

Rubik's cube

Final report on project work for course in Mathematics with Computer Science

Matevž Nolimal

5/12/2022

1.) Introduction and Project Description

Rubik's cube is the project that extends mathematics to Computer Science. As the name suggests its main goal is about the game of Rubik's cube - one of the most popular toys of the 20th century. This made the game well known to the public but not everyone is capable of solving its puzzle. A normal 3 x 3 cube entangles an enormous number of possible permutations - to be precise 43,252,003,274,489,856,000 (i.e. roughly 43 quintillion).

The goal of the project has been not just to write a program to rotate, slice and shuffle the cube, but also to solve the puzzle (if possible in an efficient way). While writing the code I restricted myself to follow the so-called clean code principles and tried not to write a single line of production code until without its advance written failing test. The so-called test-driven-development i.e. TDD enforces the rules of the game to be met in even its smallest parts. The code is written in Python as I find this programming language the closest to me and nevertheless I gained plentiful of experience working with it.

2.) Motivation for Testable Code

Like outlined above I did my best to follow the Test Driven Development (in Computer Science more known under the abbreviation TDD). TDD stands for coding while the programmers restrict themselves to the fact that not a single line of production code can be written without the failing test written upfront. In that manner the programmer makes sure it has a fully tested code that matches expectation/its wanted behaviour. I learned about the clean coding principles from the book of Robert C. Martin [1] Clean Code: A Handbook of Agile Software Craftsmanship.

Coding in Python and using its robust code structures and abstractions has multiple advantages over the non-structured way of coding.

3.) Start of the Project

Due to the fact that I never truly tackled the problem prior to this course, I first had to find/study the material on Rubik's cube solvers. Following many online forums I stumbled upon the Beginner Rubiks Soltuion reference paper [2] which gave me a good understanding of the problem I shall try to solve. I was lucky enough that it also explains in detail a possible way to solve the cube. What I found admirable about the paper is that it was written in a "programming-way" which facilitated for me the implementation of Rubik's cube game.

4.) Implementation of the game

Following the above mentioned paper's proposal of the terminology I envisioned writing several Python classes, whereas each one of them defines one layer of information. The only way to come up with the

framework of such game in my opinion is to write bottom up and think of the smallest particle that holds any relevant information.

The most intuitive implementation of such classes that define objects with data and its functionalities in Python are the so called (data)classes. They are constructed by adding the dataclass decorator to the somewhat improved normal class structure. Having some good experiences with such class structures I often emphasize, this is the best way one can build the foundations of programs using Python.

4.1) Move Notations

Following the proposal of the paper and WCA Regulations and Guidelines one should define move notation. clockwise face rotations:

`F - front, R - right, U - upper,`

`L - left, B - back, D - down`

Clockwise slice rotations:

`M - middle, E - equatorial, S - standing,`

`X - whole cube around x axis, Y - whole cube around y axis, Z - whole cube around z axis`

All counterclockwise rotations are defined with `i` added at the end of the character.

4.2) Piece

In the cube everything starts with a piece. The 3 x 3 x 3 cube consists of 27 such pieces, whereas each one has its position and colour. This implied coming up with a structure/class holding that information. The most natural name for it shall always be the most obvious i.e. the “Piece”. There are multiple types of pieces - 8 corners, 6 centres and 12 edges and a hidden one. Not all pices can be rotated, these are obviously differently coloured centres. This implies a method to rotate them is needed.

Each of the 27 Rubik’s pieces can be positioned in the three dimensional Cartesian coordinate system. Like that each `Piece` can have its own `position` tuple (`x`, `y`, `z`) where component is `{-1, 0, 1}`. Obviously `{0, 0, 0}` is the center of the cube.

Let: - `x` axis point in `R` direction - `y` axis point in `U` direction - `z` axis point in `F` direction

Each `Piece` has as well `colors` Tuple (`cx`, `cy`, `cz`) with the color of the sticker along each axis. Obviously only the corner is going to be without any `None` values.

Example:

`colors = ("Orange", None, "Red")` is an edge with orange facing x-direction (`R`) and red facing z-direction (`F`).

We add a class method `rotate` to the class which does a 90-degree rotation of a piece. Following the rotation matrix [3] one can apply a matrix vector multiplication to update position tuple. Then we update the colors tuple, by swapping exactly two entries in its colors tuple.

