset.attrs["xsup"])
    max_ysup = max(max_ysup, dataset.attrs["ysup"])

    dataset_attrs = Dataset(  # Set the attributes of the dataset
        dataset.attrs["xinf"],
        dataset.attrs["yinf"],
        dataset.attrs["xsup"],
        dataset.attrs["ysup"],
        dataset.attrs["cellsize"],
        dataset.attrs["nodata_value"])

    dataset_attrs.name = dataset_name  # Add a name attribute
    self.datasets.append(dataset_attrs)

# Calculate the dimensions of the new map
rows = math.ceil((max_ysup - min_yinf) / self.size_cell) # rows = (maxY - minY) / cellsize
columns = math.ceil((max_xsup - min_xinf) / self.size_cell)  # columns = (maxX - minX) / cellsize
self.dim = (rows, columns)
self.up_left = (min_yinf, min_xinf)
self.down_right = (max_ysup, max_xsup)
```

This is achieved by first searching through every dataset, finding their min and max X and Y values that are later used to set the dimensions of the map, as well as their cell size and *nodata_values*. They are finally added to the list of datasets of the map and the dimensions are calculated in rows and columns as well as the top left and down right corners.

With the map created, we next implemented the *umt_YX* function to find the height value of a specific map in a given set of coordinates. This is achieved by first iterating through the datasets of the map object to locate the dataset where the coordinates are located. Once identified, the algorithm

```
def umt_YX(self, y, x) -> float:  # Given a pair of YX coordinates, return it's value
    """ Find the height of the given y x coordinates"""
    for current_dataset in self.datasets:  # Search the datasets
        # Check the X and Y are in the map boundaries
        if (current_dataset.xinf <= x <= current_dataset.xsup
                and current_dataset.yinf <= y <= current_dataset.ysup):

            # Convert Y and X coordinates to row and column
            row = (current_dataset.ysup - y + 1) // self.size_cell
            col = (x - current_dataset.xinf + 1) // self.size_cell
            data_grid = self.f[current_dataset.name]  # Initialize
            value = data_grid[row, col]
            return value

            # Return the nodata value if the coordinates are outside the map
    return self.nodata_value
```

computes the precise row and column within that dataset corresponding to the coordinates. After this, it retrieves the height value situated at that specific position within the dataset grid. Then, if the coordinates are not located in any dataset, the *nodata_value* of the map is returned.

Finally, the resize method is responsible for creating a scaled copy of a given map to a certain factor and formula to resolve the height. It begins with calculating the new size of the grid cells based on the factor given. After that, it goes through each original dataset to generate new row and column values. It duplicates almost all the details from the originals, excluding the cell size. Eventually, it produces a new grid by applying the specified formula such as calculating the average, the highest or the lowest values for each subgrid. Moreover, it's essential to mention that any *nodata_value* are not taking into account in this process.

```
def resize(self, factor, transform, name):  # Function to make a resizen version of a map
    """ Resize the original map given a factor and a transform function"""
    new_cellsize = self.size_cell * factor  # Multiply the size of the cells for the factor
    new_f = h5py.File(f"{name}.hdf5", "w")  # Create the new file
    attributes_to_copy = ['xinf', 'xsup', 'yinf', 'ysup', 'nodata_value']  # List of attribute names to copy

    for dataset_name in self.f:  # Search the datasets
        current_dataset_grid = self.f[dataset_name]  # Get the old grid
        new_grid_rows = math.ceil(current_dataset_grid.shape[0] / factor) # Create the new row size
        new_grid_columns = math.ceil(current_dataset_grid.shape[1] / factor)  # Create the new column size
        # Create the new grid
        new_dataset_grid = new_f.create_dataset(dataset_name, (new_grid_rows, new_grid_columns), dtype=current_dataset_grid.dtype)

        for attr_name in attributes_to_copy:  # Copy the old attributes of the dataset
            if attr_name in current_dataset_grid.attrs:
                new_dataset_grid.attrs[attr_name] = current_dataset_grid.attrs[attr_name]
        new_dataset_grid.attrs["cellsize"] = new_cellsize  # Replace the cellsize attribute

