RUBIK'S CUBE[2x2]

ABSTRACT:

The Rubik's Cube is a fascinating, timeless puzzle with quintillions of possible states. Solving a Rubik's Cube is a real world problem. As a cube geek, I found myself trying to figure out how to solve this programmatically. Like everyone, my first intuition for this problem was brute force. I have tried and implemented the brute force algorithm in this code.

The idea behind brute force is to divide the cube into six individual sides, each of these sides further divided into 2 rows, and each of these divided into left and right cells. In other words, each cube has six sides: up, down, left, right, front, back; each side has two rows: upper and lower rows; and each of these rows have a left and a right cell. We can further note that every side is capable of three types of twists: clockwise, anticlockwise and one-eighty degree twists. As we have sides, this turns out to be eighteen different possible twists for every move.

In this method, you can find me define three classes:

1. Row.h : to define the left and the right cells along with the function to flip the left and the right sides of a row when going between the non corresponding halves and correct the alignment.
2. Side.h : to define the upper and lower rows along with the functions to define the configuration of the sides when subjected to clockwise or anticlockwise twists.
3. Cube.h : to define the six different sides along with the functions to define the configuration of the entire cube when subjected to any of the eighteen twists.

Finally, the solver.cpp has the main program which takes an input configuration of the cube and solves it layer by layer by looking for the correct orientation of the individual cubies (A cubie is defined as one of the 8 mini cubes). The program calculates the solution in 3 main steps: solving the first layer, orienting the last layer, and solving the last layer. In the first step, I chose to solve the white face on the upper face first. The "middle layer" has to be aligned in this order (going left to right): red, blue, orange, green. Again I chose to make the front face red, which means that the rest of the sides will be solved according to this pattern. I have chosen to keep the cube and the colors constant. So the algorithms here are the ones I typically use that can be applied to many permutations while solving the first layer. The second step consists of determining which of the 7 possible states the cube is in and using the corresponding algorithm to solve the yellow face(down face) (these states and algorithms can be found here: http://www.cubewhiz.com/ortegaoll.php). Similarly, the third step involves determining which of the states the cube is in and using the corresponding algorithm to solve the middle layers, effectively solving the entire cube (states and algorithms available here: http://www.cubewhiz.com/ortegapbl.php).

MATHEMATICS:

- Since we have 8 corner cubies, they have 8! Permutations possible.
- Each of the cubies have three possible orientations: actual, clockwise rotated, anticlockwise rotated (any of the three faces can face up). But the orientations of seven of the cubies dictate the orientation of the eighth, by the laws of the cube. Therefore, there are 37ways that a corner can be oriented.
- Since I am keeping my cube stationary, there are 6 possible ways to select the 'Up' face and 4 possible ways to select the 'Front' face, reducing the number of permutations by a factor of 24.

Therefore, the total number of possible permutations of a 2x2 cube is:

$$8! * 3^7 / 24 = 36{,}74{,}160 \text{ permutations.}$$

NOTATION:

The notation used is as follows: F corresponds to the front side, B to the back side, L to the left side, R to the right side, U to the up side, and D to the down side, with regards to how the user is holding the cube. Turns are assumed to be clockwise by default (F is a clockwise turn of the front side/face), but a counter-clockwise turn is denoted by a ' (prime). Likewise, a double-turn is denoted by a 2 (it does not matter which direction the user turns in). For example, a clockwise turn of the front face followed by a counter-clockwise turn of the front face followed by two turns of the front face would be written as F F' F2. It's important to keep in mind which direction is clockwise -- R corresponds to a downward turn on the right side as viewed from the front face, but L corresponds to an upward turn on the left side as viewed from the front face. Since these mistakes are easy to make, the program displays the current state the cube should be in along the way, so that the user can compare their cube to prevent any mistakes.
Color notation:

0: white	2: blue	4: green
1: red	3: orange	5: yellow

ACCURACY:

This Rubik's cube solver is capable of solving a given cube in 10 - 40 moves. The solutions generated aren't even close to the fastest solutions possible (any 2x2 cube can actually be solved in only 11 or fewer turns), but the program reliably solves the cube every time, and makes the steps distinct and reasonably easy to follow. However, it still computes solutions in around 70 milliseconds on average, which is fast enough to not be noticeable by the user.

