

---

## RubiksCube-TwophaseSolver

### Overview

This project implements the fully developed form of the two-phase-algorithm to solve the Rubik's Cube in Python. While Python is much slower compared to languages such as C++ or Java, the implementation is efficient enough to solve random cubes in less than 20 moves on average within a few seconds on slow hardware like the Raspberry Pi3.

If your goal is simply to solve Rubik's Cube and explore its patterns, Cube Explorer might be the better choice. However, if you aim to gain a deeper understanding of the two-phase-algorithm's intricacies or if you are working on a project to construct a cube-solving robot that achieves near-optimal solutions, then this is the right resource. ## Usage

The package is available on PyPI and can be installed with

```
$ pip install RubikTwoPhase
```

Once installed, you can import the twophase.solver module into your code:

```
1 >>> import twophase.solver as sv
```

Some tables need to be created, but only on the first run. These tables take up approximately 80 MB of disk space and require about half an hour or more to generate, depending on your hardware. However, it is with these computationally expensive tables that the algorithm operates efficiently, usually finding near-optimal solutions.

A cube is defined by its cube definition string. A solved cube has the string 'UUUUUUUUUR-RRRRRRRRFFFFFFFFDDDDDDDDLLLLLLLLBBBBBBBB'.

```
1 >>> cubestring = '
    DUUBULDBFRBFRRULLLBRDFFFBLURDBFDFDRFULBLUFDURRBLBDUDL '
```

Refer to <https://github.com/hkociemba/RubiksCube-TwophaseSolver/blob/master/enums.py> for the exact format.

```
1 >>> sv.solve(cubestring,19,2)
```

This will solve the cube described by the definition string with a desired maximum length of 19 moves and a timeout of 2 seconds. If the timeout is reached, the shortest solution computed so far is returned, even if it exceeds the desired maximum length.

```
1 'L3 U1 B1 R2 F3 L1 F3 U2 L1 U3 B3 U2 B1 L2 F1 U2 R2 L2 B2 (19f) '
```

Here, U, R, F, D, L and B denote the Up, Right, Front, Down, Left and Back faces of the cube. 1, 2, and 3 denote a 90°, 180° and 270° clockwise rotation of the corresponding face.

---

If you prefer to allocate a constant time  $t$  for each solution and receive only the shortest maneuver found within that time  $t$ , use the following command:

```
1 >>> sv.solve(cubestring,0,t)
```

You can test the performance of the algorithm on your machine with something similar to

```
1 >>> import twophase.performance as pf
2 >>> pf.test(100,0.3)
```

This example generates 100 random cubes, solves each one in 0.3 s and displays a statistic about the solution lengths.

You also have the possibility to solve a cube not to the solved position but to some favorite pattern represented by the goalstring.

```
1 >>> sv.solveto(cubestring,goalstring,20,0.1)
```

will grant e.g. 0.1 s to find a solution with  $\leq 20$  moves.

---

Another feature is to start a local server listening on a port of your choice. It accepts the cube definition string and returns the solution.

```
1 >>> import twophase.server as srv
2 >>> srv.start(8080, 20, 2)
```

Alternatively, start the server in the background:

```
1 >>> import twophase.start_server as ss
2 >>> from threading import Thread
3 >>> bg = Thread(target=ss.start, args=(8080, 20, 2))
4 >>> bg.start()
```

Upon successful initiation, a message like

```
Server socket created
Server now listening...
```

indicates the proper functioning of the server. In this example, the server is listening on port 8080, with a desired maximum length of 20 moves and a timeout of 2 seconds.

You can access the server, which may also run on a remote machine, using various methods:

<http://localhost:8080/DUUBULDBFRBFRRULLLBRDFFFBLURDBFDFDRFULBLUFDURRBLBDUDL>

via a web browser and the server on the same machine on port 8080.

---

`http://myserver.com:8081/DUUBULDBFRBFRRULLLBRDFFFBLURDBFDFDRFRULBLUFDURRBLBDUDL`

