Intelligent Systems

rubicks-cube

Beka Bekeri
Samuel González Linde

Repository:
https://github.com/Samuglz6/rubiks-cube

Escuela Superior de Informatica
Universidad de Castilla-La Mancha

6 October 2019

Contents

1 Application Requirements

The goal of the laboratory assignment is to implement a program that is able to solve a Rubik's cube with an optimal solution using searching techniques. The cube dimensions are going to be NxNxN and the value of N is going to be determined by the json file provided to us.

So the representation of the cube must be carried on, so as the movements of the cube as it is shown in *Figure 1*.

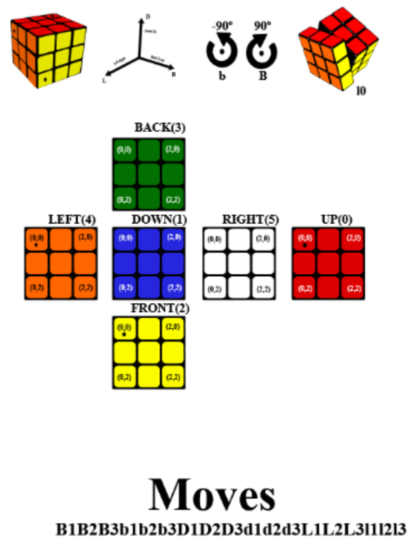
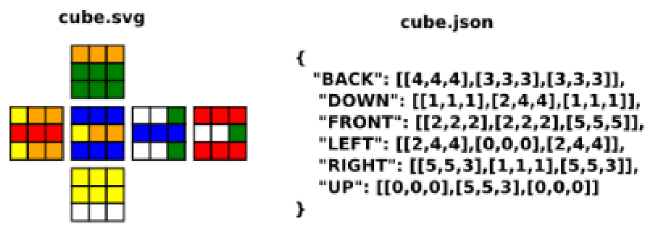


Figure 1: Extract from File1.svg

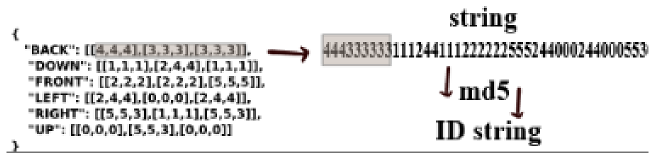
The state of the cube is going to be defined by the json we have talked about previously. So taking a look to the Figure 2 we can have an idea of how the structure of the json file is going to be so as which number represents each color.



```
{
  "BACK": [[4,4,4],[3,3,3],[3,3,3]],
   "DOWN": [[1,1,1],[2,4,4],[1,1,1]],
   "FRONT": [[2,2,2],[2,2,2],[5,5,5]],
   "LEFT": [[2,4,4],[0,0,0],[2,4,4]],
   "RIGHT": [[5,5,3],[1,1,1],[5,5,3]],
   "UP": [[0,0,0],[5,5,3],[0,0,0]]
}
```

Figure 2: Extract from File2.svg

For every state, we are going to work with a unique identification, the ***md5*** code, which is going to be generated by concatenating every number contained in each face of the cube as shown in *Figure 3*.



```
{
 "BACK": [[4,4,4],[3,3,3],[3,3,3]],
  "DOWN": [[1,1,1],[2,4,4],[1,1,1]],
  "FRONT": [[2,2,2],[2,2,2],[5,5,5]],
  "LEFT": [[2,4,4],[0,0,0],[2,4,4]],
  "RIGHT": [[5,5,3],[1,1,1],[5,5,3]],
  "UP": [[0,0,0],[5,5,3],[0,0,0]]
}
```

```
444333333111244111222222555244000244000553
```

Figure 3: Extract from File2.svg

The searching strategies that are going to be implemented for solving the cube are the following ones:

- Breath First Search (BFS)
- Depth First Search (DFS)
- Uniform Cost Search (UCS)
- Greedy
- A*

In order to find a solution by applying one of those strategies we are going to have nodes which are going to be used in a tree generated by the selected strategy.

2 Programming Language

First task, and one of the most important tasks we have to deal with, is the selection of the programming language we are going to be working with.
In spite of the freedom of choice of programming language, we have always had in mind from the very beginning the use of the language **'python'**.

The main reasons is the ease to work with the data structures, which is a feature provided by the language itself and also the versatility of the libraries that we can find and that surely can help us to perform the tasks so as its ease to install them and import them to the project.