FUTURE SCOPE:

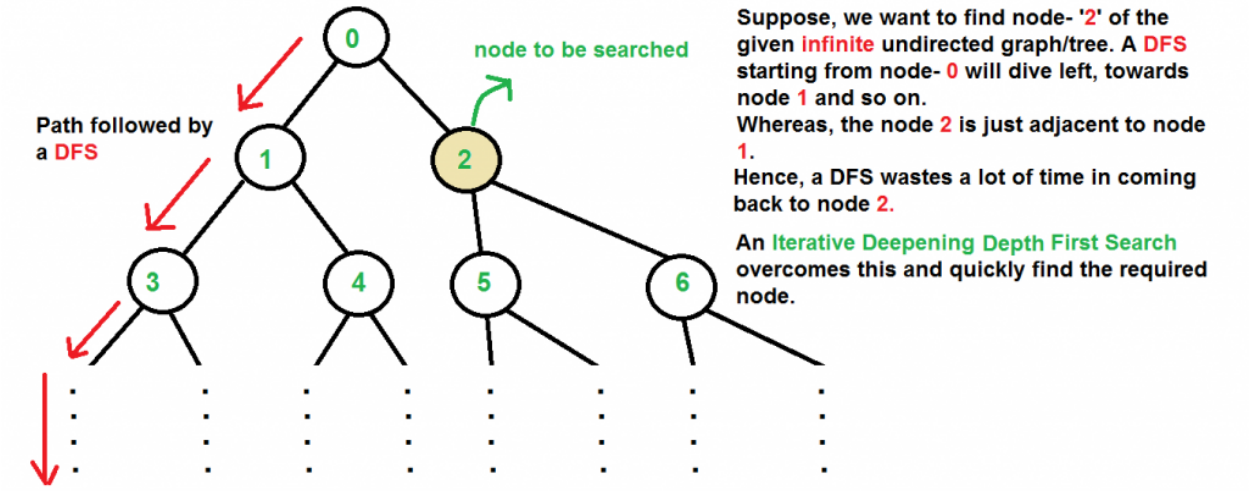
In the future, I plan to implement the cube with the help of the following algorithms:

1. IDDFS (Iterative Deepening Depth First Search).
2. A* algorithm.

3. Korf's algorithm.

THE NEXT STEP:

IDDFS: There are two common ways to traverse a graph, BFS and DFS. Considering a Tree (or Graph) of huge height and width, both BFS and DFS are not very efficient due to the following reasons: DFS first traverses nodes going through one of the adjacent nodes of the root, then the next adjacent. The problem with this approach is, if there is a node close to root, but not in the first few subtrees explored by DFS, then DFS reaches that node very late. Also, DFS may not find the shortest path to a node (in terms of number of edges); and BFS goes level by level, but requires more space. The space required by DFS is O(d) where d is depth of the tree, but space required by BFS is O(n) where n is the number of nodes in the tree.



IDDFS combines depth-first search's space-efficiency and breadth-first search's fast search (for nodes closer to root). IDDFS calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond the given depth. So basically we do DFS in a BFS fashion.

Algorithm:

```
// Returns true if target is reachable from src within max_depth

bool IDDFS(src, target, max_depth)
  for limit from 0 to max_depth
    if DLS(src, target, limit) == true
      return true
  return false

bool DLS(src, target, limit)
  if (src == target)
    return true;
  // If reached the maximum depth, stop recursion.
  if (limit <= 0)
    return false;
```

foreach adjacent i of src
  if DLS(i, target, limit?1)
    return true

return false

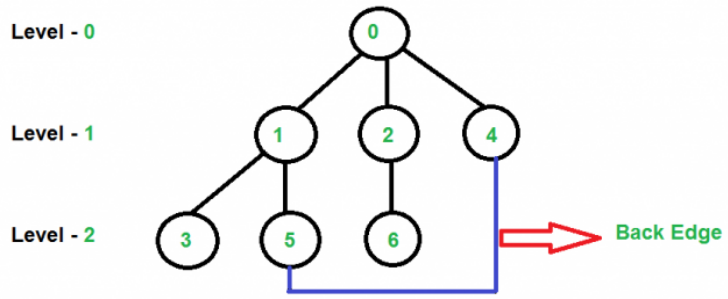
An important thing to note is, we visit top level nodes multiple times. The last (or max depth) level is visited once, second last level is visited twice, and so on. It may seem expensive, but it turns out to be not so costly, since in a tree most of the nodes are in the bottom level. So it does not matter much if the upper levels are visited multiple times.

