Rubik's Cube, Part I

State Representation:

Representation of the Cube state is done by a string with the following format:

"WWWW RRRR GGGG YYYY OOOO BBBB"

Which then can be represented in a 2-Dimesnional format by the **toGrid()** function. We can run this program by calling the **"print"** command from the terminal.

```
um43@tux3:~/cs510/a2$ sh run.sh print
   WW
   WW
OO GG RR BB
OO GG RR BB
   YY
   YY
```

By default, the command will print the **solved state**. It can also print a state if provided by user

```
um43@tux3:~/cs510/a2$ sh run.sh print "RWOR GOYB BOGW BRBW YWYO RGYG"
   RW
   OR
YW BO GO RG
YO GW YB YG
   BR
   BW
```

Goal State:

To check whether the current cube state is matches the goal state, the program calls the method **isSolved(state).** This method can be invoked by the **"goal"** command. If the state provided by user reaches solved state, it returns **True**, else it will return **False**.

```
um43@tux3:~/cs510/a2$ sh run.sh goal "RWOR GOYB BOGW BRBW YWYO RGYG"
False
```

```
um43@tux3:~/cs510/a2$ sh run.sh goal "WWWW RRRR GGGG YYYY BBBB OOOO"
True
```

Making a Move:

- The command **"applyMoveStr"** will call the method **"applyMoveStr(movesStr)".**
- Using the given move permutations in dictionary **MOVES** the methods **"applyMove(move)"** and **"applyMoveStr(movesStr)"** apply a single move or a sequence of moves.
- The methods can apply move on the given state, or it will take the default state ("WWWW RRRR GGGG YYYY OOOO BBBB").
- The available move set includes ["U", "U'", "R", "R'", "F", "F'", "D", "D'", "L", "L'", "B", "B'"], where U, R, F, D, L, B are Up, Right, Front, Down, Left, and Back by order. The letter

```
um43@tux3:~/cs510/a2$ sh run.sh applyMovesStr "R U' R'" "WWWW RRRR GGGG YYYY
OOOO BBBB"
   GW
   WR
WB OG YR BR
OO GW GR BB
   YO
   YY

um43@tux3:~/cs510/a2$ sh run.sh applyMovesStr "R U' R'"
   GW
   WR
WB OG YR BR
OO GW GR BB
   YO
   YY
```

Shuffle:

The command **"shuffle"** will scramble the default cube ("WWWW RRRR GGGG YYYY OOOO BBBB") using n number of random moves. It calls the method **"shuffle(n)",** which returns n random moves from move set and gives the result state

```
um43@tux3:~/cs510/a2$ sh run.sh shuffle 8
F F B' L' U D B' B'
   WR
   YO
GG OW BB WO
YO YY RW RR
   BB
   GG
```

Random Walk:

The **"random"** command-line command receives a positive integer, N, a sequence of moves, and a time limit in sec, T. It calls the method **"doRandomWalk()"**, which in turn does the following:

• apply the sequence of moves to the default state ("WWWW RRRR GGGG YYYY OOOO BBBB") and create an initial state,

• select one move at random, from the available move set ["U", "U'", "R", "R'", "F", "F'", "D", "D'", "L", "L'", "B", "B'"] ,

• execute that move,

• and stop if we've reached the goal or executed N moves, otherwise, go to the initial state and repeat if the running time limit, T, has not been reached.

And prints the following:

• The resulting sequence of moves and relative states to solve the initial state and if no solution was found in the time limit T, it prints “No solution in the time limit!!”

• the number of iterations for which the loop runs N times till it finds the solution.

• and the time that the function took to find the solution.

```
um43@tux3:~/cs510/a2$ sh run.sh random "L D' R' F R D'" 6 10
U B' U' F L F'
   BW
   GW
OY RR BB OY
WR BY OO WG
   YG
   RG

   GB
   WW
RR BB OY OY
WR BY OO WG
   YG
   RG

   WR
   WW
RR BB OG YG
GR BY OB OW
   YG
   OY

   RW
   WW
YG RR BB OG
GR BY OB OW
   YG
   OY

   RW
   RG
YY BR WB OG
GG YR WB OW
   OB
   OY

   WW
   GG
GY RR WB OO
GY RR WB OO
   BB
   YY

   WW
   WW
GG RR BB OO
GG RR BB OO
   YY
   YY
```

```
67555
1.98


um43@tux3:~/cs510/a2$ sh run.sh random "L D' R' F R D' F" 7 10
F' D L' U' R F L'
   BW
   RY
OY BR GB OY
WG YR WO WG
   OB
   RG

   BW
   GW
OY RR BB OY
WR BY OO WG
   YG
   RG

   BW
   GW
OY RR BB OY
WG WR BY OO
   RY
   GG

   RW
   WW
YG RR BB OG
OW GR BY OB
   OY
   YG

   WW
   RW
OG YG RR BB
OW GR BY OB
   OY
   YG

   WG
   RR
OG YY BR WB
OW GG YR WB
   OO
   YB

   WG
   WG
OO GY RR WB
OO GY RR WB
   YB
   YB
```

```
   GG
   GG
OO YY RR WW
OO YY RR WW
   BB
   BB

122194
4.13
```

After running several iterations various observations can be made, where:

- The number of moves N required to solve the cube can be less than or equal to the number of shuffles,
- Since the moves generated are always random, the time taken to solve the cube varies on each run. The scrambled move cube may solve within the first few iterations (got the solution in 0.006 sec in some instances for the above example) or it may not solve at all within the given time T.
- Few optimizations that I could think of were to store the closed moves (will remove duplicates), the first solution move should not include the first move executed during shuffle (will remove randomness to some degree).