3 Libraries

In this section we are going to describe the libraries used along the development of the project.
In some cases we are going to need to install a specific library in order to use it. This task can be carried on by following the next short steps:

```
Open a new terminal/command-line, then execute:

Linux:        pip install library_name
Windows:      py -m pip install library_name
```

Where library_name is the name of the library we want to get install.

3.1 json

To help us with the management of the json files we have decided to use the **'json'** library which allows us to read json files and store them in data structures ready to be used by our program.
This library also helps us to write our data into json files so we can have an easy track of the results we are receiving from the program.

3.2 hashlib

It is a library for encoding messages in different types of codes.
In our case and as we need to encode the cube object into an **md5** representation, we have used this library for this only purpose.

3.3 os

For everything related with the files and directories we have used **'os'**. This library allows us to get the files in a certain path, its very useful when we are looking for json as inputs for the program. It is also very useful when we have a tree folder' structure project as it allows us to add the path where we have to look for to use some specific libraries (as we have done with our test classes).

3.4 sys

We have used this library in the *JsonManager* class.
The purpose of the use of this library is to get information about the operative system in order to manage the execution of the program as the path of the working directory is taken for the execution of the program and the path changes between **Windows** and **Linux**.

The main reason to use it, it is because the program was developed to be easy used in a Linux terminal, but as we also had to code and execute in a windows machine.

3.5 numpy

One of the core libraries we are going to use is **'numpy'**, that gives us powerful functionalities with respect on N-dimensional arrays. We are going to use it to make the cube movements possible as it grants us the 90^o matrix rotation of each face.

3.6 copy

This library has the functionalities of returning a **shallow copy** or a **deep copy** of an object.
A shallow copy constructs a new compound object and then inserts references into it to the objects found in the original.
A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.
In this case, as we need copies that does not refers to the original we have used the deep copy functionality, because with the shallow copy any change made in an object have repercussions into the original.

3.7 Used libraries for the data structure

For the frontier data structure we have selected a short variety of libraries and then tested them and compared between themselves in order to select one.

The following libraries are the mentioned above:

- **sortedcontainers**: It includes all kind of sorted colections, such as queues, lists, sets...
  In our case SortedList adequates to our problem as we are looking for a data structure capable of have a sorted collection of items.
  By passing a key to the SortedList we can have a SortedKeyList which is going to sort the items inside by any value we want to use.

- **heapq**: It implements a priority queue algorithm including the method to order the elements of the queue.
  The only problem that we finally have encountered to this library once we have used it, is that the only item that is ordered is the first one.
  As we want to have the whole set of elements ordered, this was the least valued candidate.

- **bisect**: This library helps us to keep a list ordered after every insertion.
  Its called bisect due to the mechanisms that uses to keep the list ordered which is a basic bisection method.

3.8 Selected library

After testing every one of the 3 libraries we came up with an idea of which one would fit better.
As we have mentioned **heapq** had some characteristics that were kind of the idea we were looking for but not the exact ordered data structured we needed.
We have also encountered some troubles with the library **bisect** while ordering the elements inside the frontier. After testing it and using it for a while we discovered that at some point the algorithm stops ordering properly.
So finally, the one that performs the best is the SortedKeyList (SortedList + key) from **sortedcontainers**

4 Structure of the Artifacts

4.1 Cube

This is our core class, the *Cube.py* class. This class describes the state of the cube as it represents the values each face has and the dimension of the cube.

Attributes belonging to this class:

- **Faces:** a dictionary where every face-name is going to be stored as a key and the value corresponding to that face is going to be a list of lists with the value of each cell.
- **size:** the dimension of the cube

```
def __init__(self, json):
       self.size = len(list(json.values())[0])
       self.faces = json
```

The methods we can find in this class are some of the basic operations the cube has:

- **b:** counter clockwise movement of the rubik's cube. Takes as input a number between 0 and n(dimension of the cube).

```
def b(self, num):

       if num == 0:
           self.faces["BACK"] =
               np.rot90(self.faces["BACK"]).tolist()
       if num == self.size - 1:
           self.faces["FRONT"] =
               np.rot90(self.faces["FRONT"]).tolist()

       for x in range(self.size):
           bubble = self.faces["LEFT"][num][x]
           self.faces["LEFT"][num][x] = self.faces["DOWN"][num][x]
           self.faces["DOWN"][num][x] = self.faces["RIGHT"][num][x]
           self.faces["RIGHT"][num][x] = self.faces["UP"][num][x]
           self.faces["UP"][num][x] = bubble
```