Illustration:

There can be two cases-

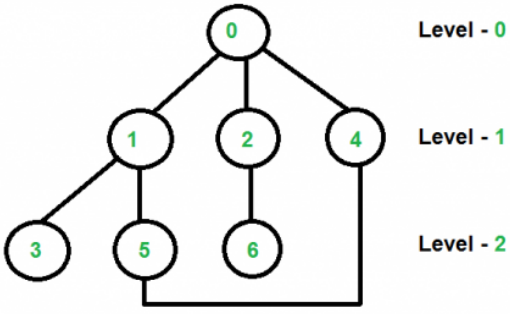
a) When the graph has no cycle: This case is simple. We can DFS multiple times with different height limits.

b) When the graph has cycles. This is interesting as there is no visited flag in IDDFS.



Although, at first sight, it may seem that since there are only 3 levels, so we might think that Iterative Deepening Depth First Search of level 3, 4, 5,...and so on will remain same. But, this is not the case. You can see that there is a cycle in the above graph, hence IDDFS will change for level-3,4,5..and so on.

Depth	Iterative Deepening Depth First Search
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4 5
3	0 1 3 5 4 2 6 4 5 1



The explanation of the above pattern is left to the readers.

Time Complexity: Suppose we have a tree having branching factor 'b' (number of children of each node), and its depth 'd', i.e., there are **bd** nodes.

In an iterative deepening search, the nodes on the bottom level are expanded once, those on the next to bottom level are expanded twice, and so on, up to the root of the search tree, which is expanded d+1 times. So the total number of expansions in an iterative deepening search is-

(d-1)1 + (d)b + (d-1)b2 + ... + 3bd-2 + 2bd-1 + bd

That is,

Summation[(d + 1 - i) bi], from i = 0 to i = d

Which is same as O(bd)

After evaluating the above expression, we find that asymptotically IDDFS takes the same time as that of DFS and BFS, but it is indeed slower than both of them as it has a higher constant factor in its time complexity expression.

IDDFS is best suited for a complete infinite tree

A comparison table between DFS, BFS and IDDFS

	Time Complexity	Space Complexity	When to Use ?
DFS	$O(b^d)$	O (d)	=> Don't care if the answer is closest to the starting vertex/root. => When graph/tree is not very big/infinite.
BFS	$O(b^d)$	$O(b^d)$	=> When space is not an issue => When we do care/want the closest answer to the root.
IDDFS	$O(b^d)$	O(bd)	=> You want a BFS, you don't have enough memory, and somewhat slower performance is accepted. In short, you want a BFS + DFS.

A* ALGORITHM:

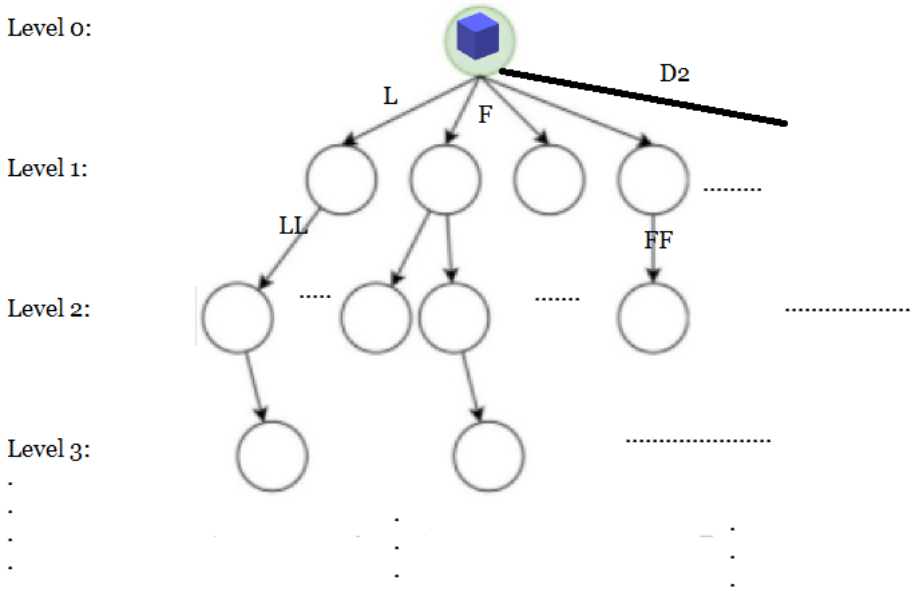
Used in path-finding and graph traversals. Many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).
At each step it picks the node according to a value-'**f**' which is a parameter equal to the sum of two other parameters – '**g**' and '**h**'. At each step it picks the node/cell having the lowest '**f**', and processes that node/cell.

$$f = g + h$$

We define '**g**' and '**h**' as simply as possible below
g = the movement cost to move from the starting point to a given node on the graph, following the path generated to get there.
h = the **estimated** movement cost to move from that given node on the graph to the final destination. This is often referred to as the heuristic
Dijkstra is a special case of A* Search Algorithm, where h = 0 for all nodes.

