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

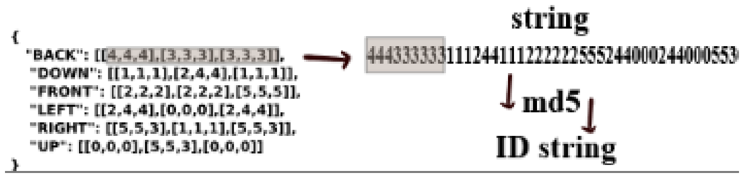
1 Task 1

1.1 Application Requirements

The goal of the laboratory assignment is to implement a program that is able to solve a rubick'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.

For this first laboratory task we are told to implement the internal representation for the cube and also a set of basic operations of the cube such as the movements for example. Another feature we are asked to implement is the ***json*** file treatment. We must be able to read the json file we have talked about and create the cube state. The same in the opposite case, we must be able to write a json file to store the current state of the cube.

Finally, we are going to work with a unique identification for every state, the ***md5*** code which is going to be generated by concatenating every number contained in the cube state as shown in *Figure 1*.



```
{
  "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 1: Extract from File2.svg

1.2 Selection of the programming language

The first important task of this assignment is to select the programming language we are going to be working with. As the programming language is of free choice and from the very first moment of the project we had **'python'** in mind, we finally decided to use it.

The main reasons was 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.

1.3 Used Libraries

We are going to use five libraries that will help us along the development of the first task.

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

For everything related with the files and directories we have used **'os'**. This library allows us to get the files in a certain path, so its very usefull when we are looking for json as inputs for the program.

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

In order tu use numpy package we have to install it. To get numpy installed we are going to proceed as follows in the **terminal/cmd**:

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

We are going to use a library called **'pprint'**, which will give us the possibility to print in a more readable way dictionaries and lists. As its name suggests it allows us to pretty print data structures.

And finally, **'hashlib'** which is a library used for the encoding of messages in different types of codes, in our case md5.

```
import hashlib
import numpy as np
import json
import pprint
import os
```

1.4 Structures of the Artifacts

1.4.1 Cube Class

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 internal code with an

MD5 representation.

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.
- **md5:** the MD5 internal code of the cube's state.

```
def __init__(self,json):
    self.Faces = json
    self.md5 = self.generateMD5()
```

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

- **generateMD5:** as input it gets the object cube itself and it uses its attribute Faces in order to generate a string with the numbers of each face and then code it in a representation in md5. The library **'hashlib'** helps us in this task.

```
def generateMD5(self):
    cadena = ''
    for value in self.Faces.values():
        for x in value:
            cadena += ''.join(map(str,x))
    md5 = hashlib.md5(cadena.encode('utf-8')).hexdigest()

    return md5
```

- **b:** counter clockwise movement of the rubik's cube. Takes as input a number between 0 and 2.

```
def b(self, num):
    if num == 0:
        self.Faces["BACK"] =
            np.rot90(self.Faces["BACK"]).tolist()
        if num == 2:
            self.Faces["FRONT"] =
                np.rot90(self.Faces["FRONT"]).tolist()

        for x in range(3):
            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

        self.md5 = self.generateMD5()
```

- **B:** clockwise movement of the rubik's cube. It also takes as input a number between 0 and 2.

```
    def B(self, num):
        if num == 0:
            self.Faces["BACK"] =
                np.rot90(self.Faces["BACK"],-1).tolist()
        if num == 2:
            self.Faces["FRONT"] =
                np.rot90(self.Faces["FRONT"],-1).tolist()

        for x in range(3):
            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

        self.md5 = self.generateMD5()
```

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.
- Every letter has a value from 0 to 2, which represent the face is going to be moved. *Example: B0 is going to rotate Back face 90º; B1 is going to rotate the middle face (same way as B0); B2 is going to rotate Front face.*

1.4.2 Main Class

At first, in the main class we started to create everything related with the json file but finally we though about to make a separated class for that functionality. Since we have only implemented the basis of the project, we only check if the cube is well created as well as the reading operation of the json file.

```
def main():
    cube = Cube(jManager.jsonReading())
    print(cube.md5)
    pprint(cube.Faces)
```

1.4.3 JsonManager Class

This class, as its name suggests, its used to manage everything related with the json files.

For the moment, two methods have been implemented in this class in order to communicate our project with the input from a json file, and create an output in another json file:

- **jsonReading:** in this method, with the help of the package **'json'** and its method, **'json.load'** we charge in memory the representation of the cube. We also try to avoid errors with the seeking of the .Json file in the directory.

