CUBE IMPLEMENTATION DOCUMENTATION

Practical Work Intelligent Systems

**Team Members: Repository name:** A1-05

* Bernalte Sánchez, Raúl raulbs7
* Rodríguez-Barbero González, Aharon1377

Aarón

* Rodríguez Villafranca, Enrique ERV501

Index

[1. Language chosen 3](#_Toc24238980)

[2. Structure of the Cube 3](#_Toc24238981)

[3. Code 3](#_Toc24238982)

[3.1. Class *Cube* 3](#_Toc24238983)

[3.1.1. Method *generateMoves* 4](#_Toc24238984)

[3.1.2. Method *move* 5](#_Toc24238985)

[3.1.3. Method *printState* 5](#_Toc24238986)

[3.1.4. Method *turnRight* 5](#_Toc24238987)

[3.1.5. Method *cubeString* 6](#_Toc24238988)

[3.1.6. Method *cubeMD5* 6](#_Toc24238989)

[3.1.7. Method *json2cube* 6](#_Toc24238990)

[3.1.8. Method *cube2Json* 6](#_Toc24238991)

[3.2. Class *Frontier* 6](#_Toc24238992)

[3.3. Class *Node* 6](#_Toc24238993)

[*3.4.* Class *Sucessors* 7](#_Toc24238994)

[*3.5.* Class *search\_algorithm* 7](#_Toc24238995)

[*3.5.1.* Method *createListNodes* 7](#_Toc24239004)

[*3.5.2.* Method *createSolution* 8](#_Toc24239014)

# 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

# Structure of the Cube

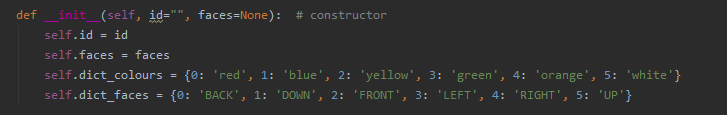
The structure of the cube consists of an MD5 identifier (as required), whose structure is a String, and the representation of the cube’s faces which has been implemented as a dictionary of lists, each list containing a face of the cube.

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.