IMPLEMENTATION [IDA*] :

Consider a cube as the root of a tree, i.e. depth 0. Applying each possible twist of the cube (L, F, R, etc.) brings the cube to a new node at depth 1, any of which may be the solved state. If not, applying each combination of two moves (LF, LR, LU, etc.) may solve the cube. That continues until a solution is found.



IDDFS alone would take too much time to solve most scrambled cubes. There are 18 possible face twists of the cube, a large branching factor. After the first set of twists, some of the moves can be redundant. For example, turning the same face twice is redundant: FF is the same as F2; FF' is the same as no move; FF2 is the same as F'; and so on. Also, some moves are commutative: FB is the same as BF; U2D is the same as DU2; etc. But even after removing these, the branching factor is over 13, so searching for a solution with raw IDDFS would take thousands of years on a modern computer!

For A* to operate correctly — specifically, to guarantee an optimal path — the heuristic must never overestimate the distance. IDA* works the same way as IDDFS, but rather than starting the search at depth 0 it queries a heuristic for an estimated distance to the goal state and starts at that depth. During the search, if the estimated distance through a node to the solved state exceeds the search depth, then the node is ignored from the tree. In other words, the heuristic is used to see if moving from one state of the cube to another brings the cube closer to or farther away from the solved state. If the twist moves the cube farther away from the solved state, then that branch of the search can be ignored.

Putting this in lay terms, a permissible heuristic for the Rubik's Cube is a function that accepts a scrambled cube and estimates or underestimates the number of moves to solve the cube. One way to create such a heuristic function is to create a database of every possible scramble of a the

cubies. For example, a database that takes in the indexes and orientations of the 8 corner cubies and returns the number of moves to solve just the corners.

PATTERN DATABASES: Korf suggests using multiple databases:

1. stores the number of moves required to solve the corner pieces of any scramble.
2. two additional databases: one for 6 of the 12 edges, and another for the other 6 edges.
3. one additional database that holds the permutations of the 12 edges and so on.

Using larger edge databases and the additional edge permutation database results in a huge speed increase. Larger databases would result in an even bigger performance increase, but it's easy to use an enormous amount of memory.

An implementation detail that Korf uses in his algorithm is how to create indexes into these pattern databases. That is, given a scrambled cube, how to create a perfect hash out of the indexes and orientations of the corners.

The indexing works by first generating the Lehmer code of the cubie index permutation, then converting the Lehmer code to a base-10 "rank." (The Lehmer code is a way of numbering permutations lexicographically). For the 8 corners, the ranks would be.

```
Permutation         Rank
(0 1 2 3 4 5 6 7)   0
(0 1 2 3 4 5 7 6)   1
(0 1 2 3 4 6 5 7)   2
(0 1 2 3 4 6 7 5)   3
...
(4 0 1 2 3 5 6 7)   20160
...
(7 6 5 4 3 2 1 0)   40319
```

Then the permutation of the orientations of 7 of the corner cubies is ranked in a similar manner. (Recall from above that the orientation of 7 of the corners dictate the orientation of the 8th.)

```
Permutation       Rank
(0 0 0 0 0 0 0)   0
(0 0 0 0 0 0 1)   1
(0 0 0 0 0 0 2)   2
(0 0 0 0 0 1 0)   3
...
(1 0 0 0 0 0 0)   729
...
(2 0 0 0 0 0 0)   1458
...
```

- `(2 2 2 2 2 2 2)  2186`

Lastly, those two ranks are combined together to form an index into the database.

```
index = rankWithoutReplacement(cornerIndexes) * 3^7 +
        rankWithReplacement(cornerOrientations)
```

Since the Korf algorithm uses multiple pattern databases, the estimated number of moves to the solved state, the A* heuristic, is the **maximum** value returned from all of the pattern databases. More concretely, given a cube state where the 8 corner cubies are 4 moves away from solved, 7 edge cubies are 6 moves away from solved, and the other 7 edge cubies 3 moves away from solved, the lower-bounds estimated number of moves to the solved state is

$$max(4,6,3)=6.$$

More details on implementation is available here.