```
def jsonReading():
    print("Select the json file:")
    for file in os.listdir('../json'):
        if os.path.splitext(file)[1] == ".json":
            print('-'+os.path.splitext(file)[0])

    selected = input("Selected file:")
    json_file = '../json/'+selected+'.json'

    if json_file:
        with open(json_file) as output:
            data = json.load(output)
    return data
```

- **jsonWriting:** this method is used to write, in a previously opened .Json file, the new cube.

```
def jsonWriting(name, cube):
    with open(name+'.json','w') as file:
        json.dump(cube.Faces, file)
```

1.4.4 Testing Class

Last but not least, we have also decided to create a separate class for testing the movements: *Testing.py.*

This class is used to check wheter a movement is made in a properly way or not, since we are in the most early part of the development of our project we have just made a subgroup of all the possible action.

- **menu:** This is the menu of this class, here we can select if we want to check just one movement or if we want to check all the possible actions of the cube

```
def menu(cube):
    while True:
        selection = 0
        try: selection = int(input("Options for testing:\n\t1 -
            Test a move\n\t2 - Test every
            movement\nSelection:"))
            except ValueError:
            print("Error, selection must be integer")
            continue
        if selection == 1:
            testOneMove(cube)
            break
        if selection == 2:
            testingBackFront(cube)
            break
        else: print("Not a valid selection")
```

- **testOneMove:** this method is used when the user selects the first option of the menu, so only one action is going to be performed. In our case as only the movements [**"B0", "B1", "B2", "b0", "b1", "b2"**] are implemented, we can only choose one of them. So once a movement is selected, wheter its clockwise(B) or counter clockwise(b) the specific method is called from the Cube class.

```
def testOneMove(cube):
    moves = ["B0", "B1", "B2", "b0", "b1", "b2"]
```

```
while True:
    key = 0
    try: key = input("Select move type B0/B1/B2 (90) or
        b0/b1/b2 (-90):")
    except ValueError:
        print("Error, selection must be integer")
        continue
    if key in moves:
        if list(key)[0] == 'b':
            cube.b(int(list(key)[1]))
        else:
            cube.B(int(list(key)[1]))
        break
    else:
        print("Not a valid selection")

jManager.jsonWriting('testing', cube)
print("Results have been saved in testing.json")
```

- **testingBackFront:** this method is used to perform every possible action for the movement Back-Front, which means that movements B0-2 and b0-2 are going to be performed. *The sequence of movements B0B1B2b0b1b1 is going to be performed.*

```
    def testingBackFront(cube):
        for n in range(3):
            print("Aplying B%d movement" %n)
            cube.B(n)
            pprint(cube.Faces)

        for n in range(3):
            print("Aplying b%d movement" %n)
            cube.b(n)
            pprint(cube.Faces)
```

1.5 Execution of the program

Once we execute our Main.py class the following output is going to be displayed in the terminal:

```
line for executing: $ python3 Main.py

    Select the json file:
    -cube
    -testing
```

```
Selected file:cube
6b09b2076aa6a349c7ad3dd5cee99438
{'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]]}
```

As it can be shown we have a selection of any json file available on the folder */json* and once we have selected one, the md5 representation and the Faces are shown.

1.6 Testing example

For the testing part, if we execute the Testing.py the following output is displayed in the terminal:

```
line for executing: $ python3 Main.py

    Select the json file:
    -cube
    -testing
    Selected file:cube
    Options for testing:
   1 - Test a move
    2 - Test every movement
    Selection:1
    Select move type B0/B1/B2 (90) or b0/b1/b2 (-90):B0
    Results have been saved in testing.json
```

As we can see, as well as in the main execution we have the selection of the json file to work with and then we have 2 options available. In this case we have selected to Test a move and we have introduced the move B0.

The result is stored in the *testing.json*:

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

2 Task 2

2.1 Application Requirements

In this task we are asked to develop the next part of the system, which creating the main elements of the system, the artifacts, starting to use data structures for the implementation of the frontier and also testing our program to check for early fails so that they wont grow no more for the next tasks.

So as explained, in this part you are going to observe how the selection of the frontier has been made, how we implemented the artifacts and also how the main structure of the system has been created, **State space**, **Frontier**, **nodes of the search tree**, ...

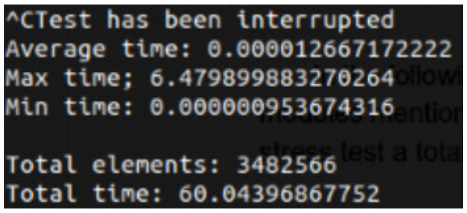
2.2 Used Libraries

2.2.1 SortedContainers

- pip install sortedcontainers

SortedContainers, is a Python library which includes all kind of sorted collections, such as queues, sets ....

From these librarie we decided to check the class **SortedList** since its description on the webpage seemed correct to our problem.