        # Assign the new values to the grif using the given function
        for i in range(0, new_grid_rows):
            for j in range(0, new_grid_columns):
                # ValueY = row * factor:(row + 1) * factor || ValueX = column * factor:(column + 1) * factor
                cells = current_dataset_grid[i * factor:(i + 1) * factor, j * factor:(j + 1) * factor]
                cells_without_nodata_value = cells[cells != self.nodata_value]
                # Transform only the cells wiht valid values
                new_dataset_grid[i, j] = transform(cells_without_nodata_value) if cells_without_nodata_value.size > 0 else self.nodata_value

    new_map = Map(f"{name}.hdf5")  # Save the new map to a hdf5 file
    return new_map  # Return the map
```

Task 2: Problem (Definition)

The second task involved developing a function within the *search.py* file. This function constructs a problem formed by two states (starting and goal states) each defined by a pair of coordinates. Additionally, it includes a successor function responsible for identifying valid successors in the four cardinal directions for movement. These functionalities were done through the implementation of three classes: Problem, State, and Descendant.

First, the problem is created through the __init__() method of the Problem class that stores the map it belongs to and the coordinates for the initial and goal test. This class also includes the goal test, which verifies whether a given state has reached the destination.

```python
class Problem:  # Class to define a problem and it's goal
    def __init__(self, island_map, initial, goal):
        self.island_map = island_map  # Map
        self.initial = State(initial[0], initial[1], island_map)  # Initial state
        self.goal = State(goal[0], goal[1],  island_map)  # Goal

    def goal_test(self, current):  # Check if a state is a goal state
        return current.X == self.goal.X and current.Y == self.goal.Y
```

In addition, the State class generates states used by the problem. It includes the successor method, responsible for identifying valid successors in the four main directions within a specified distance (represented as a factor of cells) with a maximum slope that is allowed to be traversed. When called from a state, it goes through every direction, calculating the new coordinates values, creating the new states and then checking the slope traversed and lastly checking, with the use of the umt_YX() function, that neither the given state nor the successor has a *nodata_value*. Finally, the information related to the successors (direction of movement used when generating it, the state associated with it, the distance traveled and the slope traversed) is stored in a classed that we decided to call it as The Descendant.

```python
def successors(self, factor, max_slope):  # Generate the valid successors
    directions = ['N', 'E', 'S', 'W']  # Possible cardinal directions
    successors = []  # List to store all the successors
    direction_changes = {  # Dictionary with the increments or decrements of the coordinates
    'N': (0, 1),
    'E': (1, 0),
    'S': (0, -1),
    'W': (-1, 0)
    }
    for direction in directions:  # Go through all possible directions
        change_X, change_Y = direction_changes.get(direction, (0, 0))  # Get the chnages on the dictionaries

        succ_X = self.X + self.island_map.size_cell * factor * change_X  # Find the X coordinate of the succesor
        succ_Y = self.Y + self.island_map.size_cell * factor * change_Y  # Find the Y coordinate of the succesor

        succesor_state = State(succ_Y,succ_X, self.island_map)  # Create the succesor state with the updated coordinates
        slope = abs(  # Calculate the slope |umt_YX orginial - umt_YX succesot|
            self.island_map.umt_YX(self.Y, self.X) - self.island_map.umt_YX(succ_Y, succ_X)
        )
        # Check if the succesor is valid (the origin and the succesor are in the map and the slope is acceptable)
        if (self.island_map.umt_YX(self.Y, self.X) != self.island_map.nodata_value and
            self.island_map.umt_YX(succ_Y, succ_X) != self.island_map.nodata_value and
            slope < max_slope
        ):
            descendant = Descendant(direction, succesor_state ,self.island_map.size_cell * factor, slope)
            successors.append(descendant)  # Add the descendant
    return successors
```

Task 3: Search Algorithm

For the third task, we're asked to create a search algorithm to go from a given origin state to a goal state using an organizable data structure for a frontier and allowing the use of three different strategies. This was implemented using eight classes *(QueueElement, PriorityQueue, SearchStrategy, BFS, DFS, UCS, TreeNode and VisitedTreeNodes*) located across the *search.py*, *strategy.py* and *priority_queue.py* files.

The algorithm starts in the *search.py* file with a method outside any class called search_algorithm() that performs the search of a given problem with the specified strategy, taking jumps of factor cells without overcoming slopes bigger than the maximum specified and making the maximum depth of the tree explored.