- **B:** clockwise movement of the rubik's cube. It also takes as input a number between 0 and n.

```
def B(self, num):
      if num == 0:
          self.faces["BACK"] = np.rot90(self.faces["BACK"],
              -1).tolist()
      if num == self.size - 1:
          self.faces["FRONT"] = np.rot90(self.faces["FRONT"],
              -1).tolist()

      for x in range(self.size):
          bubble = self.faces["LEFT"][num][x]
          self.faces["LEFT"][num][x] = self.faces["UP"][num][x]
          self.faces["UP"][num][x] = self.faces["RIGHT"][num][x]
          self.faces["RIGHT"][num][x] = self.faces["DOWN"][num][x]
          self.faces["DOWN"][num][x] = bubble
```

Following the indications given by the *File1.svg*, every method related with the movements is going to work this way:

- We have three types of movements: **Back-Front**, **Down-Up** and **Left-Right**
- Each movement is represented by the letters **B/b**, **D/d** and **L/l** respectively.
- If the letter is a lowercase that means the movement is going to be counter clockwise, otherwise if its an uppercase letter is going to be a clockwise movement.
- The cube rotates all the possible segments for a specific movement, it means, that if we have a 5x5 cube, we need to rotate 5 tiles per face, in the same way, well need to move only 3 tiles per face in a 3x3 cube
- We have considered 2 special cases, if ure moving the first column, the leftmost one, or the last column, the rightmost one, since we also have to rotate the whole left or the right face.

4.2 Main

As the name says, this is the main class from our project, the one in charge of running the whole system. In this class we can find 3 main functionalities.

- The retrieval of the data, which is how we obtain the information we need to perform the search. The first thing we need to know is what arguments do we need to perform the search properly:

- **Json**: which is the .Json file that contains the problem of this execution.
- **Strategy**: the strategy the search is going to follow.
- **Pruning**: whether the pruning is going to be applied or not, this is also known as optimization.
- **Max-depth**: in case that the search selected previously is **DLS**, a maximum depth has to be settled.
- **Increment**: in case that the search selected preciously is **IDS** an increment for the depth in each iteration has to be settled.

Here we can see how its been implemented:

```
def askData():
    json = jManager.jsonSelection()
    switch = {0: ['Breath First Search', 'BFS'], 1: ['Depth First
        Search', 'DFS'], 2:
    ['Depth Limited Search', 'DLS'], 3: ['Iterative Deepening
        Search', 'IDS'],
    4: ['Uniform Cost Search', 'UCS'], 5: 'A*', 6: 'Greedy'}

    while 1:
        print("\nSelect now the searching strategy:")
        for element in switch.keys():
            if(element < 5):
                print(element,' - ',switch.get(element)[0],
                    '(',switch.get(element)[1],')')
            else:
                print(element, ' - ', switch.get(element))
        strategy = input("Selection: ")

        if strategy.isdigit():
            strategy = int(strategy)
            if strategy not in switch:
                print("ERROR. The number introduced is not valid.")
            else: break
        else:
            print("ERROR. The introduced strategy is not valid")


    while 1:
        pruning = input("Do you want to pruning? (y/n)\n")
        if pruning == 'y' or pruning == 'yes':
            pruning = 1
            break
        elif pruning == 'n' or pruning == 'no':
            pruning = 0
            break
        else: print("ERROR. The answer is not valid. You can use:
             yes/y or no/n.")
```

```
if strategy == 2 or strategy in switch.get(2):
    max_depth = int(input("Choose the maximum depth: "))
else:
    max_depth = 6
if strategy == 3 or strategy in switch.get(3):
    increment = input("Specify the increment: ")
else: increment = 1
return strategy, pruning, json, max_depth, increment
```