```
^CTest has been interrupted
Average time: 0.000012667172222
Max time; 6.479899883270264
Min time: 0.000000953674316

Total elements: 3482566
Total time: 60.04396867752
```

Here we can see the results of the testing of this data structure. As you can see the times of insertion and the total amount of elements are quite good, the problem comes with some specific insertion that can take way too long to be inserted, like in this case, where we have an insertion that took around 6 seconds, which is, obviously, a huge amount of time.

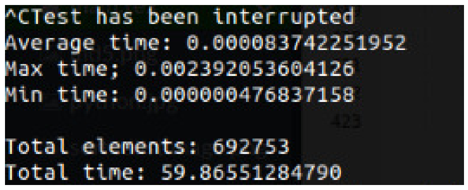
2.2.2 Heapq

Heapq this library is used to implement the priority queue algorithm including the method to order the elements of the queue.

The results of the testing has shown that the insertion times can be in the order of 16Million of elements in one minute, the problem comes with the ordering of these elements, which at some point gets weird and non reliable.

2.2.3 Bisect

Bisect is a python library that helps to keep a list ordered after every insertion, its called Bisect due to the mechanism that uses to keep the list ordered which is a basic bisection method



```
^CTest has been interrupted
Average time: 0.000083742251952
Max time; 0.002392053604126
Min time: 0.000000476837158

Total elements: 692753
Total time: 59.86551284790
```

2.2.4 Selected library

Aftrer testing the 3 data structures selected, the best, or at least the most stable results had been given by the Bisect package, so by now we are going to use this one as our data structure. The problem with the other two, Heapq and SortedContainer, have been, by now discarded, gave us a non-expected behaviour, since sometimes SortedContainer took too much to insert a single object and the Heapq ended up not giving back the list ordered.

2.3 Structures of the Artifacts

In this part, we are going to explain how we decided to solve and implement the main elements of the system.

2.3.1 State space

The state space, in a theoretical approach is the whole set of configurations that a specific problem could have, in our case, the cubes composition will determine it. This is represented in the class **StateSpace.py**.

```
def StateSpace():
    def __init__(self, json):
        print(json)

    def Successor(state):
        acc = 0
        newState = 0
        costAct = 0

        print(acc, newState, costAct)
```

We can observe the method Successor which will return the whole set of new states given an actual state, so it will apply all the possible actions to the state and will calculate the cost of those actions.

2.3.2 Problem

This element, represented in the class **Problem.py**, is used to represent the problem of this exact execution, it means, apart from having the initial state for the execution it has a mechanism to decide whether a goal has been reached or not and which one was the goal.

```
class Problem():
    def __init__(self, json):
        self.initial = State(Cube(json))
        self.stateSpace = StateSpace()

    def isGoal(self, state):
        goal = False
        for key in state.current.faces.keys():
            for element in state.current.faces[key]:
                if len(set(element)) == 1:
                    goal = True
                else:
                    return False
        return goal
```

With the method isGoal, we can check at any moment if the goal has been reached or not.

2.3.3 Node of the search tree

This element, implemented in the class **TreeNode.py**, is used as the nodes of the search trees, so apart from containing information about the domain (cube) it has additional information as the cost to reach it and the depth of that node.

```
class TreeNode:
    def __init__(self, state, cost, action, depth, parent=None):
    self.parent = parent
    self.state = state
    self.pathCost = cost
    self.action = action
    self.d = depth
    self.f = random.random() * 1000000
```

2.3.4 Frontier

In this kind of problems the frontier is an important element to take into account, since its correct development can increase the performance of the system, in our case we need to implement a frontier to insert the nodes that we have already visited, which are going to be inserted taking into account, by now, a random number *f*.

So taking this into account a testing has been made, the results will appear later

2.4 Testing classes

2.4.1 Frontier test

The purpose of this class is to check the correct behaviour of our frontier in its own operation, so as, the insertion, the removal of the elements and so on.

It also checks whether the fringe is ordered or not.