4.3) Point

Next level that is required is the “Point” which is a 3-d vector defining the position of the before mentioned pieces in the cube. Point dataclass is used to define face rotations by the above mentioned `position` tuples (`x`, `y`, `z`) where components take values of `{-1, 0, 1}`. With it one can define the any positions along `X`, `Y`, `Z` axis respectively. This enables me adding the `R`, `L`, `U`, `D`, `F`, `B` face rotations to the constants.

4.4) Matrix

Following the above mentioned paper I figured the main functionality of the program will be backend that serves/enables rotation of “Points” in different ways along the three axis (x - pointing to the right, y - direction up, z - direction to the front) of the Rubik’s cube. I quickly realised some sort of rotation matrix multiplication will have to be implemented in the code. I followed the reference [3] which gave me sufficient information on the subject needed for this task.

Such level of information is embeded in the dataclass called “Matrix” which is a list representation of rotation matrices. This enabled me defining clockwise (ROT_XY_CW, ROT_XZ_CW, ROT_YZ_CW) and counter-clockwise (ROT_XY_CC, ROT_XZ_CC, ROT_YZ_CC) plane rotations in the constants.

4.5) Cube

Last but not least comes a class that connects pieces with face and plane rotations called “Cube”. It accepts a string defining a cube and recognises its pieces to segregate them among faces, edges and corners. The total length of such string can only be 54, whereas the cube is interpreted like that:

UUU	0	1	2
UUU	3	4	5
UUU	6	7	8
LLL FFF RRR BBB	9	10	11 12 13 14 15 16 17 18 19 20
LLL FFF RRR BBB	21	22	23 24 25 26 27 28 29 30 31 32
LLL FFF RRR BBB	33	34	35 36 37 38 39 40 41 42 43 44
DDD	45	46	47
DDD	48	49	50
DDD	51	52	53

On the left hand side we see the string representation of the cube, while on the right hand side its numerical sequence of the pieces. Since the cube string called `cube_str` is expected to recognise all Rubik’s cube pieces no matter the scramble, we add a method to check if the string actually represents a valid Rubik’s cube. In addition it has a method that recognises already solved cubes which is called `is_solved`.

On the other hand Cube rotation methods manage the rotation of its faces, slices and multiple pieces with previously defined face and plane rotations.

5.) Solving method

I chose the layer based solving method, which I learned about in the mentioned reference paper [2] which was referenced as well by other similiar projects of Rubiks cube solvers. It is based on the principle of cube being solved in three layers - with some further minor steps in them.

5.1) Cross

Starting with the first layer one must first form the cross where corners have to be colored correctly to match the colors of the neighbouring faces. This is done by first recognising the initial pieces of the cross: `f1`, `fr`, `fu`, `fd`. Afterwards we try to turn them right or left to get the piece in the correct place in `z = -1`. In other words the main goal of this task is to bring the white edges from bottom to the top.

Example: Any bottom pieces can be either faced down or front. If it is faced down one must simply rotate twice, if front then do the following `Fi Ri Di R` - and then from bottom to top by `F F`. If there no white edge in the bottom we can always bring it there. If it is in the middle layer we must first bring it down by doing `Ri Di R` - and then from bottom to top by `F F`.

5.2) Cross corners

Like in the previous step we first need to locate some pieces **fld**, **flu**, **frd**, **fru**. To solve the corners one can use the move **Ri Di R D** that many times that you replace **fru** with **frd**. Similarly can be done with **flu** with **fld**.

5.3) Second Layer

First locate some pieces **rd**, **ru**, **ld**, **lu**. Moving the cube upside down means **Z** rotation. To solve the second layer we will use two algorithms where one will send a top piece to the left and the other that places it to the right, depending on the color combination of the top “edge” piece. The left algorithm **Ui Li U L U F Ui Fi** and right, **U R Ui Ri Ui Fi U F**. If the middle piece is in the wrong location i.e. not matching with colors one can use the same right algorithm to move it to the top and then back with the right one in place. Be aware that While solving the second layer the first layer is still intact.

5.4) Back face

Prior starting to work on this, just flip the cube to the unsolved part.

We will first need to check the shape of the back face edges. There are 4 possible states that we are interested into (dot -> L -> line -> cross). Our goal is to reach the cross in the back face. There is a sequence/algorithm of moves that transforms from one pattern to the next - **F R U Ri Ui Fi**. If starting with a dot always make sure that the L shape is pointed correctly before doing the moves (one axis to the back and the other to the left).