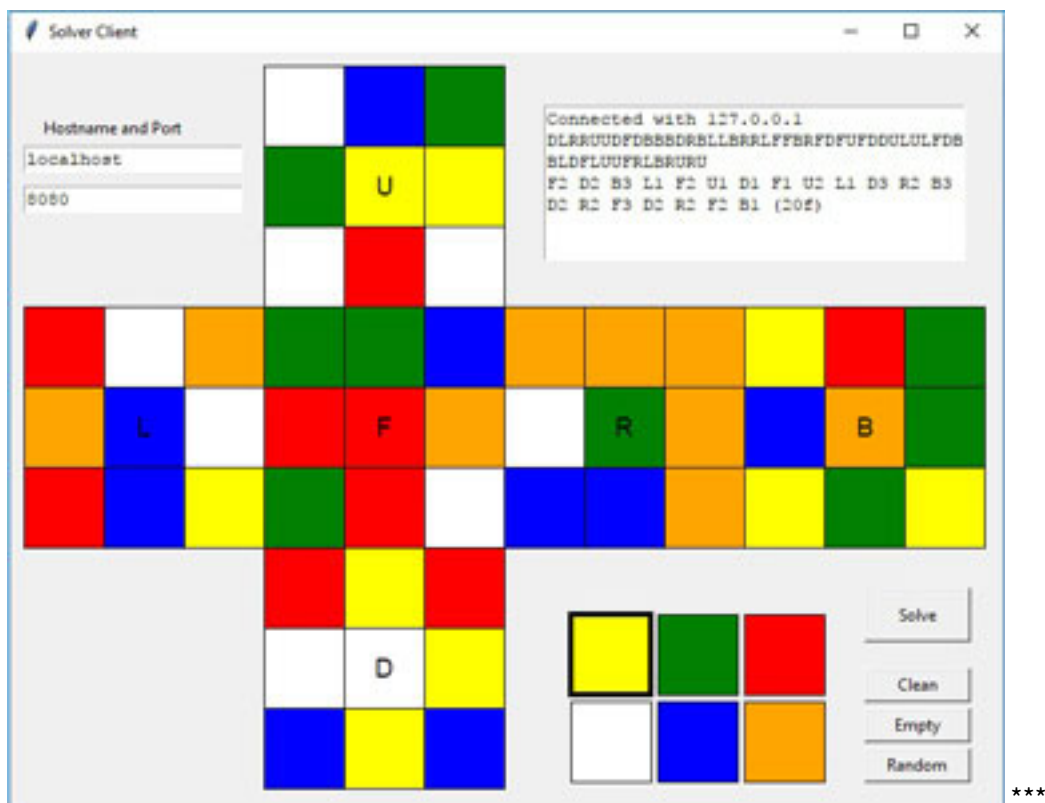
via a web browser and the server on the remote machine `myserver.com` on port 8081.

```
echo DUUBULDBFRBFRRULLLBRDFFFBLURDBFDFDRFRULBLUFDURRBLBDUDL | nc  
localhost 8080
```

using netcat (nc) and the server on the same machine on port 8080.

You can also communicate with the server using a small GUI program that allows you to interactively enter the cube definition string:

```
1 >>> import twophase.client_gui
```



Please note that the following module is experimental and requires the installation of the OpenCV package, which can be done with

```
$ pip install opencv-python
```

Additionally, you'll need the numpy package, which can be installed with

```
$ pip install numpy
```

Load the module with

```
1 >>> import twophase.computer_vision
```

---

Ensure that the webserver is running, and a webcam is connected to the client.

You can use the webcam to input the facelet colors. There are several parameters that affect the quality of the facelet detection. If you are using a Raspberry Pi with the Raspberry Pi Camera Module and not a USB webcam make sure you do “sudo modprobe bcm2835-v4l2” first.

You can find more information on how to set the parameters here: [Computer vision and Rubik's cube](#)

---

## Performance Metrics

We conducted tests solving 1000 random cubes in various scenarios on a Windows 10 machine with an AMD Ryzen 7 3700X 3.59 GHz. Differentiating between the standard CPython interpreter and PyPy (pypy3) with a Just-in-Time compiler yielded speedups of approximately 10x.

test(1000, t) generates 1000 random cubes, the computing time for each cube is t seconds. The distribution of the solving lengths also is given.

**Standard CPython** test(1000,30): {14: 0, 15: 2, 16: 12, 17: 74, 18: 279, 19: 534, 20: 99, 21: 0}, average 18.63 moves

test(1000,10): {14: 0, 15: 1, 16: 8, 17: 51, 18: 242, 19: 532, 20: 166, 21: 0}, average 18.79 moves

test(1000,1): {14: 0, 15: 2, 16: 4, 17: 28, 18: 127, 19: 401, 20: 405, 21: 33, 22: 0}, average 19.27 moves

test(1000,0.1): {15: 0, 16: 2, 17: 6, 18: 46, 19: 186, 20: 451, 21: 293, 22: 16, 23: 0}, average 20.02 moves

**PyPy (pypy3) with Just-in-Time compiler** test(1000,10): {14: 0, 15: 1, 16: 11, 17: 100, 18: 423, 19: 433, 20: 32, 21: 0}, average 18.37 moves

test(1000,1): {14: 0, 15: 1, 16: 10, 17: 49, 18: 259, 19: 535, 20: 145, 21: 1, 22: 0}, average 18.76 moves

test(1000,0.1): {15: 0, 16: 4, 17: 23, 18: 100, 19: 429, 20: 401, 21: 43, 22: 0}, average 19.33 moves

test(1000,0.01): {16: 0, 17: 1, 18: 25, 19: 95, 20: 349, 21: 461, 22: 69, 23: 0}, average 20.45 moves

To achieve an average of less than 19 moves, a computation time of 10 s in the case of CPython or 1 s in the case of PyPy is sufficient. For an average of 0.5 moves more, a computation time of 1 s with CPython and 0.1 s with PyPy is sufficient.

## Star History