```
def main():
    problem = Problem(jManager.jsonReading("../../json/x4cube.json"))
    fringe = Frontier()

    for i in range(10):
        node = TreeNode(problem.initial, 0, 0, 0)
        fringe.insert(node)

    for element in fringe.frontier:
        print(element[0])

    for i in range(10):
        fringe.remove()

    print("isEmpty: ", fringe.isEmpty())
    print("isGoal: ", problem.isGoal(problem.initial))
```

2.4.2 Movement test

The purpose of this test is to check whether we are doing the movements properly or not. So the main functionality is basically applying each kind of movement once to a cube and see the results of these.

```
def testMove(cube):
    letters = ["B", "b", "D", "d", "L", "l"]
    moves = []
    for element in letters:
        for number in range(cube.size):
            moves.append(element + str(number))
    while True:
        key = 0
        print("\nOption's movements: ")
        print(",".join([item for item in moves]))
        try:
            key = input("Selection:")
        except ValueError:
            print("Error, selection must be integer")
            continue
        if key in moves:
            if list(key)[0] == 'b':
                cube.b(int(list(key)[1]))
            elif list(key)[0] == 'B':
                cube.B(int(list(key)[1]))
            elif list(key)[0] == 'l':
                cube.l(int(list(key)[1]))
            elif list(key)[0] == 'L':
                cube.L(int(list(key)[1]))
            elif list(key)[0] == 'D':
                cube.D(int(list(key)[1]))
            elif list(key)[0] == 'd':
                cube.d(int(list(key)[1]))
            break
        else:
            print("Not a valid selection")

    jManager.jsonWriting('../../testing/output/', 'testing', cube)
    print("Results have been saved in testing.json")
```

2.4.3 Stress test

The purpose of this test is to check the limit of insertions in our datas structure, and calculate at how much elements inserted the structure indeed or the memory starts to fail.

This is so important is this kind of problems because sometimes the state space can be such big that the structure cant deal with it.

```
def stress_test(p, f):
    total_nodes = 0
    t_avg = 0
    t_max = 0
    t_min = 9999
    try:
        while True:
            try:
                if total_nodes == 0:
                    node = TreeNode(p.initial, 0, 0, total_nodes)
                else:
                    node = TreeNode(p.initial, 0, 0, total_nodes)
                init_time = time.time()
                f.insert(node)
                end_time = time.time()
                total_nodes += 1
                current_time = end_time - init_time
                t_avg += current_time
                if current_time > t_max: t_max = current_time
                if current_time < t_min: t_min = current_time

            except MemoryError:
                print("Memory Full")
                printData(t_avg, t_max, t_min, total_nodes)
                sys.exit(1)
    except KeyboardInterrupt:
        print("Test has been interrupted")
        printData(t_avg, t_max, t_min, total_nodes)
```

Apart from these method we also obtain different statistical data about the insertion times.

```
def printData(avg, t_max, t_min, total):
    print("Average time: %.15f" % (avg / total))
    print("Max time; %.15f" % t_max)
    print("Min time: %.15f" % t_min)
    print("\nTotal elements: %d" % total)
```

3 Task 3

3.1 Application Requirements

For this delivery we need to develop the **basic search algorithm**, which is also known as the **generic search algorithm**. Here we can see the pseudo-code of the search algorithm

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return node
    fringe ← INSERTALL(EXPAND(node, problem), fringe)
end
```

This is the most simplified version of the search, in which we can observe some some of the artifacts that we defined in the last delivery, such as the *fringe* or the *node*.
Apart from implementing the basic search algorithm, we need to implement also some kind of strategies, which our search will have to follow, this strategies are going to be the next ones:

- **Breath-first search:** one of the classical search strategies, in which the search is performed uniformly by layers of depth, what means that a node of depth=2 will never be expanded before a node with depth=1.
- **Depth-first search:** the other one of the classical search strategies, here the search is performed in a upright way, what means that the search will be expanded in depth, the expanded node will be the first selected.
- **Depth-limited search:** the same as the DFS but with a limited depth, due to the inconvenience of this strategy
- **Iterative Deepening Search:** the same as DLS but increasing the depth, so the search will be executed several times increasing the delimiter depth
- **Uniform cost:** here the cost to reach the child node is always the same.

3.2 Algorithm description

In this part the development of the main search methods is going to be explained, as well as how we have modified the structure of it to perform with the strategies that we need to use.