- The search.
  The search is divided in two methods:
  - **search**: this is more like a wrapper for the actual search, with this method we can control the flow execution of the program in the case that we have an **IDS** or a **DLS** with the *while* loop.
    This are the input variables:
    - **problem**: the internal representation of the problem we are looking to solve, what is our initial cube
    - **strategy**: the selected strategy to perform the search
    - **max-depth**: if the strategy selected is **IDS** this variable will have a value that will be the limit in relaunch of the search with a higher depth limit.
    - **increment**: if the strategy selected is **IDS** this variable will contain a value that will make the depth limit of the relaunched search increment such that the new limit will be *new depth = last depth + increment*
    - **prunning**: this is a variable that will tell the search if prunning has to be applied or not, if the value is 1, prunning has to be applied.

```
def search(problem, strategy, max_depth, increment, pruning):
    actual_depth = increment
    total_nodes = 0
    solution = None
    while (not solution) and (actual_depth <= max_depth):
        solution = bounded_search(problem, strategy,
            actual_depth, max_depth, pruning, total_nodes)
        actual_depth = actual_depth + increment

    return solution
```

– **bounded-search**: this is the general search algorithm instantiated for our problem.
  This are the input variables:
  * **problem**: the internal representation of the problem we are looking to solve, what is our initial cube
  * **strategy**: the selected strategy to perform the search.
  * **max-depth**: if the strategy selected is **IDS** this will be the variable that will be the limit for this specific execution of the search.
  * **prunning**: this is a variable that will tell the search if prunning has to be applied or not, if the value is 1, prunning has to be applied.
  * **total-nodes**: this variable is used, both as an id for the generated nodes, and as a counter for the nodes generated

```
def bounded_search(problem, strategy, actual_depth, max_depth,
     pruning, total_nodes):
    frontier = Frontier()
    init_node = TreeNode(total_nodes, problem.initial,
         strategy)
    frontier.insert(init_node)
    solution = False
    while (not solution) and (not frontier.isEmpty()):
        actual_node = frontier.remove()

        if(problem.isGoal(actual_node.state)):
            solution = True
        else:
            successors_list =
                 problem.stateSpace.successors(actual_node.state,
                 actual_node.d, max_depth)
            node_list = []
            for (action, state, cost) in successors_list:
                total_nodes += 1
                node = TreeNode(total_nodes, state, strategy,
                     actual_node, cost, action)
                node_list.append(node)
            if pruning == 1:
                for node in node_list:
                    if node.state.md5 not in problem.visitedList:
                        frontier.insert(node)
                        problem.visitedList[node.state.md5] =
                             node.f
                    elif strategy == 1 and node.f >
                         problem.visitedList[node.state.md5]:
                            frontier.insert(node)
                            problem.visitedList[node.state.md5]
                                 = node.f
                    elif node.f <
                         problem.visitedList[node.state.md5]:
                            frontier.insert(node)
                            problem.visitedList[node.state.md5]
                                 = node.f
            else:
                for node in node_list:
                    frontier.insert(node)
    if solution:
        return actual_node
    else:
        return None
```

- If a solution has been found, the **Main.py** class will be in charge of generating the solution sequence and write it in a json format and in a .txt file. Remember that the way we obtain a solution is with the attributes *parent and action* of the treenode, so that if we consult repeatedly the parent node until there is no parent, it means that we have obtained the full path until the starting node.

```
def generateSol(solution, node):
    if node is None:
        solution = None
    elif(node.parent is None):
        solution.append("["+str(node.id)+"]([" +str(node.action)+"]"+str(node.state.md5)+",
            cost = "+str(node.pathCost)+", depth = "+str(node.d)+",
            f = "+str(node.f)+")")
    else:
        generateSol(solution, node.parent)
        solution.append("["+str(node.id)+"]([" +str(node.action)+"]"+str(node.state.md5)+",
            cost = "+str(node.pathCost)+", depth = "+str(node.d)+",
            f = "+str(node.f)+")")

def writeSolution(solution, node, strategy):
    switch = {0: ['Breath First Search', 'BFS'], 1: ['Depth First
        Search', 'DFS'], 2:
    ['Depth Limited Search', 'DLS'], 3: ['Iterative Deepening
        Search', 'IDS'],
    4: ['Uniform Cost Search', 'UCS'], 5: 'A*', 6: 'Greedy'}

    if node is not None:
        print("A solution has been found.")
        print("The solution has been saved:
            rubiks-cube/output/solution.txt")
        print("You can also check the result of the faces:
            rubiks-cube/output/solution.json ")

        jManager.jsonWriting('solution', node.state.current)
    else:
        print("There is no solution for the options you have
            chosen.")
        jManager.jsonWriting('solution', None)

    with open(jManager.currentDirectory()+"output/solution.txt",
        "w+") as text_file:
        text_file.write(' '.join(switch[strategy]))
        text_file.write('\n====================\n')
        text_file.write('\n'.join([element for element in
            solution]))

    text_file.close()
```

4.3 JsonManager

The purpose of this class is being the intermediary between the execution of the program and the output in *.Json* files. It has tow main tasks:

- Selecting the input file for the problem which is in a *.Json* file.
- Creating an output *.json* with the solution and execution of the program once its done.

