CUBE IMPLEMENTATION DOCUMENTATION

Practical Work
Intelligent Systems

Team Members: Repository name:	A1-05
• Bernalte Sánchez, Raúl	raulbs7
• Rodríguez-Barbero González, Aarón	Aharon1377
• Rodríguez Villafranca, Enrique	ERV501

Index

1. Language chosen

We have decided to use Python as the programming language for our project due to its versatility and simplicity, just as the used libraries have proven to be

2. Structure of the Cube

The structure of the cube consists of an MD5 identifier (as required), whose structure is a String, and the representation of the cube's faces which has been implemented as a dictionary of lists, each list containing a face of the cube.

The reasoning why we implemented these specific structures was in hopes of the higher efficiency and comprehension level as well as the lowest memory waste.

3. Code

In this section, we are going to explain the main functionalities of the most important parts of the code:

3.1. Class *Cube*

```
def __init__(self, id="", faces=None):  # constructor
    self.id = id
    self.faces = faces
    self.dict_colours = {0: 'red', 1: 'blue', 2: 'yellow', 3: 'green', 4: 'orange', 5: 'white'}
    self.dict_faces = {0: 'BACK', 1: 'DOWN', 2: 'FRONT', 3: 'LEFT', 4: 'RIGHT', 5: 'UP'}
```

Our Cube is defined by:

- id: it's the md5 code that contains the current cube's faces codified.
- faces: position of each face in the following format (see image below).
- dict_colours: python dictionary used in the graphical representation of the cube, it contains the color-number relation.
- dict_faces: python dictionary used in the movement of the faces.

```
{
 "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]]
}
```

3.1.1. Method *generateMoves*

Here we compute the valid methods for the cube, depending on its dimensions (NxNxN).

```
def generateMoves(self):  # this method computes the valid moves for the cube

    n = len(self.faces["UP"])  # here we get the dimension of the cube NxNxN

    ret = ()
    aux = ""

    for i in range(n):
        aux = "L" + str(i)
        ret += (aux,)
        aux = "l" + str(i)
        ret += (aux,)
        aux = "D" + str(i)
        ret += (aux,)
        aux = "d" + str(i)
        ret += (aux,)
        aux = "B" + str(i)
        ret += (aux,)
        aux = "b" + str(i)
        ret += (aux,)

    return ret;
```

3.1.2. Method *move*

Performs the select move:

```
length = len(self.faces["UP"])  # here we get the dimension of the cube NxNxN
layer = int(movement[1])  # here we get the layer where the move is performed
mov = movement[0]  # here we get the type of movement. Example L0 layer is 0, mov = L
inv = length - 1 - layer  # here we get the inverse layer of the selected layer (useful for some movements)
aux = [0 for a in range(length)]  # we create an auxiliaty array to store a row/column of a face

if (mov == "L"):
    self.moveL(layer, inv, aux, length)
if (mov == 'l'):
    for i in range(3):
        self.moveL(layer, inv, aux, length)
if (mov == "D"):
    self.moveD(layer, inv, aux, length)
if (mov == 'd'):
    for i in range(3):
        self.moveD(layer, inv, aux, length)
if (mov == "B"):
    self.moveB(layer, inv, aux, length)
if (mov == 'b'):
    for i in range(3):
        self.moveB(layer, inv, aux, length)
self.cubeMD5()
filename = "" + self.id + "-" + movement + ".json"
self.cube2Json("saved/"+filename)
```

3.1.3. Method *printState*

Draws and prints the current state of the cube (using Turtle library) taking into account the positions and the colors we specified in the dictionary “dict_colours”.