After the cross we will organize the yellow edges to match the side stickers. First try align some of the edges and on the other two use **R U Ri U R U U Ri U**. If they are on the opposite sites, repeat the algorithm.

5.5) Last layer corners position & orientation

First locate the corners **c1**, **c2**, **c3**, **c4** of the upper face where we enumerate them in the following way:

```
U face:
# 4-3
# ---
# 2-1
```

We need to put them in the correct position and then rotate them in the next phase. We try to find a piece for which its three colors hold - orient the cube with this piece in the **fru** position and do the moves **U R Ui Li U Ri Ui L**. Repeat this same procedure if the pieces did not come to its right destination in the first iteration.

Once done remember the algorithm to swap the top and down pieces of the cube from step 5.2.) Cross corners. We will use this same algorithm to orient the last layer corners in this final step.

Try to identify the corner where top color appears on the right face. Make sure it is point toward you, like before in its **fru** position and repeat the following algorithm **Ri Di R D** until the wanted sticker faces up. Then rotate the top face to bring another misaligned corner to the front right position. Do the moves until the wanted sticker faces up. In best case scenario you will just have to turn the up face to finish the cube.

What if all corners are wrongly orientated? Do the moves until it matches, bring another misaligned corner piece, do the moves, bring another misaligned corner piece and do the moves again, and do the moves for one last time. Once they are correctly orientated just finish the cube by turning its upper face.

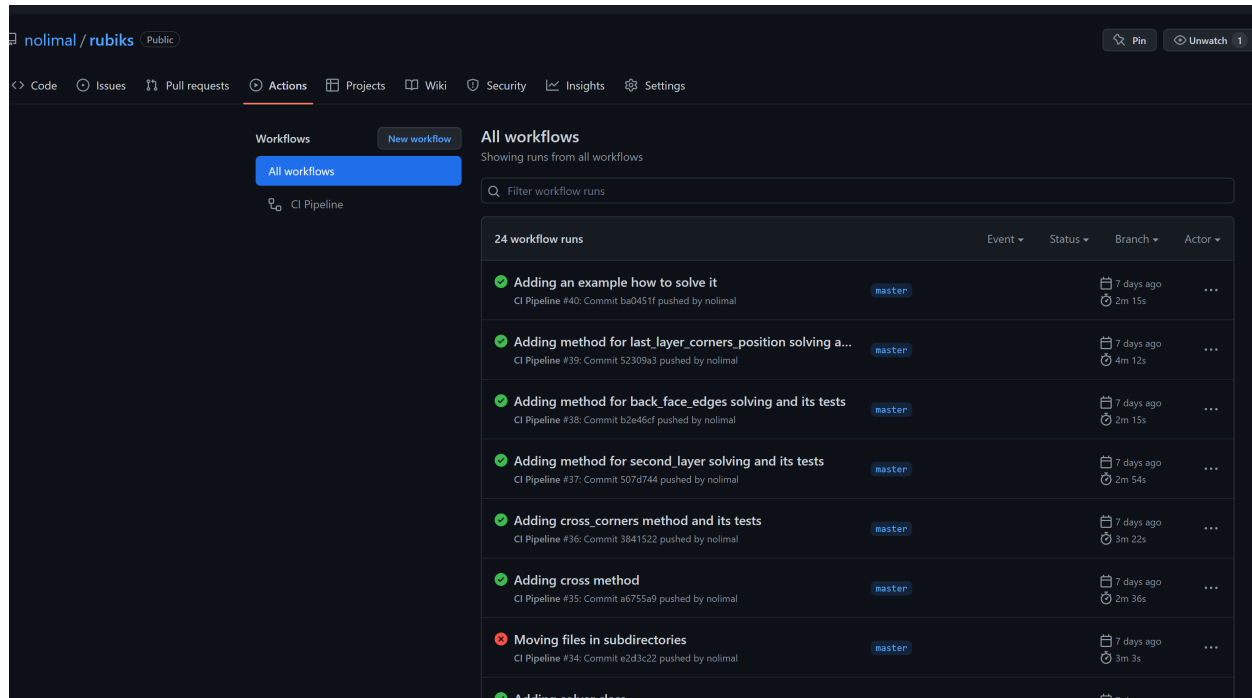


Figure 1: Passing pipeline

6.) CI Pipeline

In order to be sure that the code with all its tests is working I added a YAML file defining CI pipeline to my repository (can be seen in `.github/workflows`). Like that we make sure that nothing gets merged to the repository by any developer working on a project without all passing tests. One can always control for that on the following tab in github: <https://github.com/nolimal/rubiks/actions>

