A computer algorithm to solve the Rubik's Cube

Jahaan Rawat

Part 1

Consider a 3x3 Rubik's Cube. There are $43,252,003,274,489,856,000$ possible combinations, and only 1 of them is considered a solution. Many computer scientists and mathematicians were interested in this simple yet NP-COMPLETE puzzle and found out the God's number, which is the minimum number of moves required to solve the cube, to be 20 moves. This project will be an attempt to find solution for a scrambled 3x3 Rubik's Cube in polynomial time.

The ADT is derived from Trie and Tree. The main purpose is to explore different paths from root to leaves, and find the path where the value of the leaf is true. The depth of ADT is predefined by the user. Each parent and child has a Boolean. The branch between parent and child is a *string* defined by the user. There are 2 main operations: $findpath()$ returns the path from root to child which is true, and $addnext(L)$ inserts the children for L, a leaf in the structure.

Let the Rubik's Cube engine have methods for R, U, L, D, F, B, MV, MH, R', U', L', D', F', B', MV', and MH'. Also, it is assumed that a 3x3 Rubik's Cube engine takes $\mathcal{O}(1)$ time for each turn. The ADT will start from a completely scrambled Rubik's Cube. The branching represents the move performed and the children of each parent represent the state after the move was performed. The Boolean stored in each child checks if the Rubik's cube is solved. The children of the root represents the state after all possible moves, and for the subsequent branching consider the following rules to avoid same states of Rubik's Cube deep in branching:

- No move will be performed 3 times in a row, for example the branch R, R, R can't exist since it is the same as R' because the state of the Rubik's Cube will be the same.
- Moves performed 2 times in a row will be chosen at random, for example R, R and R', R' will branch randomly because the state of the Rubik's Cube after the moves will be the same.
- After performing a move its inverted move will not be performed. For example After R branch, R' can not exist.

Part 2

NOTE: For both implementations if the move does not exist Null/None will be stored, if the move does not solve the Rubik's Cube False will be stored, and if the move solves the Rubik's Cube True will be stored.

Implementation derived from Trie.
Let a trie structure exist with the following changes:

1. The branches labeled left to right R, U, L, D, F, B, MV, MH, R', U', L', D', F', B', MV', and MH' are pointers to the nodes. All nodes have parent pointer except the root.
2. The rules for subsequent branching are in place from part 1.
3. Each node represents the state of the Rubik's cube after the moves were performed. The hash table stores the label and the Boolean. The first set of keys is the current depth and the second set of keys are the labels which holds the value Boolean. The hashtable is perfect becasue of the uniqueness of the sequence of moves.
4. $findpath()$ and $addnext(L)$ are the operations. $findpath()$ returns the required label from hash table, and $addnext(L)$ helps in building the data structure after instantiation.
5. Depth is predetermined.

$findpath()$
run while loop i $\leftarrow 1$, $i++$
Perform binary search on $(depth - i)$ key of the hashtable till a label with Boolean = True is found. Let that equal desiredLabel.
end while loop when $i = depth$ or Boolean = True
return desiredLabel

$addnext(L)$

if L has no parent then update all pointers and hashtbale accordingly
if L is the leaf after 2 repetition of any branch in [R, U, L, D, F, B, MV, MH, R', U', L', D', F', B', MV', MH'] then update all pointers except the same branch and the ' variation of the same branch, and the hashtable accordingly
if L is any another leaf in the Trie then update all pointers except the ' variation of the same branch and the hashtable accordingly
else L is not a leaf in the Trie

Implementation derived from Trees and using Rubik's Cube engine.

Consider Rubik's Cube Engine.
As described in part 1, the move methods are the operations performed in constant time and a Boolean that checks if it is solved in constant time.

Consider a Node.
Attributes: *item* which stores the Rubik's Cube engine, *children* an array of nodes and size 16, and *parent* which is Node or Null/None.

Consider a Tree.
Attributes: *root* which is the initial Node, *depth* which is defined by the user.