3.1.4. Method *turnRight*

Rotates a face of the cube 90º to the right.

```
def turnRight(self, index): #This is an internal method that allows us to rotate a face of the cube 90 degrees to the right

    matrix = self.faces[self.dict_faces[index]]

    res = [[0 for i in range(len(matrix))] for j in range(len(matrix[0]))]
    aux = [0 for i in range(len(matrix))]

    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            aux[j] = matrix[i][j]

        for j in range(len(matrix)):
            res[j][len(matrix) - 1 - i] = aux[j]

    for i in range(len(matrix)):
        for j in range(len(matrix)):
            self.faces[self.dict_faces[index]][i][j] = res[i][j]
```

3.1.5. Method *cubeString*
Creates the string with the current cube state.

3.1.6. Method *cubeMD5*
Codifies the previously created String into md5.

3.1.7. Method *json2cube*
Opens the json file we are provided and transforms it into an object cube.

3.1.8. Method *cube2Json*
Writes into a new json file the state of the cube.

3.2. Class *Frontier*
Defines our frontier. We decide to create a class, in which we have an array representing the frontier. The principal porpose to have a class with only this attribute, is to have two methods to manage this special array.

It contains the insertNode method, used to include nodes with an increasing order of the f value in the frontier, and the removeNode method in order to pop the node in the first position.

3.3. Class *Node*
Definition of the cube's node.

```
def __init__(self, id=0, state=None, cost=0, action="", d=0, f=0, parent=None):
    self.state = state
    self.parent = parent
    self.cost = cost
    self.action = action
    self.d = d
    self.f = f
```

3.4. Class Sucessors

Generates the sucessors of the cube state.

```
ret = [ Cube(None,None) for i in range(12)]

state.cube2Json("state.json") # create a new json file with the a copy of the cube's state

for i in range(12):
    ret[i].json2cube("state.json") # reads the previously created json file with the last state

moves = iter(state.generateMoves())

for i in range(12):

    movement = next(moves)

    ret[i].move(movement)
    print(movement,ret[i],1)

return ret
```

3.5. Class search_algorithm

Contains the search algorithms that will be used for finding the cube's solution

3.5.1. Method createListNodes

Create the list of corresponding nodes from the list of successors for the current node depending of the strategy selected

```
def createListNodes(ls, current_node, max_depth, strategy):
    cost_current = current_node.cost
    d_current = current_node.d
    f_current = current_node.f
    ln = [Node() for i in range(len(ls))]

    if d_current == max_depth:
        print(0)
        return None

    for i in range(len(ls)):
        ln[i].action = ls[i][0]
        ln[i].state = ls[i][1]
        ln[i].cost = ls[i][2] + cost_current
        ln[i].parent = current_node
        ln[i].d = d_current + 1
        if strategy == 'DFS' or strategy == 'LDS' or strategy == 'IDS':
            ln[i].f = 1 / (ln[i].d + 1)

        elif strategy == 'BFS':
            ln[i].f = ln[i].d

        elif strategy == 'UCS':
            ln[i].f = ln[i].cost
        else:
            print("ERROR: Not a valid type of algorithm")
            exit

    return ln
```

3.5.2. Method createSolution

Generates a list with the solution nodes, starting in the current node (final node) and going back to its parents until its reachs the starting node

```
def createSolution(current_node):
    sol = []
    node = current_node
    while node is not None:
        sol.append(node)
        node = node.parent
    return sol
```

3.5.3. Method nodeVisited

Looks if the node has been visited or not. If it has been visited, we won't introduce it on the frontier.

```
def nodeVisited(node, closed, optimization):
    cube1 = Cube()
    cube2 = Cube()
    cube1.faces = node.state.faces
    cube1.cubeMD5()
    for n in closed:
        cube2.faces = n.state.faces
        cube2.cubeMD5()
        if cube1.id == cube2.id:
            if optimization:
                if node.f < n.f:
                    return False
                else:
                    return True
            else:
                return True
    return False
```

3.6. Class problem

We define our problem, creating the Initial and Goal states

Method createInitialCube

Creates the Initial state of the cube according to our starting JSON file

```
def createInitialCube(self, Initial_state):
    c = Cube()
    c.json2cube(Initial_state)
    self.Initial_state = c
```