In addition to the above it also makes sure that all developers or “forkers” of the project, work on the same packages as CI pipeline always syncs its packages based on the `requirements.txt` file which I added to the repository as well. Last but not least it also makes sure to reformat the code and python imports with the help of package `black`. Nevertheless it also enforces formatting rules to be met with `flake8`. Due to same discrepancies between the two I also had to add the config files for both of them. Details of how I defined the CI pipeline can be seen on the bottom picture or directly in github (<https://github.com/nolimal/rubiks/blob/master/.github/workflows/main.yml>).

7.) Self-Reflection on the Testing

Tests allow me to be fairly sure in the code produced by this project, from the bellow snippet this can be confirmed as I ran `pytest` on the whole directory to confirm that all tests are passing.

40 lines (37 sloc) | 1.08 KB

```
1  name: CI Pipeline
2
3  on:
4    push:
5      branches: [ master ]
6    pull_request:
7      branches: [ master ]
8
9
10 jobs:
11   ci-pipeline:
12     runs-on: windows-latest
13     steps:
14       - name: Checkout
15         uses: actions/checkout@v2
16       - name: Get changed files
17         uses: dorny/paths-filter@v2
18         id: filter
19         with:
20           list-files: shell
21           filters: |
22             py_modified:
23               - added|modified: "**/*.py"
24       - name: Setup Python
25         if: ${ steps.filter.outputs.py_modified == 'true' }
26         uses: actions/setup-python@v2
27         with:
28           python-version: 3.10.0
29           architecture: x64
30           cache: 'pip'
31       - name: Install dependencies
32         run: pip install -r requirements/requirements.txt
33       - name: Run black
34         if: ${ steps.filter.outputs.py_modified == 'true' }
35         run: black ${ steps.filter.outputs.py_modified_files }
36       - name: Run flake8
37         if: ${ steps.filter.outputs.py_modified == 'true' }
38         run: flake8 ${ steps.filter.outputs.py_modified_files }
39       - name: Run pytest
40         run: pytest
```

Figure 2: main.yaml

```

86     "cube1, original_scramble",
87     [
88         ("solved_cube", "D1 R L F R1 L U U F D D R1 L L F F D R R B B D D L L U1 B B U1"),
89         ("solved_cube", "U1 F B R1 U R1 F1 D D L R U1 F F L L U1 D L L U1 B B U1"),
90         ("solved_cube", "F U1 F F D1 B L L F D D B U U B B U U L B R F F U1 L R R"),
91         ("solved_cube", "R R F F L L F1 U U B1 D D B B L U F F L1 R1 U R1 U R1 U L"),
92         ("solved_cube", "R R F F L L F1 U U B1 D D B B L U F F L1 R1 U R1 U R1 U L"),
93         ("solved_cube", "F F D1 B B D1 B B R R U R1 D1 R R D B B R R B B F1 D1 F1 L1"),
94         ("solved_cube", "R1 U B U U R R F1 R1 F1 U1 L L U F F U1 D1 F1 L L D D")
95     ]
96 )
97
98 def test_integration_test_solve_parametrized_cube(cube1, original_scramble, request):
99     cube = request.getfixturevalue(cube1)
100     assert cube.is_solved()
101
102     cube.sequence(original_scramble)
103     assert not cube.is_solved()
104     solver = Solver(cube)
105     solver.solve()
106     assert solver.cube.is_solved()

```

```

593     return (
594         self.cube[FRONT + DOWN].color
595         and self.cube[FRONT + RIGHT]
596         == self.cube.right_color()
597         and self.cube[FRONT + DOWN].
598         == self.cube.front_color()
599         and self.cube[FRONT + RIGHT]
600         == self.cube.front_color()
601     )
602
603     count = 0
604     while not self.cube.is_solved():
605         for _ in range(4):
606             if fish_pattern():
607                 self.move(fish_move)
608                 if self.cube.is_solved():
609                     return
610             else:
611                 self.move("Z")
612
613     if h_pattern1():

