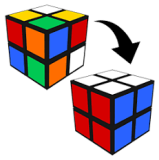
Artificial Intelligence



Rubik's Cube II (12pts)

Part II: Searching for Solutions

In the previous part, we implemented key components of a solver for $2 \times 2 \times 2$ Rubik's Cube. In this part, we continue with this implementation and significantly extend it to find puzzle solutions using various methods.
Please start with your code from part 1 and extend it as directed below. As before, the code for this part should be written in Python to run on tux.cs.drexel.edu, and we will use the same **run.sh** shell script for testing. Again, **you may only use built-in standard libraries (e.g., math, random, time, etc.); you may NOT use any external libraries or packages like Numpy** (if you have any questions at all about what is allowable, please email the instructor).

Terminology

Default state: the solved Rubik's Cube

```
"WWWW RRRR GGGG YYYY OOOO BBBB"
```

Initial state: a permuted version of the default state (shuffled or scrambled version)

Breadth-First Search (BFS) (3pts)

Write a method that does a breadth-first search from a given cube, returning the first sequence of moves found that reaches the solution state. Add a "**bfs**" command-line command that receives a sequence of moves; applies the sequence of moves to the default state to create an initial state; and then runs the breadth-first search on it.
The output should print the following:

- the entire sequence of moves and relative states to solve the changed cube with **3 cubes** per line,
- the number of nodes that were explored,
- and the time that the function took to find the solution.

Here are two examples:

```
> 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
```

```
1691107
29.45

> 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


553868
4.20
```

Depth Limited Search (DLS) (3pts)

Similar to BFS, write a method that does a depth-limited search from a given cube, returning the first sequence of moves found that reaches the solution state. Add a "**dls**" command-line command that receives a sequence of moves and depth, D; applies the sequence of moves to the default state to create an initial state; and then run the depth-limited search on it for the depth D. The output should print the following:

- the entire sequence of moves and relative states required to solve the changed cube with **3 cubes** per line (print an error message if the algorithm fails to find a solution at the given depth),
- the number of total nodes that were explored across all levels,
- and the time that the function took to find the solution.

Here are two examples:

```
> sh run.sh ids "L D' R' F R D'" 8
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

nodes: 45543
```

```
time: 0.42

> sh run.sh ids "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

257524
2.35
```

Iterative Deepening Search (IDS) (2pts)

Write a method that does an iterative deepening search from a given cube, returning the first sequence of moves found that reaches the solution state. Add an "**ids**" command-line command that receives a sequence of moves and depth, D; applies the sequence of moves to the default state to create an initial state; and then runs the iterative deepening search on it for the depth D. The output should print the following:

- the algorithm's current search depth, and the number of nodes it explored at that depth (print an error message if the algorithm fails to find a solution at the given depth),
- the entire sequence of moves and relative states required to solve the changed cube with **3 cubes** per line,
- the number of total nodes that were explored across all levels,
- and the time that the function took to find the solution.

Here are two examples:

```
> sh run.sh ids "L D' R' F R D'" 20
Depth: 0 d: 0
Depth: 1 d: 12
Depth: 2 d: 132
Depth: 3 d: 1320
Depth: 4 d: 13092
Depth: 5 d: 129732
Depth: 6 d: 45543
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

189831
1.67

> sh run.sh ids "L' B' U' D L' F B" 20
Depth: 0 d: 0
Depth: 1 d: 12
Depth: 2 d: 132
Depth: 3 d: 1320
Depth: 4 d: 13092
Depth: 5 d: 47499
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


62171
0.54
```

A Search (3pts)*

Write a method that does an A* search from the given cube, returning the first sequence of moves found that reaches the solution state. Note that as part of this process, you will need to choose and implement an *admissible* heuristic function $h(n)$, such that A* can reasonably estimate the minimum cost from a given state to the goal state. For example, one simple heuristic for solving a Rubik's Cube is a three-dimensional version of the Manhattan distance. For each corner, compute the minimum number of moves required to move them to their correct locations, and sum these values over all corners. To be admissible, this value must be divided by 4, since every move moves 4 cubes. Of course, you are free to develop your own admissible heuristics and perhaps find better results.