The Node will store the Rubik's Cube engine, where the initial state will be represented as the root in the Tree. The Tree will perform the operations $findpath()$, and $addnext()$ on the root to find the desired solution and add more children to the leaves. The indices of Node.children represents the moves R, U, L, D, F, B, MV, MH, R', U', L', D', F', B', MV', and MH' in ascending order.

$findpath()$
A modified DFS using Stack to find the node with the Boolean of the Rubik's Cube as True. Let that equal desiredNode.
Push indices from the desiredNode to its parent all the way till the root into empty stack A.
return stack A

$addnext(L)$
if L has no parent then update all value of array with new Nodes with item as the Rubik's Cube engine after the move was performed
if L is the leaf after 2 repetition of any branch in [R, U, L, D, F, B, MV, MH, R', U', L', D', F', B', MV', MH'] then update all indices except the same branch and the ' variation of the same branch with new Nodes with item as the Rubik's Cube engine after the move was performed
if L is any another leaf in the Trie then update all indices except the ' variation of the same branch with new Nodes with item as the Rubik's Cube engine after the move was performed
else L is not a leaf in the Trie

Part 3

NOTE: Majority of the analysis is done in terms of the depth D of the ADT.

Worst case analysis of $findpath()$ implementation derived from Trie

Size of the outer hastable is *depth* of the Trie.
Let the *depth* be D and $M = D - 1$
The cost of accessing items from hastable is $\mathcal{O}(1)$.
The cost of performing binary search on $Trie.hashtable[i]$ is $\mathcal{O}(\log(16^i))$.

$T(M) = \sum_{i=1}^{M} \log(16^i)$
$\Rightarrow T(M) = \log(1) + \log(16) + ... + \log(16^M)$
$\Rightarrow T(M) = 0 + \log(2^4) + ... + \log(2^{4M})$
$\Rightarrow T(M) = 0 + 4\log(2) + ... + 4\log(2^M)$
$\Rightarrow T(M) = 4(0 + \log(2) + ... + \log(2^M))$
$\Rightarrow T(M) = 4(0 + 1 + 2 + ... + m)$
$\Rightarrow T(M) = 4(\sum_{i=0}^{M} i)$
$\Rightarrow T(M) = 2((M)(M+1))$
$\Rightarrow T(M) = 2M^2 + 2M = 2(D-1)^2 + 2(D-1) \in \mathcal{O}(D^2)$

The worst case occurs when no solution is found, or when the Rubik's Cube is not perfectly scrambled such that the solution is not any of the leaves and no branching exists with the degree of permutation(https://ruwix.com/the-rubiks-cube/mathematics-of-the-rubiks-cube-permutation-group/) that returns to a solve state near the given *depth* D.

Average case analysis of $findpath()$ implementation derived from Trie

Let the total number of Nodes in the Trie be N. We know the worst case $T(n) = 2M^2 + 2M$ where M is defined as the *depth* $D-1$ of the Trie. Performing random moves repeatedly on the Rubik's Cube is a special case of Markov chain. Hence, the probability of any child as solution is $\frac{1}{\sum_{i=0}^{M} 16^i}$. We know that $N \approx \sum_{i=0}^{M} 16^i$. T_i is a random variable that shows if binary search was performed on the correct depth and the probability from above shows if the desired child was found.

$\mathbf{E}[T] = (\sum_{i=1}^{M} 2i^2 + 2i)(\frac{1}{\sum_{i=0}^{M} 16^i})$
$\Rightarrow \mathbf{E}[T] = (\sum_{i=1}^{M} i^2 + i)(\frac{2}{\sum_{i=0}^{M} 16^i})$
$\Rightarrow \mathbf{E}[T] = ((\sum_{i=1}^{M} i^2) + (\sum_{i=0}^{M} + i))(\frac{2}{\sum_{i=0}^{M} 16^i})$
$\Rightarrow \mathbf{E}[T] = ((\sum_{i=1}^{M} i^2) + (\sum_{i=0}^{M} + i))(\frac{2}{\sum_{i=0}^{M} 16^i})$
$\Rightarrow \mathbf{E}[T] = ((\frac{(M)(M+1)(2M+1)}{6}) + (\frac{(M)(M+1)}{2}))(\frac{2}{\sum_{i=0}^{M} 16^i})$