```
def search_algorithm(problem, strategy, factor, max_slope, max_depth):  # The search
    visited_tree_nodes = VisitedTreeNodes()  # Create the visited node list
    frontier = PriorityQueue(keys=[lambda item: item.frontier_value,  # Create the frontier priority queue
        lambda item: item.id])

    frontier.insert(TreeNode.root(problem.initial, strategy)) # Insert the root node to frontier
    tree_node_id = 1
    while not frontier.is_empty():  # If the frontier has items

        tree_node = frontier.remove()  # Remove one element from frontier
        state_id  = tree_node.state.get_id()  # Get the state

        if problem.goal_test(tree_node.state):  # If the retireved node's state the goal, return it
            return tree_node

        if tree_node.depth == max_depth:  # Check if the depth of the node is the maximum depth
            continue

        if visited_tree_nodes.belongs(state_id):  # Check if the state has already been visited
            continue

        visited_tree_nodes.insert(state_id) # Add it to the visited list

        for descendant in tree_node.state.successors(factor, max_slope):  # Add the descendants to frontier
            frontier.insert(tree_node.create_child(tree_node_id, descendant))
            tree_node_id += 1  # Increment the id of the next node
```

Firstly, it starts by creating the visited node list with the corresponding class that stores nodes that have, as the name suggests, already been considered in the search, allowing users to add to the list and check if a given state is part of the visited tree nodes.

```
class VisitedTreeNodes:  # Class to store the already visited nodes in the search
    def __init__(self):
        self.visited_tree_nodes = {}

    def belongs(self, state_id):  # Check if a given node has been visited or not
        return state_id in self.visited_tree_nodes and self.visited_tree_nodes[state_id]

    def insert(self, state_id):  # Add a node to the visited list
        self.visited_tree_nodes[state_id] = True
```

After the list is created, the root node of the tree is created and added to the frontier, which in this case is implemented as a priority queue in the *priority_queue.py* file which will be explained later.

Before that, it is necessary to mention the implementation of the tree nodes which basically serve as wrappers for state, including all the necessary components for the node. Taking into account this, the

initial step involves adding the root node to the frontier. The search algorithm then follows a loop that first verifies if the frontier is empty. If not, it removes the initial node from the frontier and processes its states. Afterward, it performs three checks: first, it evaluates if the current node matches our goal, returning that tree node if found. Second, it checks if the node has reached the maximum allowed depth, skipping it if true. Lastly, it skips the node if it's encountered this state before. The tree node is then added to the visited list, and its descendants are calculated and incorporated into the frontier as tree nodes. It's crucial to note that if the function doesn't return the goal tree node, it means that a solution with the conditions given doesn't exist.

In the creation of a tree node, as previously mentioned, these nodes possess several attributes: a linked state object, a unique identifier, a strategy designation, a distance cost, the maximum slope surpassed up to that point, the node's depth within the tree, its parent node, the movement direction employed to reach it, the frontier value, and a heuristic that will be explained in task 4

```python
class TreeNode:  # Class that stores information related with the tree nodes
    def __init__(self, state, strategy, id = 0, distance_cost=0, maximum_slope_cost=0,
                 depth=0, parent=None, action=None, frontier_value=None, heuristic=None):
        self.state = state  # State
        self.id = id  # Given Id to identify it
        self.strategy = strategy  # The strategy used
        self.distance_cost = distance_cost  # The distance cost
        self.maximum_slope_cost = maximum_slope_cost  # The max slope cost
        self.depth = depth  # Depth in which the node is located
        self.parent = parent  # The parent node
        self.action = action  # In what direction we moved
        self.frontier_value = frontier_value  # Value of the frontier
        self.heuristic = heuristic  # Heuristic

        self.frontier_value = self.strategy.frontier_value(self)
```

Moreover, with respect to the frontier value, we decided to implement the *SearchStrategy* as an abstract class which has the abstract method **frontier_value()** with computes the frontier using different ways based on the chosen strategy. For instance, the reason why we decided to use this pattern was because it improves modularity and flexibility in our code. By employing this method, it becomes easier to add new functionalities, such as integrating heuristics for task 4 or selecting the appropriate strategy by the user later on. This simplifies the process, making it straightforward and practically effortless

Furthermore, the uniformed strategies and their corresponding frontier values are the following:

- **BFS**: the value is calculated as just the depth of the given node
- **DFS**: the value is calculated as the inverse of the depth plus one
- **UCS**: the value is calculated as just the accumulated distance cost

```python
class SearchStrategy:  # Class to process each of the different strategies