3.6.1. Method createGoal

Creates the goal state of the cube (solved Rubik's cube)

```
def createGoal(self):

    goal_cube = Cube()
    length=0
    string = ""

    for state in self.Initial_state.faces.values():
        for i in state:
            for j in i:
                length+=1
        break

    color2position = {0: 3, 1: 1, 2: 2, 3: 4, 4: 5, 5: 0}

    for i in range(6):
        for j in range(int(length)):
            string+=str(color2position[i])
    goal_cube.cubeMD5(string)

    self.goal = goal_cube.id
```

3.6.2. Method *isGoal*

We check if the current state is the goal state

```
def isGoal(self,state):

    if state is not None:
        if state.id == self.goal:
            return True
    return False
```

3.7. Class statespace

Definition of our state space

```
def successorsCube(state):

    moves = iter(state.generateMoves())

    ret = [ Cube() for i in range(len(state.generateMoves()))]

    for i in range(len(state.generateMoves())):

        ret[i] = copy.deepcopy(state)
        movement = next(moves)
        ret[i].move(movement)
        ret[i] = (movement,ret[i],1)

    return ret
```

3.8. Class *main*

Main class in order to execute the code, ask the user the necessary parameters and prints the cube's movement sequence in order to reach the solution.

```
from problem import Problem
from search_algorithm import limited_search, search

filename_output = "solution.txt"

def writeFile(filename, text):
    with open(filename, 'a') as outfile:
        outfile.write(text+"\n")

print('--------- Introduce the type of algortihm: ---------')
algorithm = str(input())
print('--------- Introduce the json filename of the Initial_state: ---------')
filename_input = str(input())
print('--------- Introduce the maximum depth of the nodes of the problem: ---------')
depth = int(input())
if algorithm == "IDS":
    print('--------- Introduce the incremental constant of the depth (if it is necessary for IDS
    inc_depth = int(input())
else:
    inc_depth = 0
Prob = Problem(filename_input) #Clase Problem with InitialState and functions isGoal and sucesso
sol = search(Prob, algorithm, depth, inc_depth)
if sol is not None:
    print('############### The SOLUTION of the cube with movements is the next one: ############
    for i in range(len(sol)-1,0,-1):
        print(sol[i-1].action)
        writeFile(filename_output, sol[i-1].action)
        writeFile(filename_output, str(sol[i-1].state.faces))
else:
    print('############### No solution was found ################')
```

4. Strategies execution

Now, we are going to test every strategy in the same cube and show the corresponding results using BFS,DFS,LDS,IDS and UCS

Cube:

```
[
{
"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]]
}
]
```

maximum Depth = 3

4.1. BFS

```
========== Number of created nodes:  3636  =============================
========== Memoria utilizada:  0.1109619140625  MB ============================
========== Execution time of the search:  1.6670212745666504  seconds ===================
############### The SOLUTION of the cube with movements is the next one: ##################
b1
d2
```

4.2. DFS

```
========== Number of created nodes:  14436  ==============================
========== Memoria utilizada:  0.4405517578125  MB ============================
========== Execution time of the search:  19.154411554336548  seconds ==================
############### No solution was found #################
```

4.3. LDS

```
========== Number of created nodes:  14436  ==============================
========== Memoria utilizada:  0.4405517578125  MB ============================
========== Execution time of the search:  18.7975115776062  seconds ==================
############### No solution was found #################
```

4.4. IDS

Incremental Depth constant = 2

```
========== Number of created nodes:  3528  ============================
========== Memoria utilizada:  0.107666015625  MB ==========================
========== Execution time of the search:  1.7769582271575928  seconds =================
############### The SOLUTION of the cube with movements is the next one: #################
b1
d2
```

4.5. UCS

```
========== Number of created nodes:  3636  ============================
========== Memoria utilizada:  0.1109619140625  MB ===========================
========== Execution time of the search:  1.6760327816009521  seconds =================
############### The SOLUTION of the cube with movements is the next one: #################
b1
d2
```