4.4 State

This class is created to represent a state in our search, so a state in our search has this attributes:

- Cube: the internal representation of the cube
- MD5: which is the representation of the cube in a string, so we can use it as a unique id in the search.

In this class we can also see how we obtain the md5 code of the cube

```
def generateCode(self):
        string = ''
        for value in self.current.faces.values():
            for x in value:
                string += ''.join(map(str, x))
        return hashlib.md5(string.encode('utf-8')).hexdigest()
```

4.5 State Space

This class has two main functionalities

- Generating the initial state space, what means reading the json file to obtain the actual state space, what will be the start of our class **Problem.py**
- Once the search is being done, this class will also generate the succesors, so it cointains the generation of the successors

```
def successors(self, state, depth, x):
        nodes = []
        if depth+1 == x:
            return []
        else:
            for move in state.current.validMovements():
                aux = State(state.current.clone())
                method = getattr(aux.current, move[0])
                method(int(move[1]))
                newState = State(aux.current)
                acc = move
                costAct = 1
                nodes.append((acc, newState, costAct))
            return nodes
```

As we can see, this method takes as input the actual state of the cube and generates all the possible successors, depending of the valid movements. It also takes some parameters for some specific searchs, such as the **DLS** where if the state we are trying to expand is in the limit of the depth it will not return any successor.

4.6 Problem

This class has two main functionalities:

- Containing the internal representation of our problem, what means that this class has to contain information about the situation of the initial cube, and what is considered as a solution for our problem
- This class, as said before, contains a mechanism to know whether a cube has reached our goal or not, and that is made by this method:

```
def isGoal(self, state):
        goal = False

        for face in state.current.faces.values():
            color = set()
            for x in range(state.current.size):
                color = color.union(set(face[x]))
            if (len(color) == 1):
                goal = True
            else:
                return False
        return goal
```

4.7 TreeNode

This class represents a node in our search tree, so apart from containing the internal representation of the cube it has more information:

- **id**: which is the id of the generation of that node.
- **parent**: link to its parent node.
- **state**: the actual state of the cube and its md5 representation
- **pathCost**: this is the real cost that takes to get to this node
- **d**: actual depth.

- **action**: the action that has been performed in parent to generate this successor.
- **f**: this is the value that tries to represent the distance to the solution and the variable that is used to index the nodes to be selected

```
def __init__(self, id, state, strategy, parent=None, cost=0,
    action=None, depth=0):
       self.id = id
       self.parent = parent
       self.state = state
       if self.parent is not None:
           self.pathCost = self.parent.pathCost+cost
           self.d = self.parent.d+1
       else:
           self.pathCost = cost
           self.d = depth
       self.action = action
       self.f = self.selection(strategy)
```

We can observe that f has a special method to be generated, lets see it in depth

```
def selection(self, strategy):
        if strategy == 0: return self.d
        if strategy == 1: return round(1/(self.d+1),2)
        if strategy == 2: return round(1/(self.d+1),2)
        if strategy == 3: return round(1/(self.d+1),2)
        if strategy == 4: return self.pathCost
        if strategy == 5: return self.pathCost+self.calculateHeuristic()
        if strategy == 6: return self.calculateHeuristic()
```

Here we can observe that the f does not only depend on the cost and in the depth, depending on the strategy, but also in the informed searchs we have a value to be considered, the heuristic value of that node, lets see what is this heuristic:

```
def calculateHeuristic(self):
       h = 0
       entropy = {"BACK": 0,"DOWN": 0,"FRONT": 0,"LEFT": 0,"RIGHT":
           0,"UP": 0}
       N = self.state.current.size

       #As we are going to calculate for each face we set a for loop
       for face in self.state.current.faces:
           counter = {0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0} #n of colors
               in a face

           #Now we get the number of tiles of each color for a specific
               face
```

```
        for list in self.state.current.faces[face]:
            for color in list:
                counter[color] += 1

        #calculation of the entropy for each face
        for c in range(6):
            if counter[c] > 0.0:
                entropy[face] = entropy[face] + counter[c]/(N*N) *
                    math.log(counter[c]/((N)*(N)),6)

    #Addition of the entropy of each face in order to calculate h
    for face in entropy:
        h += abs(entropy[face])

    return round(h,2)
```

4.8 Frontier

The funcionality of this class is to offer to the search a data structure such that can be ordered depending on some value, so, the search can instantiate the frontier of the search with this class.