    @abstractmethod
    def frontier_value(self, tree_node):  # Get the frontier value depending on the strategy
        pass

class BFS:  # Value = depth
    def frontier_value(self, tree_node):
        return tree_node.depth

class DFS:  # Value = 1 / (depth + 1)
    def frontier_value(self, tree_node):
        return 1 / (tree_node.depth + 1)

class UCS:  # Value = distance cost
    def frontier_value(self, tree_node):
        return tree_node.distance_cost
```

Besides this, in the file *priority_queue.py* is where we implemented the data structure responsible for sorting the values within the frontier of each node is a priority queue containing multiple keys with distinct priorities. The primary priority is given to the minimum frontier value, and in case of equality, it resorts to the minimum tree node ID. Various libraries and modules were available to handle this task, but one of the most efficient was the *bisect* module. This module maintains the list in a sorted order without requiring sorting after each insertion. It employs a bisection algorithm, where the library functions are specifically designed to locate insertion points. For instance, the *insort* functions have a complexity of O(n) because the logarithmic search step is dominated by the linear time insertion step. Therefore, the program expends minimal time in building the solution path, specially with the strategies which use heuristics. Lastly, as it is shown, the *QueueElement* class serves as a detailed representation of queue items. Its coding approach offers considerable flexibility, allowing sorting of various items based on a set of priorities

```python
class PriorityQueue:  # Class to implement the priority queue (frontier)

    def __init__(self, keys=None, compare_data=False):
        self.queue = []  # List to store the elements in the priority queue
        self.keys = keys

    def insert(self, element):  # Add an element to the queue
        order_elements = tuple(func(element) for func in self.keys) # Tuple for the ordering
        queue_element = QueueElement(order_elements, element)  # Create the queue element
        insort_left(self.queue, queue_element)  # Push the element to the prioty queue (back)

    def remove(self):  # Remove and return element from queue
        return self.queue.pop(0).data  # Pop an element (top) from the queue
```

Finally, the *TreeNode* class includes several functions. For instance, the *classmethod* root which generates a *TreeNode* instance which requires the initial state of the tree node and the search strategy as parameters. Additionally, a *'str'* method is used to display information about the tree node in a format way as well as a method that generates a list of elements from the root to a specific node, creating a visual representation of the path taken to reach the goal.

```
    @classmethod
    def root(cls, state, strategy):
        return cls(state, strategy)

    def solution_path_from_root(self):  # Generate the solution list with all the travelled nodes
        solution = []
        while self is not None:
            solution.insert(0, self)
            self = self.parent
        return solution

    def __str__(self):
        id_parent = self.parent.id if self.parent else None
        heuristic = f"{self.heuristic:.2f}" if self.heuristic is not None else 0.0

        return (
            f"[{self.id}][({self.distance_cost:.3f},{self.maximum_slope_cost:.3f}),{self.state.get_id()},{id_parent},{self.action}"
            f",{self.depth},{heuristic},{self.frontier_value:.4f}]"
)
```

Task 4: Heuristic

On this fourth and final task, we are asked to include into the search algorithm the A* and the greedy strategies together with 2 kinds of heuristics. This is implemented using five classes (Greedy, A*, Heuristic, Manhattan and Euclidean) across the strategy.py and heuristic.py files.

We make use of the already existing **search_algorithm()** function by adding more possibilities into the *SearchStrategy* class, this time the Greedy and A* classes that calculate their frontier value with the use of a heuristic function given when invoking the search algorithm.

```
class Greedy:  # Value = heuristic = heursitic value of a given node
    def __init__(self, heuristic):
        self.heuristic = heuristic

    def frontier_value(self, tree_node):
        tree_node.heuristic = self.heuristic.heuristic_value(tree_node)
        return tree_node.heuristic

class AStar:  # Value = distance cost + heuristic = distance cost + heuristic value of a given node
    def __init__(self, heuristic):
        self.heuristic = heuristic