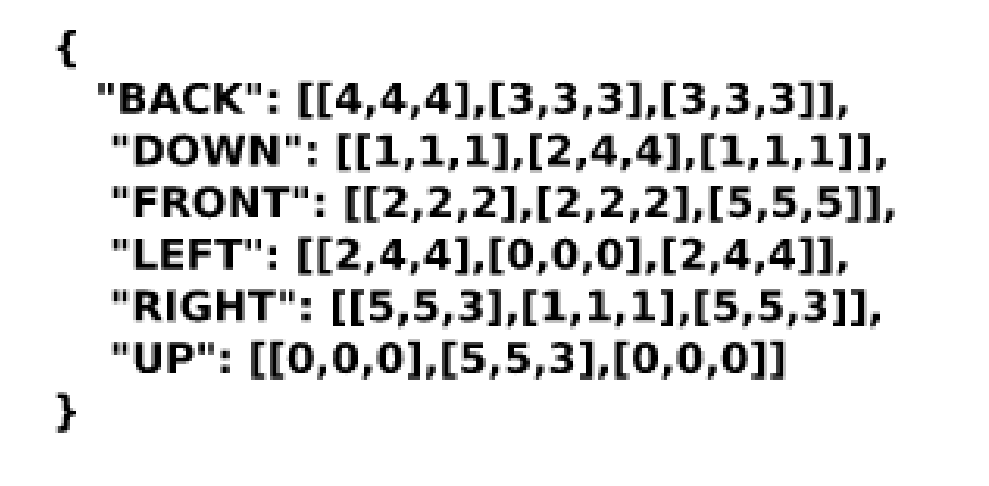
# Code

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

## Class *Cube*

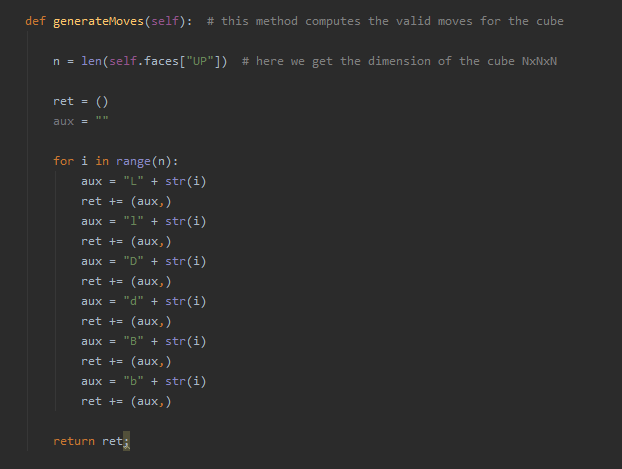


Our Cube is defined by:

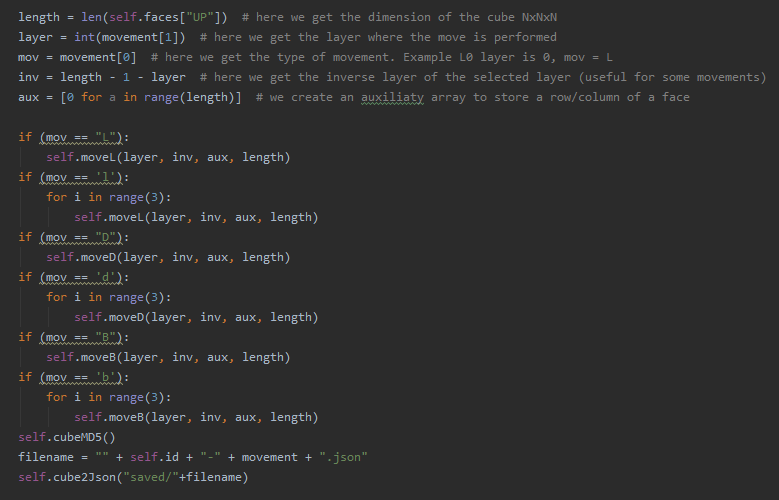
* id: it’s the md5 code that contains the current cube’s faces codified.
* faces: position of each face in the following format (see image below).
* dict\_colours: python dictionary used in the graphical representation of the cube, it contains the color-number relation.
* dict\_faces: python dictionary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

### Method *generateMoves*

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



### Method *move*

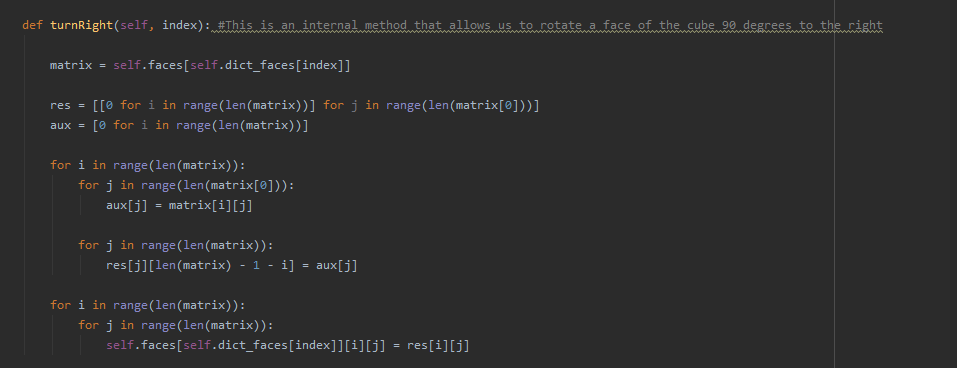
Performs the select move: 

### 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”.

### Method *turnRight*

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



### Method *cubeString*

Creates the string with the current cube state.

### Method *cubeMD5*

Codifies the previously created String into md5.

### Method *json2cube*

Opens the json file we are provided and transforms it into an object cube.

### Method *cube2Json*

Writes into a new json file the state of the cube.