```

```

test session starts
collecting ... collected 7 items

test_solver.py::test_integration_test_solve_parametrized_cube[solved_cube-D1 R L F R1 L U U F D D R1 L L F F D R R B B D D L L U1 B B U1]
test_solver.py::test_integration_test_solve_parametrized_cube[solved_cube-U1 F B R1 U R1 F1 D D L R U1 F F L L U1 D L L U1 B B U1]
test_solver.py::test_integration_test_solve_parametrized_cube[solved_cube-F U1 F F D1 B L L F D D B U U B B U U L B R F F U1 L R R]
test_solver.py::test_integration_test_solve_parametrized_cube[solved_cube-R R F F L L F1 U U B1 D D B B L U F F L1 R1 U R1 U R1 U L]
test_solver.py::test_integration_test_solve_parametrized_cube[solved_cube-R R F F L L F1 U U B1 D D B B L U F F L1 R1 U R1 U R1 U L]
test_solver.py::test_integration_test_solve_parametrized_cube[solved_cube-F D1 B B D1 B B R R U R1 D1 R R D B B R R B B F1 D1 F1 L1]
test_solver.py::test_integration_test_solve_parametrized_cube[solved_cube-R1 U B U U R R F1 R1 F1 U1 L L U F F U1 D1 F1 L L D D]

===== 7 passed in 3.91s =====

```

Figure 3: Integration test

```

PS C:\Users\35269\Documents\fax\WZR\rubiks> pytest .
===== test session starts =====
platform win32 -- Python 3.9.1, pytest-7.1.0, pluggy-1.0.0
rootdir: C:\Users\35269\Documents\fax\WZR\rubiks
collected 118 items

game\tests\test_cube.py ..... [ 50%]
game\tests\test_matrix.py ... [ 63%]
game\tests\test_piece.py .... [ 68%]
game\tests\test_point.py ..... [ 75%]
solving_methods\tests\test_solver.py ..... [ 94%]
solving_methods\tests\test_utilities.py ..... [100%]

===== 118 passed in 4.31s =====
PS C:\Users\35269\Documents\fax\WZR\rubiks>

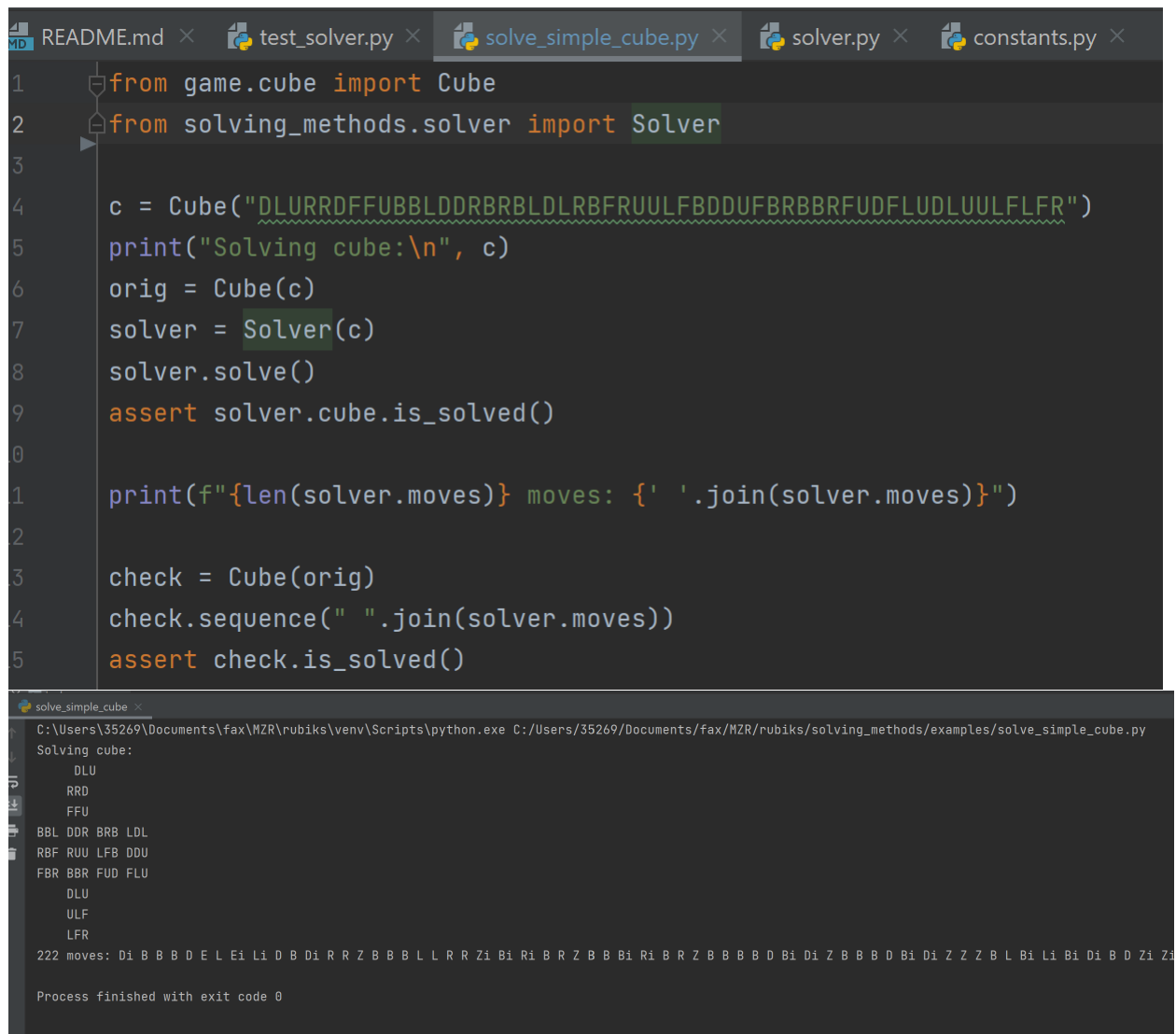
```