We have selected a data structure called *SortedKeyList* that once it receives a value as a **key** the elements that will be inserted in the list will be ordered in order of that value, we can see how we use the attribute f as the primary key of the list:

```
def create(self):
    return SortedKeyList(key=TreeNode.getF)
```

5 Testing Classes

As the complexity of the problem was quite big, even more at the time of designing the movements, we have created several testing classes to test things. As those test classes are not asked for the project (but helped a lot along the development) we are going just to give a brief commentary about each one. In order to have a bit more information about them, they can be tested by executing them.

```
$ cd ./testing
$ python3 'Test_Name.py'
```

The testing classes we have implemented are:

- **Frontier**: in order to select a proper data structure we had to try several and select the one that had the better performance, not the fastest one, we also need to implement an ordering in the list.
- **Goal function**: in the file *Test-Goal.py* we can find how we tested the goal function so that we know that if we cant find the goal its not because of the design of that function.
- **Heuristics**: Once implemented the heuristic by using this test we can check by using examples if it is computed properly without any errors. We compute the f value of the initial state of any json then compare them to the examples we have in the folder *examples*
- **Movements**: the correct design of the movements is crucial for the correct development of the search, so we had to be sure that the movements were exactly what we needed.
- **Stress test**: since we are going to deal with big numbers of states, we also needed structure that supported this, so in the file *Test-Stress.py* we have used it to test a few different data structures with their insertion times, maximum capacity...
- **Successors**: just to make sure for every different N-sized cube, successors are being generated properly we have developed a little test just to make sure the output its what we need.
- **x10Cube test**: we have been provided with a json of a 10-sized cube as well as their moves with the new jsons generated. So we have compared the md5 representations resulting of applying our movements to the initial json with the md5 generated by the jsons of each move.

6 User's Manual

6.1 System requirements

In order to be able to execute the program you should have installed on your computer Python3.

Also some packages are going to be used by the program. Take into account that some of them have to be installed in order to be able to use them. The following packages are used by the program:

- json
- hashlib
- os
- sys
- numpy
- copy
- sortedcontainers

6.2 How to install Python3

Installing Python3 in Ubuntu it's an easy task:

```
$ sudo apt-get update

$ sudo apt-get install python3.6
```

6.3 How to install packages

6.3.1 Sortedcontainers

```
$ sudo pip install sortedcontainer
```

6.3.2 Numpy

```
$ sudo pip install numpy
```

6.4 Executing the program

Now we are going to execute the program. To do that we have to be either in the project folder (*rubiks-cube*) or the source code folder (*rubiks-cube/src*).
In both cases we are going to execute the *Main.py* file that is in the *src* folder.

For the first one the execution will be as follows:

```
$ python3 ./src/Main.py
```

For the second option, will be like this:

```
$ python3 Main.py
```

Any other way of executing the program could lead to problems in the project resources management.

Now you just have to follow the instructions given by the program:

1. Select the json of the cube to be used in the program
2. Select the searching strategy to find the solution
3. Choose if you want to use pruning or not
4. Wait for the solution
5. You can consult the solution of the cube in the project's folder output:
    - *solution.json* shows the final result of the cube.
    - *solution.txt* has the information of the path to reach the solution.

7 Personal Opinion

We have to put a lot of different concepts that we have learned in other different subjects as **Data Structures** or **Programming Methodology** plus the knowledge that we have gained with this subject itself: **Intelligent Systems**. But the part that we think is the most important is that this lab project made us put all our efforts on trying to find and compare the things we need to use to look for the most appropriated ones and the most optimal.

As from the very beginning we have told to search for ordered data structures that are optimal in time insertion/deletion, we have also put a lot of attention on times, which in some real projects is a crucial characteristic.