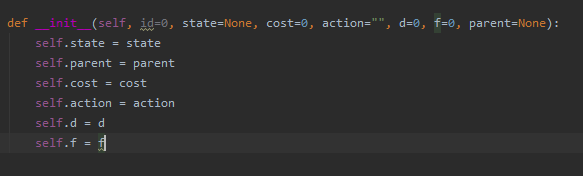
## Class *Frontier*

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.

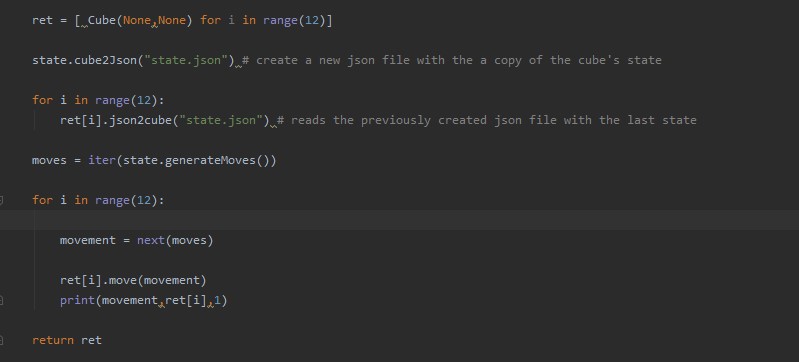
## Class *Node*

Definition of the cube’s node.



## Class *Sucessors*

Generates the sucessors of the cube state.



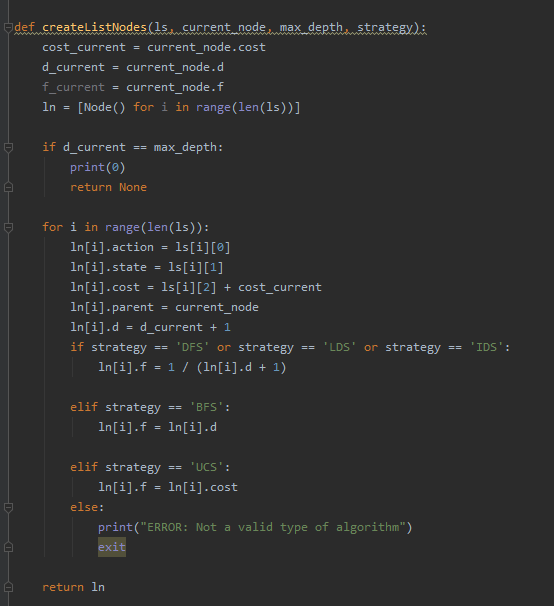
## Class *search\_algorithm*

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



## Method *createListNodes*

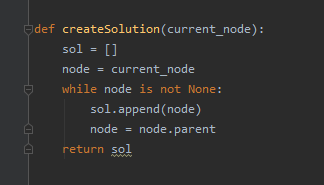
Create the list of corresponding nodes from the list of successors for the current node depending of the strategy selected