First of all we need to explain what arguments needs the system to know how to perform, which are:

- File: the file from which the problem is going to be obtained
- Strategy: which strategy has to follow the search
- Depth: The maximum depth for the depth-first-search, depth-limited-search and iterative-deepening-search, in the last case, also the increment of the depth will be needed
- Prunning: if the prunning method is going to be applied, which means that there is going to be a visited list and a key value to consider in the search for each node

Now lets see how we implemented the necessary mechanisms so that the search performs its execution following the desired strategy. To understand this explication we need to understand this piece of code:

```
switch = {0: self.d , 1: -(self.d), 2: -(self.d), 3: -(self.d), 4:
    self.pathCost}
```

So what does this mean? Here we can observe how we give a value to the attribute F of the node, which is the key for the fringe and the visited list, so playing with this values will make the algorithm to expand first one node or another. In this case, 0 is referred to BFS and the following ones in order are: DFS, DLS, IDS and UCS.
So we can see that those strategies that are Depth-Oriented the f is minus the depth, so that in the fringe the rightmost nodes of the tree will be the last ones to be expanded.

In this section of code we can observe how we deal the problem of the iterative deepening search. Knowing that this strategy requires to repeat the execution as many times as necessary or until the limit is reached, the whole search is wrapped with an auxiliary method.

```
actual_depth = increment
solution = None
while (not solution) and (actual_depth <= max_depth):
    solution = bounded_search(problem, strategy, actual_depth)
    depth = actual_depth + increment
return solution
```

So now lets finally see the whole Search method:

```
def bounded_search(problem, strategy, max_depth, pruning):
    frontier = Frontier()
    init_node = TreeNode(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:
                node = TreeNode(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 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
```

We can see the previously defined inputs of the search. The method consists of an uninterrupted while loop, unless there is a solution or all the combinations have been checked, apart from that controller, the main behaviour consists in the addition of the successors of the actual state to the fringe, which as we have said, will automatically rearrange the elements while inserting so that the next node selected will always be the one desired for the strategy.

3.3 Input data retrieval

In this section we are going to explain how we obtain the data we need for the problem, what means, the variables that have been described in the section 3.2.

```
while 1:
    print("\nSelect now the searching strategy:")
    for element in switch.keys():
        print(element,' - ',switch.get(element)[0],
             '(',switch.get(element)[1],')')
    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")
```

In the above code we can observe how the selection of the strategy is taken, we also check spelling errors or input mismatch errors, apart we've got a similar while to select whether the execution has to be prunned or not.

Apart from that we have the particular cases that depend on the search strategy selected, such as the depth increment for the Iterative-deepening-search and so on.

```
if strategy == 2 or strategy in switch.get(2):
    max_depth = int(input("Choose the maximum depth: "))
else:
    max_depth = 999

if strategy == 3 or strategy in switch.get(3):
    increment = input("Specify the increment: ")
else: increment = 1
```

In this section of the code we can see how we handle the specific cases where the strategy implies specific information.

3.4 Change in the data structure

In the last delivery we explained the selection of the data structure to implement the fringe, or the visited list. We get to the conclusion that the best performance was the bisect package.

In this task, we had to implement the real fringe, and we get to a problem in which the indexation of the elements was not the desired one, so we had to change it for the next one that gave us the best results, the SortedContainers and specifically the data structure called SortedKeyList.
SortedKeyList provides a data structure with auto indexing, in which with this sentence:

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

We can select that the attribute we need to index the list is the F, so every node inserted will be in an ascending order, so the leftmost ones will be the ones with the lower f value.

We can also consider the changes that it represents in the insertion and in the elimination of items from the list, lets see how they changed:

```
def insert(self, node):
    if isinstance(node, TreeNode):
        self.frontier.add(node)
    else:
        print("Not a valid node.")

def remove(self):
    return self.frontier.pop(0)
```

Here we can see how we eliminate, always, the first node of the fringe, so that the selected strategy will be followed due to its sorting.

Now lets see how we made the insertion in the last delivery:

```
def insert(self, node):
    if isinstance(node, TreeNode):
        bisect.insort(self.frontier, (node.f, node))
    else:
        print("Not a valid node.")
```

As we can see, to the bisection data structure we needed to pass the argument that we needed the list to be sorted by, in our case the F, so we can see that with the SortedKeyList, once the key is defined, the structure automatically sorts itself.