    def frontier_value(self, tree_node):
        tree_node.heuristic = self.heuristic.heuristic_value(tree_node)
        return tree_node.distance_cost + tree_node.heuristic
```

In the case of Greedy, the value is calculated by using a heuristic function on the state but in A* the distance cost is added to the calculated heuristic value. Note that we have used the same pattern in where the heuristic value is an abstract method that will return the corresponding value depending on the circumstances.

Moreover, these heuristics can be categorized into two types, each serving a distinct purpose. The Manhattan heuristic is computed using the formula **sqrt((Dx-x)^2+(Dy-y)^2)**, while the Euclidean heuristic is calculated as **|Dx-x|+|Dy-y|**.

User manual

Upon starting the program, a resized map will be calculated to perform the search on and after a few seconds the user will be asked to input the following data about the search problem:

1. Introduce the YX starting coordinates
2. Introduce the YX goal coordinates
3. Introduce the factor for the size cell
4. Introduce the maximum slope
5. Introduce the maximum depth

After all the data has been confirmed to be valid, a menu will appear asking for the strategy to be followed to perform the search. The options are:

1. DFS
2. BFS
3. UCS
4. A* with Euclidean heuristic
5. A* with Manhattan heuristic
6. Greedy with Euclidean heuristic
7. Greedy with Manhattan heuristic
8. Exit

Selecting one of the 7 options will perform the search, the result of which will be shown on screen as a list from the origin state to the goal state formatted as such:
[ID][COST,STATE,ID_PARENT,ACTION,DEPTH,HEURISTIC,VALUE>]

In case there is no possible solution with the given parameters and/or strategy, the program will inform of it by displaying "**No Solution**".

After the search is complete, and the path has been displayed, the strategy menu will be displayed again, allowing the user to try the same problem on different strategies.
Pressing 8 on this menu will exit the application

If the map has to be changed or a new map is required to be created, it must be done through the code, modifying the marked values in *main.py*:

```
44      # CHANGE THESE VALUES TO CREATE NEW MAP
45      factor = 300
46      function = mean_func  # Possible functions: mean_func / max_func / min_func
47      name = "Zoom300Gomera"
48      resized_island_map = original_island_map.resize(factor, function, name)
49      # ====================================
```

Personal opinions

Alejandro

From my perspective, throughout this project, I've learned numerous functionalities of Python. I've come to appreciate its ease and flexibility in performing various tasks. Implementing the concepts taught in theoretical lessons, particularly generating tree nodes to complete the tree, fascinated me. For a programmer, applying previously learned knowledge in practice is crucial, and doing so was quite instructive. Initially, we encountered difficulties as we were unsure about organizing the code appropriately and collaborating effectively as a team. However, eventually we addressed this by initially seeking documentation and references. Understanding key aspects before commencing a task was essential, as we made the mistake of coding without a prior plan; essentially, we were improvising. So at the end, we invested several weeks in strategizing, researching documentation, and investigating deep into the project's requirements. Once everything was prepared, we started implementing the rest of the project without major difficulties.

Adrian

In my opinion, this project was a very long but useful lab project. It has helped me improve my knowledge on state search problems, on how by understanding the contents taught in theoretical classes make it easier to develop the different tasks, and very satisfactory when in the implementation of the functionalities you understand why you have to follow a similar approach to other theoretical exercises. Also, the theoretical classes coordinated with the laboratory help us, too.
In addition, using python with its extensive library has helped us a lot during the development of the different functionalities, as well as the documentation to give us a boost to implement the classes needed. Finally, the distribution of the work in four tasks, I think, was very helpful resulting in what I believe is a good-organized code, even though it wasn't quite clear at the beginning but, in future projects we could take as reference this project's organization.

Sergio

I think that, given the complexity of the problem, python was one of if not the most appropriate languages for the resolution as with not only it's easy to understand syntax but also its extensive module library, even the most complex tasks were a lot easier to perform, specially the node tree. The resolution of the problem itself also gave some new insights on how to approach search problems, that up to this point had always been solved in what for me was a more difficult to understand using recursive algorithms. In regard to the actual process of solving it, the division of the problem in 4 incremental tasks was definitely the right call as the project as a whole on it’s own could end up being quite suffocating to approach, but the individual tasks help to perfect the basic building blocks upon which the next task will rest, easing the process of the implementation.

Work distribution

In general, the work was carried out in a collaborative way and all the participants integrating the group were involved in development of the lab project. More specifically, the distribution has been shared in the following way:

Alejandro del Hoyo Abad: In this case, I think that I did a little bit more than the rest, especially in some parts of the implementation such as the search algorithm for the tree nodes. Moreover, in other aspects of the project such as the documentation, testing the results, resolving bugs, and implementing the initial tasks, I think I made an equivalent contribution compared to my teammates.

Adrián Gómez del Moral Rodrígez-Madridejos: In my case, in the early phase of the project I was more involved with the implementation but, during the last tasks my lack of knowledge in Python make not as useful as at the beginning so, I tried to help more in the design of the tasks for the implementation.

Sergio Pozuelo Martín-Consuegra: In this case, although in the later tasks of the implementation my role was a bit more passive, I think my involvement on the earlier ones and documentation was on par with my teammates.