Rubik's Cube, Part 2

Breadth-First Search (BFS)

> sh run.sh bfs [SHUFFLE MOVES]

To run the Breadth-First Search algorithm use the **bfs** command in terminal along with the **shuffle moves** (in the above example "L' B' U' D L' F B" are the shuffle moves).

This command will trigger the **doBFS()** method in the code, which give the following as the output.

- The sequence of move to solve the Initial State
- The relative state of the cube when those moves were applied
- The number of nodes that were explored
- The time that the function took to find the solution.

The sample output as run on tux:

```
um43@tux5:~/cs510/a2$ sh run.sh bfs "L D' R' F R D'"
U F' R' U R U'
   BW           GB           GB
   GW           WW           OO
OY RR BB OY  RR BB OY OY  RW BY GY OY
WR BY OO WG  WR BY OO WG  WW BB YO WG
   YG           YG           RR
   RG           RG           RG

   GW           OG           OO
   OO           OW           OO
RW BB YO GY  BB YO GY RW  BB YY GG WW
WW BO GY RG  WW BO GY RG  WW BB YY GG
   RY           RY           RR
   RB           RB           RR

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

1737399
125.48



um43@tux4:~/cs510/a2$ sh run.sh bfs "L' B' U' D L' F B"
F F U U F
   GG           GG           GG
   BB           BW           YY
WW RR YY RR  WY OR BY RR  WG OO BY RR
BB OO GG OO  BY OR BG OO  BY RR WG OO
   YY           GY           BB
   WW           WW           WW
```

```
   YG            YY            YY
   YG            GG            YY
OO BY RR WG  BY RR WG OO  BB RR GG OO
BY RR WG OO  BY RR WG OO  BB RR GG OO
   BB            BB            WW
   WW            WW            WW

581650
8.92
```

Depth Limited Search (DLS)

> sh run.sh dls [SHUFFLE MOVES] [DEPTH]

"dls" command will take a sequence of moves along with depth D. It will apply a sequence of moves to the default state and create an initial state and then run the depth limited search on it for depth D.

The dls command triggers the **doDLS()** method, which gives the following as output:

- The sequence of move to solve the Initial State
- The relative state of the cube when those moves were applied
- The number of nodes that were explored
- The time that the function took to find the solution.

The sample output as run-on tux:

```
sh run.sh dls "L D' R' F R D'" 8
U U F R F' U' F' U'
   BW            GB            WG
   GW            WW            WB
OY RR BB OY  RR BB OY OY  BB OY OY RR
WR BY OO WG  WR BY OO WG  WR BY OO WG
   YG            YG            YG
   RG            RG            RG

   WG            WO            WO
   RB            RY            BO
BY BO WY RR  BY BO BW BR  BY OG WW BR
WG YY BO WG  WG YG OY GG  WR BY OY GG
   OO            OW            YG
   RG            RR            RR

   OO            OO            OO
   WB            OO            OO
BR BY OG WW  BB YY GG WW  WW BB YY GG
WR BY OY GG  WW BB YY GG  WW BB YY GG
   YG            RR            RR
   RR            RR            RR

354463
2.49
```

```
um43@tux5:~/cs510/a2$ sh run.sh dls "L' B' U' D L' F B" 8
U U F F U U F'
   GG           BG           BB
   BB           BG           GG
WW RR YY RR  RR YY RR WW  YY RR WW RR
BB OO GG OO  BB OO GG OO  BB OO GG OO
   YY           YY           YY
   WW           WW           WW

   BB           BB           YB
   BY           YY           YB
YY OR GW RR  YG OO BW RR  OO BW RR YG
BY OR GG OO  BW RR YG OO  BW RR YG OO
   GW           GG           GG
   WW           WW           WW

   YY           YY
   BB           YY
BW RR YG OO  BB RR GG OO
BW RR YG OO  BB RR GG OO
   GG           WW
   WW           WW

371857
2.63
```

Iterative Deepening Search (IDS)

> sh run.sh ids [SHUFFLE MOVES] [DEPTH]

"ids" command will take a sequence of moves along with depth D. It will apply a sequence of moves to the default state and create an initial state and then run the depth limited search on it for depth D.

The ids command triggers the **ids()** method, which gives the following as output:

- Number of nodes traversed for each depth before the goal state is achieved
- The sequence of move to solve the Initial State
- The relative state of the cube when those moves were applied
- The number of nodes that were explored
- The time that the function took to find the solution.

The sample output as run-on tux:

```
um43@tux4:~/cs510/a2$ sh run.sh ids "L' B' U' D L' F B" 20
Depth: 0 d: 0
Depth: 1 d: 12
Depth: 2 d: 156
Depth: 3 d: 1590
Depth: 4 d: 15864
Depth: 5 d: 57064
IDS found a solution at depth 5
F F U U F
```

```
   GG           GG           GG
   BB           BW           YY
WW RR YY RR  WY OR BY RR  WG OO BY RR
BB OO GG OO  BY OR BG OO  BY RR WG OO
   YY           GY           BB
   WW           WW           WW

   YG           YY           YY
   YG           GG           YY
OO BY RR WG  BY RR WG OO  BB RR GG OO
BY RR WG OO  BY RR WG OO  BB RR GG OO
   BB           BB           WW
   WW           WW           WW

74686
0.4

um43@tux5:~/cs510/a2$ sh run.sh ids "L D' R' F R D'" 20
Depth: 0 d: 0
Depth: 1 d: 12
Depth: 2 d: 156
Depth: 3 d: 1590
Depth: 4 d: 15864
Depth: 5 d: 157956
Depth: 6 d: 58194
IDS found a solution at depth 6
U F' R' U R U'
   BW           GB           GB
   GW           WW           OO
OY RR BB OY  RR BB OY OY  RW BY GY OY
WR BY OO WG  WR BY OO WG  WW BB YO WG
   YG           YG           RR
   RG           RG           RG

   GW           OG           OO
   OO           OW           OO
RW BB YO GY  BB YO GY RW  BB YY GG WW
WW BO GY RG  WW BO GY RG  WW BB YY GG
   RY           RY           RR
   RB           RB           RR

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

233772
0.41
```