Add an "**astar**" command-line command that allows a user to perform the search.
Similar to the BFS, the output should print the following:

- the entire sequence of moves and relative states to solve the changed cube with **3 cubes** per line,
- the number of nodes that were explored,
- and the time that the function took to find the solution.

Here are two examples:

```
> sh run.sh astar "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

58958
11.48

> sh run.sh astar "L' B' U' D L' F B"
F' F' U U F
   GG             GG             GG
   BB             YG             YY
WW RR YY RR    WB RO YY RR    WG OO BY RR
BB OO GG OO    BB RO YG OO    BY RR WG OO
   YY             WB             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


5722
0.13
```

Depending on the way that you implement your heuristic, your number of **explored nodes** may differ from the number in this example but should be less than the number for BFS and IDS.

Final Notes:

- When printing out the sequence of states, it should print 3 cubes per line, and if there are more than 3 cubes, it should print the first 3 cubes and then continue the rest on a new row of cubes (similar to the examples). This will have **1 point** of the total.
- To check whether your solution really solves the puzzle or not, you can go to the simulation website and make all the input moves returned by the solution of the algorithm. These moves should solve the cube, assuming the same initial states.
- BFS is not a very efficient algorithm (time-complexity-wise), so it might take some time for it to solve the problem if the solution requires more than 5 or 6 moves.
- If you pay attention to DLS result, you will notice that the solution is not optimal. Why do you think that is?
- For finding the number of explored nodes, you can count the number of nodes that you check whether they are goal or not.

- Try to play with the number of moves you are applying to the default state. Use the shuffling function. See how long it will take for your algorithms to find the solution with 5, 6, 7, … levels of shuffling. Try to make sense of the numbers. Discuss this in your documentation.

Improving the Speed with Move Sequence (not mandatory for the assignment)

To improve the running time of your algorithm, add a few functionalities to the **Cube** class from the last part in a way that allows it to store the sequence of moves made thus far. Add whatever methods are useful for implementing the search algorithms.

One functionality that can help us solve the problem with different search algorithms that we learned in the class is to remove unnecessary moves from consideration at each state, meaning that it is important to ignore moves that will not help us in the search process. This can improve the algorithm's complexity and efficiency. Here are some ideas for you:

1. Every move has an *inverse*. Do not make a move that is the inverse of the previous move. Instead, remove it from the list of possible moves.

   inverse(F) = F', inverse(F') = F, etc.

2. Also, every move has a *complementary move*, that is, one that does the same thing to the other half of the cube. Thus, these moves combine to simply rotate the entire cube without changing anything.

   UD', D'U : rotate the whole cube left
   U'D, DU' : rotate the whole cube right
   L'R, RL' : rotate the whole cube forward
   LR', R'L : rotate the whole cube backward
   FB', B'F : rotate the whole cube clockwise
   F'B, BF' : rotate the whole cube counter-clockwise

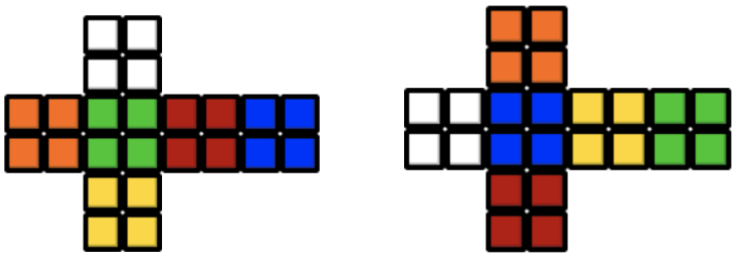
   complement(U) = D' , complement(D') = U, etc.

3. There is no point in applying the same move consecutively 3 or more times. Applying a move 4 times returns to the original state. Applying a move 3 times is equivalent to the inverse move.