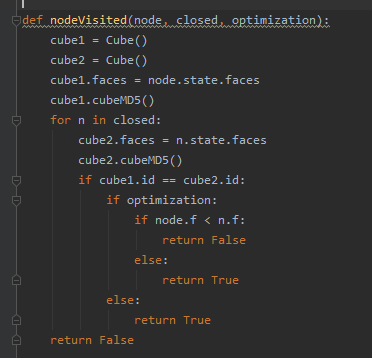
## Method *createSolution*

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



* + 1. Method *nodeVisited*

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

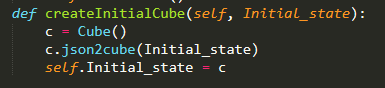


* 1. Class *problem*

We define our problem, creating the Initial and Goal states

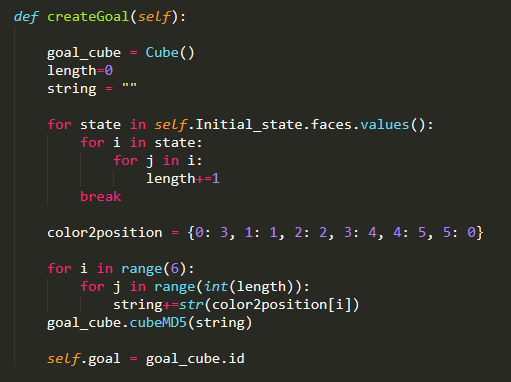
* + 1. Method *createInitialCube*

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



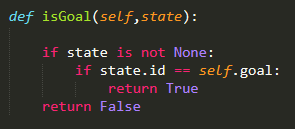
* + 1. Method *createGoal*

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



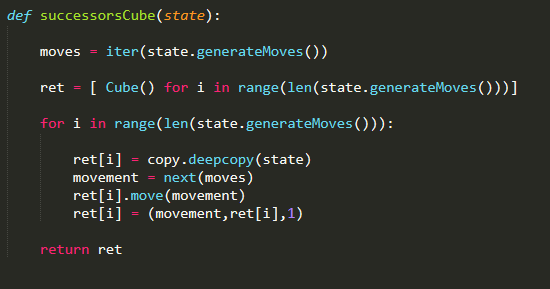
* + 1. Method *isGoal*

We check if the current state is the goal state



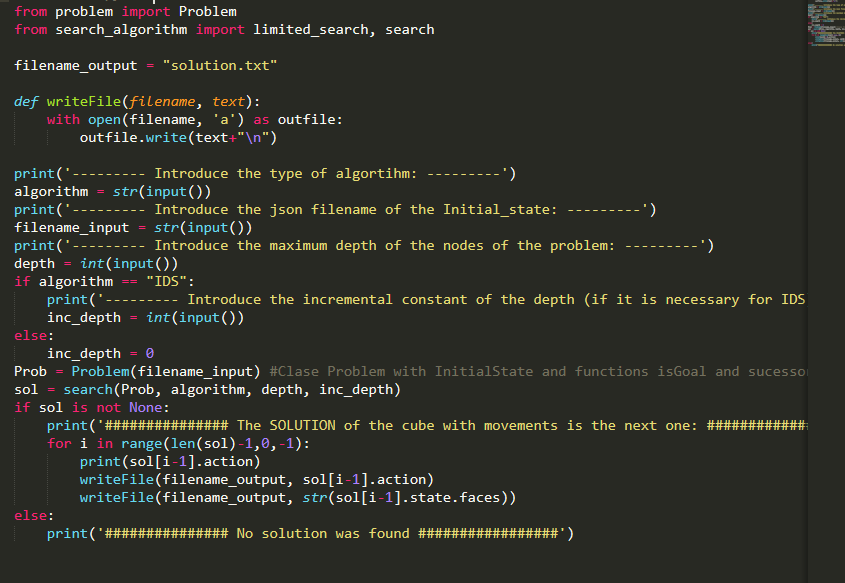
* 1. Class *statespace*

Definition of our state space



* 1. 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.



1. 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]],

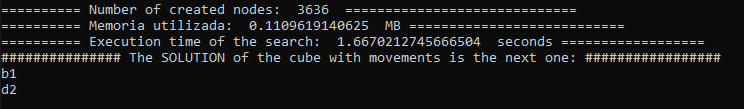
"UP":[[0,0,0],[5,5,3],[0,0,0]]

}

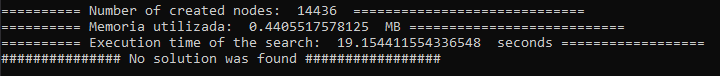
]

maximum Depth = 3

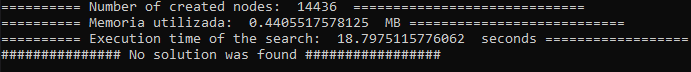
* 1. BFS



* 1. DFS

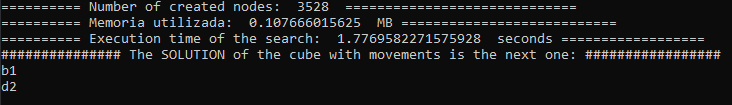


* 1. LDS



* 1. IDS

Incremental Depth constant = 2



* 1. UCS