A* Search

> sh run.sh astar [SHUFFLE MOVES]

To run the A* algorithm use the **astar** command in terminal along with the **shuffle moves** (in the above example "L' B' U' D L' F B" are the shuffle moves).

The heuristic calculates the distance between the same colors located at different positions. When the state reaches the goal state, all the color(W, G, Y, O, R, B) are positioned together and will have cost 0. Added to that is the number of moves applied to the initial state. This heuristic leans both towards finding the solution in least amount of time possible and having less number of moves. Although in some cases it can lean more towards one side.

This command will trigger the **doAstar()** method in the code, which give the following as the output.

- The sequence of move to solve the Initial State
- The relative state of the cube when those moves were applied
- The number of nodes that were explored
- The time that the function took to find the solution.

The sample output as run-on tux:

```
um43@tux4:~/cs510/a2$ sh run.sh astar "L D' R' F R D'"
D R' F' L F U'
   BW           BW           BO
   GW           GW           GO
OY RR BB OY  OY RR BB OY  OY RW BY GY
WR BY OO WG  WG WR BY OO  WG WW BB YO
   YG           RY           RR
   RG           GG           GR

   BO           OO           OO
   BB           YB           OO
OO WW RY GY  WO BW RY GG  WW BB YY GG
WG RW RB YO  GO BW RB YY  GG WW BB YY
   YG           WG           RR
   GR           RR           RR

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

5615
0.24


um43@tux5:~/cs510/a2$ sh run.sh astar "L' B' U' D L' F B"
B' B' D' D' F
   GG           BW           WW
   BB           BB           BB
WW RR YY RR  WW RR YG RO  GW RR YB OO
BB OO GG OO  WB OO GG RO  YB OO GW RR
   YY           YY           YY
   WW           GY           GG
```

```
   WW            WW            WW
   BB            BB            WW
GW RR YB OO  GW RR YB OO  GG RR BB OO
OO GW RR YB  GW RR YB OO  GG RR BB OO
   YG            GG            YY
   YG            YY            YY

2155
0.08
```

Normalization

> sh run.sh norm [state]

norm command will normalize any given state. Passing any state to the command norm will rotate the given state such that colors green, yellow, and orange are in position 10, 12 and 19 with blue, white and red as opposites. This creates a mapping where we can swap colors based on color mapping of solved cube state.

```
um43@tux4:~/cs510/a2$ sh run.sh norm "BBGG OYRR WWOY GBGB WRYY OOWR"
   WW
   YY
RB RR GO GG
OO GO BB RB
   YW
   YW


um43@tux5:~/cs510/a2$ sh run.sh norm "OWGG OBRR WWGY RBYB GROY OYWB"
   WR
   GG
GY RR WB WO
WO GO YY RB
   YB
   OB
```

Competition

> sh run.sh competition [SHUFFLE MOVES]

Using the command competition along with the shuffle moves will run the algorithm. Here, I'm made use of the A* algorithm with some added constraints for selecting the next move. Also, changing the weight of moves required according to the depth of solution helped in improving the time incase the sequence of moves are found at a depth greater than 8. Also, adding an extra step to make sure that no two moves are repeated eliminated a large number of states traversed, in short saving a lot of time in some cases. But it can often lead to having many move sequences to find the solution.

The sample output as run on tux:

```
um43@tux5:~/cs510/a2$ sh run.sh competition "L' B' U' D L' F B"
B' B' D' U' R
```

```
   GG            BW            WW
   BB            BB            BB
WW RR YY RR  WW RR YG RO  GW RR YB OO
BB OO GG OO  WB OO GG RO  YB OO GW RR
   YY            YY            YY
   WW            GY            GG

   WW            WB            WW
   BB            WB            WW
GW RR YB OO  OO GW RR YB  OO GG RR BB
OO GW RR YB  OO GW RR YB  OO GG RR BB
   YG            YG            YY
   YG            YG            YY

1734
0.0644
```

<u>Analyzing the Time and Space Complexity:</u>

BFS: The time and space complexity of BFS is **$O(b^d)$,** where b is the branching factor of nodes and d is the depth at which the solution is found. BFS will find the solution with the least number of moves i.e., the solution the lowest depth while traversing the node. BFS has the worst space complexity out of the present algorithms and depending on the shuffle moves it can take considerable time to get the solution state.

DLS: For Depth Limited Search, the time complexity for worst scenario is **$O(b^l)$**, where b is the branching factor of cube and l is the depth given. In the case of cube the program runs recursively for each node. At a worse space complexity is O(bl) as we access each node recursively.

IDS: For iterative deepening search, the space and time complexity at worst case will be similar to DLS. But IDS gives an optimal solution much faster than BFS.

A* Search: The time complexity in case A* search is dependent on the heuristics used to narrow down the search. But in the worst case, A* can match the time complexity of BFS which is **$O(b^d)$,** where b is the branching factor of nodes and d is the depth at which the solution is found.