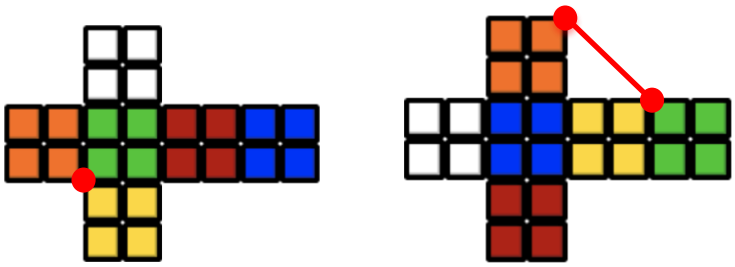
4. Perhaps there are other combinations of moves that should be removed. Please feel free to add to this list. Be sure to explain in your program documentation.

Extra Credit I – Normalization (1pts)

Notice that in the first part of this assignment, the state comparison function has a problem. Consider the following two states:



The previous function will consider these two states as different. However, it's quite obvious that the states are equivalent. In fact, there are many different configurations equivalent to the given state because merely observing the cube from a different position, without performing any operations on it, will provide a different mapping of the colors. Below, the same configurations are shown. On the left, there is a red dot at the *only* corner of the cube where green, yellow, and orange faces intersect. On the right, there are two red dots connected by a line, showing the same corner but with a cube that has first been rotated before being flattened. Our goal is to define a ***normal form*** that will allow us to recognize all cubes in similar states.



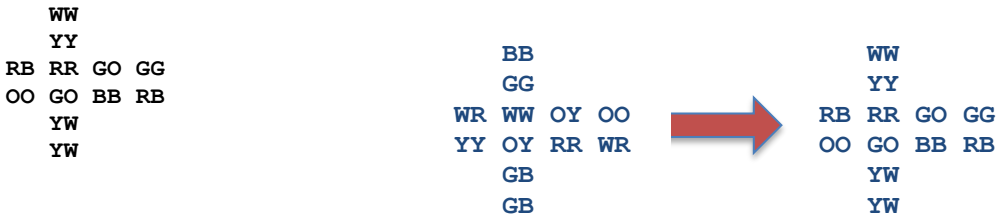
For example, assume that we are normalizing based on the corner identified by the single red dot on the left image above. This is the little cube in the corner that includes indexes **`10`**,**`12`**,**`19`**. We will change all the colors in the state as described below so they have the desired colors in that corner. Remember that this is only for checking if two states are symmetric with each other, even if the representations are different.

How to normalize a state:
Given a state, swap colors in the state so that the colors now in cells 10, 12, and 19 are changed to Green, Yellow, and Orange. Because this represents a single cube with all six colors, and the sides opposite Green, Yellow, and Orange on any cube are respectively Blue, White, and Red, this also means that we must switch the "opposite colors" to those in cells 10, 12, 19 to the "opposites" of Green, Yellow, and Orange (Blue, White, and Red). If you imagine a 3D Rubik's cube, the color "opposite" to it is the color on the opposing side of the cube if it were solved.

*The most important thing to understand is that if color **A** is opposite to color **B**, and color **C** is opposite to color **D**, then in order to swap the locations of **A** and **C**, we must also swap **B** with **D.*** The reason that we make the opposite color change is to prevent an infinite loop of swapping colors.