$\Rightarrow \mathbf{E}[T] = (\frac{2M^3+6M^2+4M}{3})(\frac{1}{\sum_{i=0}^{M} 16^i})$
$\Rightarrow \mathbf{E}[T] = (\frac{2M^3+6M^2+4M}{3})(\frac{15}{16^M-1})$
$\Rightarrow \mathbf{E}[T] = (\frac{10M^3+30M^2+20M}{2^{4M}-1}) \in \mathcal{O}(\frac{M^3}{N})$
$\Rightarrow \mathbf{E}[T] = (\frac{10(D-1)^3+30(D-1)^2+20(D-1)}{2^{4(D-1)}-1}) \in \mathcal{O}(\frac{D^3}{N})$

Amortized analysis of $findpath()$ of implementation derived from Tree

This Tree here can be considered a Graph number of vertices $V = \sum_{i=0}^{D} 16^i$. The vertices here are the nodes of the Tree and the edges are the indices of the array of size 16 which stores the children nodes. Note the cost for accessing the Rubik's Cube Engine is 1 and the cost for accessing a single index of the children array is 1, hence the cost of a single traversal is 2. There is a single path from root to the leaf.
Let the total cost of traversing $V = \sum_{i=0}^{D} 16^i$ nodes be $T_1(V) = 2(V)$.

Let the cost of accessing the index of child node is 15. There are $D-1$ indices required, hence total cost for $D-1$ be $T_2(D) = 15D - 15$.

The cost of both *push* is 1. Hence total required cost from push and pop performed $D-1$ times is $T_3(D) = D - 1$

The total number of operations is $(\frac{16^D-1}{15}) + 2D - 2$

Let the total cost of the function be $T(V, D) = 2(V) + 16D - 16$
$\Rightarrow T(D) = 2(\sum_{i=0}^{D} 16^i) + 16D - 16$
$\Rightarrow T(D) = 2(\frac{16^D-1}{15}) + 16D - 16$
The amortized cost is $\frac{2(\frac{16^D-1}{15})+16D-16}{(\frac{16^D-1}{15})+2D-2}$

Worst case analysis of $findpath()$ of implementation derived from Tree

The worst case occurs when there is no solved state, or the right most leaf in Trie is the solved state of the Rubik's Cube since in this implementation the DFS occurs from the left and gradually moves to the right.

Using the total cost from the Amortized Analysis done above the total worst case cost would be $T = 2(V) + 16D - 16$

$\Rightarrow T = 2(\sum_{i=0}^{D} 16^i) + 16D - 16$
$\Rightarrow T = 2(\frac{16^D-1}{15}) + 16D - 16 \leq 16^D + 16D$
Hence worst case complexity is in $\mathcal{O}(16^D + D) = \mathcal{O}(16^D)$

Remarks on the performance analysis
It seems that the implementation from the Trie may find the solution in polynomial time depending on the Depth while implementation from Tree takes exponential. Moreover, the analysis is an estimation and may not be the perfect analysis.

Best case and worst case for Trie implementation

- $findpath()$ $\mathcal{O}(1)$ because best case of binary search is $\mathcal{O}(1)$ and the string of moves label would be at depth $D-1$ of the hashtable. For worst case refer the analysis.
- $addnext(L)$ $\Theta(1)$ is trivial.

Best case and worst case for Tree implementation

- $findpath()$ $\mathcal{O}(D)$ because it would require DFS till the first leaf, and *pop* and *push* for $D-1$ items would be $\mathcal{O}(D)$ as well. For worst case refer the analysis.
- $addnext(L)$ $\Theta(1)$ is trivial.