Like announced in the project introduction I tried using Test Driven Development throught the project. While I managed to stick to the principles while implementing the game with its rules, I unfortunatly was not able to stick to the same Red - Green - Refactor process in the solver. I therefore did partial integration tests for each of the methods.

Finally, once I finished the “Solver” class I tried finding some real-life examples of properly scrambled Rubiks cubes. I came across the World Cube Association Regulations and guidelines [4] describing and referring to the official “scrambler” [5] that updates the scrambling moves on daily basis.

I took all of them on the 7th of May 2022 and added them to the `test_solver.py` file that consists as well of the integration test, which makes sure that the whole functionality is working as it is supposed to. The test is called `test_integration_test_solve_parametrized_cube`.

8.) Example



```
1 from game.cube import Cube
2 from solving_methods.solver import Solver
3
4 c = Cube("DLURRDFUFUBBLDDRBRBLDLRBFUULFBDDUFBRBBRFUDFLUDLUULFLFR")
5 print("Solving cube:\n", c)
6 orig = Cube(c)
7 solver = Solver(c)
8 solver.solve()
9 assert solver.cube.is_solved()
10
11 print(f"{len(solver.moves)} moves: {' '.join(solver.moves)}")
12
13 check = Cube(orig)
14 check.sequence(" ".join(solver.moves))
15 assert check.is_solved()
```

```
C:\Users\35269\Documents\fax\MZR\rubiks\venv\Scripts\python.exe C:/Users/35269/Documents/fax/MZR/rubiks/solving_methods/examples/solve_simple_cube.py
Solving cube:
  DLU
  RRD
  FFU
BBL DDR BRB LDL
RBF RUU LFB DDU
FBR BBR FUD FLU
  DLU
  ULF
  LFR
222 moves: Di B B B D E L Ei Li D B Di R R Z B B B L L R R Zi Bi Ri B R Z B B Bi Ri B R Z B B B B D Bi Di Z B B B D Bi Di Z Z Z B L Bi Li Bi Di B D Zi Zi
Process finished with exit code 0
```

9.) Possible improvements

There are numerous possible ways to improve the solver or add a new one. One way would be to optimise the existing solver by “eliminating” the redundant moves that are followed by its inverse.

10.) References

- [1] Robert C. Martin: Clean Code: A Handbook of Agile Software Craftsmanship (2008) p. 122-133
- [2] Rubik's Cube Notation <https://ruwix.com/the-rubiks-cube/notation/>
- [3] Beginner Solution to the Rubik's Cube (2005) p. 1-7
- [4] Rotation matrix https://en.wikipedia.org/wiki/Rotation_matrix

[5] World Cube Association - WCA: Regulations <https://www.worldcubeassociation.org/regulations/>

[6] World Cube Association - WCA: Scrambles <https://www.worldcubeassociation.org/regulations/scrambles/>