For extra credit, write a function that, given a state, transforms it into normal form. It should be runnable from the command line with the "**norm**" command as the first argument and the state as the second argument. See the expected effect of this function below:

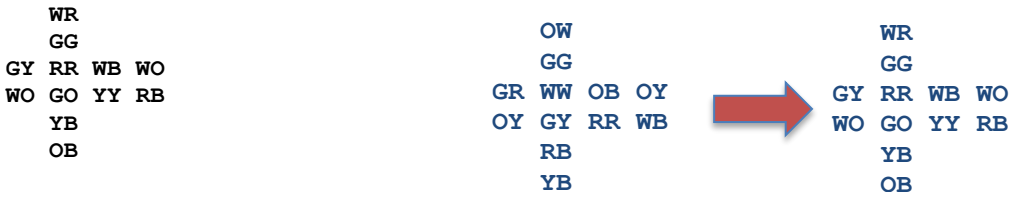
```
> sh run.sh norm "BBGG OYRR WWOY GBGB WRYY OOWR"
```



In this case, the elements in cells 10, 12, 19 are Orange, Green, Yellow. This means their opposite sides are Red, Blue, and White. Thus, the normalized form of this state can be found by creating a new state, and filling its elements according to the following mapping from colors in the original state to those in the normalized state:

```
Orange→ Green ,  Green→ Yellow ,  Yellow→ Orange,
Red   → Blue  ,  Blue → White  ,  White → Red
```

```
> sh run.sh norm "OWGG OBRR WWGY RBYB GROY OYWB"
```



In this case, the elements in cells 10, 12, 19 are Yellow, Green, Red. This means their opposite sides are White, Blue, and Orange. Thus, the normalized form of this state can be found by creating a new state, and filling its elements according to the following mapping from colors in the original state to those in the normalized state:

```
Green → Green ,  Red   → Yellow,  Yellow→ Orange,
Blue  → Blue  ,  Orange→ White ,  White → Red
```

Important notes:

- For computing the heuristic, we recommend working with the normalized version of the cube.
- If you end up creating the normal form, make sure to use the **norm()** function when checking for repeated states. Check out the numbers with and without using the **norm()** function, so you make sure that the function is working.

Extra Credit II (1pts)

For more extra points, you can try to improve the run-time of your algorithm more and compete against the rest of the class (shorter run-time is better). We will check each competitor's timing with a few cubes with solutions at depths of 7, 8, and 9. The top 5 agents will receive extra credit up to 10 percent. Remember that to have a chance to win the competition, your agent should be able to complete the run in less than 4 minutes, even for the solutions at depth 9.

There are a few ways that you can improve the algorithm. The first is to work on improving your data structure and search heuristics. Second, you could implement a variant of A* like Bidirectional A* or IDA*.

If you would like to enter the competition for extra credit, you have to specifically mention that in your document and add a "**competition**" command-line command that, just as before,

receives a sequence of moves; applies the sequence of moves to the default state to create an initial state; and then runs the algorithm on it. Again, the output should print the following:

- the entire sequence of moves and relative states to solve the shuffled cube with **3 cubes** per line,
- the number of nodes that were explored,
- and the time that the function took to find the solution.

Academic Honesty

Please remember that you must write all the code for this (and all) assignments by yourself, on your own, without help from anyone except the course TA or instructor.

Submission

Remember that your code must run on **tux.cs.drexel.edu**—that's where we will run the code for testing and grading purposes. Code that doesn't compile or run there will receive a grade of zero.

For this assignment, you must submit:

- Your Python code for this assignment.
- Your **run.sh** shell script that can be run as noted in the examples.
- We should be able to run all the functions mentioned above (**bfs**, **dls**, **ids**, **astar**, **norm** for extra credit, and **competition** if you would like to enter the competition) using your **run.sh** shell script.
- A PDF document with written documentation containing a few paragraphs
    - explaining your program,
    - analyzing the time and space complexity of algorithms and comparison (one or two paragraphs is enough),
    - explaining the heuristic you used for A*,
    - results showing testing of your routines,
    - and if you did anything extra as well as anything that is not working.

Please use a compression utility to compress your files into a single ZIP file (NOT RAR, nor any other compression format). The final ZIP file must be submitted electronically using Blackboard—do not email your assignment to a TA or instructor! If you are having difficulty with your Blackboard account, you are responsible for resolving these problems with a TA or someone from IRT before the assignment is due. If you have any doubts, complete your work early so that someone can help you.