Solving Rubik's Cube Using Graph Theory: ICCI-2017

Chapter *in* Advances in Intelligent Systems and Computing · January 2019

DOI: 10.1007/978-981-13-1132-1_24

6 authors, including:

Sandeep Sambhaji Udmale
Veermata Jijabai Technological Institute
44 PUBLICATIONS **729** CITATIONS
SEE PROFILE

Vijay Sambhe
Veermata Jijabai Technological Institute
26 PUBLICATIONS **140** CITATIONS
SEE PROFILE

Solving Rubik's Cube Using Graph Theory

Chanchal Khemani, Jay Doshi, Juhi Duseja, Krapi Shah, Sandeep Udmale and Vijay Sambhe

Abstract The most common application of graph theory is search problems. Using graph theory, this project aims to solve one such NP-hard problem, i.e., finding a path for a Rubik's cube to reach the solved state from a scrambled one. Rubik's cube is among one of the fascinating puzzles and solving them has been a challenge given its vast search space of 43 quintillion. This paper aims at demonstrating the application and performance of traditional search algorithms like breadth-first search, depth-limited search, and bidirectional search, and proposes a new approach to find the solution by integrating them. The proposed algorithm makes use of the fact that the God's number for a $3\times3\times3$ Rubik's cube is 20, i.e., the fact that any cube scramble within the 43 quintillion states can be solved within a max of 20 moves.

Keywords Bidirectional search · Breadth-first search · Depth-limited search
God's number · Rubik's cube

C. Khemani (✉) · J. Doshi · J. Duseja · K. Shah (✉) · S. Udmale · V. Sambhe
Veermata Jijabai Technological Institute, Mumbai, India
e-mail: chanchal.khemani21@gmail.com

J. Doshi
e-mail: jdoshi1995@gmail.com

J. Duseja
e-mail: dusejajuhi@gmail.com

K. Shah
e-mail: shahkrapi12@gmail.com

S. Udmale
e-mail: ssudmale@vjti.org.in

V. Sambhe
e-mail: vksambhe@vjti.org.in

N. K. Verma and A. K. Ghosh (eds.), *Computational Intelligence: Theories, Applications and Future Directions—Volume I*, Advances in Intelligent Systems and Computing 798, https://doi.org/10.1007/978-981-13-1132-1_24

1 Introduction

The 3×3×3 Rubik's cube, often referred to as the magic cube is a 3-D puzzle which has intrigued adults and children alike. This puzzle, small yet difficult to solve, has about 43 quintillion combinations in its state space, i.e., there are about 43 quintillion possible ways to scramble a cube. If you were to turn a Rubik's cube once every second, it would take you 1.4 trillion years to go through all the combinations. As intimidating as these numbers may seem, the maximum number of half turns required to solve any scrambled cube is just 20 (using the half turn metric) and that is what is known as the **God's Number** [1–3].

This paper puts forth an algorithm which gives solution of any given scrambled cube within the limits of the God's number.

2 Literature Review

This section discusses the lower and upper bounds suggested of Rubik's cube algorithm.

2.1 Lower Bounds

By 1980, with the help of counting arguments, it was understood that there exist positions which require at least 18 [2, 4, 5] moves to solve, but no positions were yet proved to require more moves. In the counting argument, first the number of cube positions that exist in total is counted and then the number of positions achievable using at most 17 [2, 4–6] moves is counted. These two countings give the result that the latter number is smaller.

Later, in 1995, it was proved that the **superflip position** (all corners solved and all edges flipped in their home positions) requires 20 moves to solve which increased the lower bound to 20 [2, 4–6].

2.2 Upper Bounds

The first upper bounds [7] were based on the "human" algorithms. The upper bound was found to be around 100 by combining the worst-case scenarios for each part of these algorithms.

In 1979, David Singmaster simply counted the maximum number of moves required by his cube-solving algorithm and gave an upper bound of 277. Later, Elwyn Berlekamp, John Conway, and Richard Guy proposed an algorithm that took at most 160 moves. Following this, Conway's Cambridge Cubists algorithm could solve the cube in at most 94 moves [1].

In July 1981, Thistlethwaite proved that 52 moves suffice. This bound was lowered to 42 in 1990 by Kloosterman. In 1992, this bound came down to 39 by Reid, and then to 37 by Winter. In 1992, Kociemba introduced his two-phase algorithm. This algorithm had not very high memory requirements and it gave near-optimal solutions in a short time.

In 1995, upper bound was cut down to 29 by Reid. In 1997, Korf proposed the algorithm for optimally solving arbitrary positions in the Rubik's cube group. The runtime of this algorithm was about a day. In 2005, Radu lowered the upper bound to 28. In 2006, it was further lowered to 27. In 2007, Kunkle and Cooperman used computer clusters to lower the bound to 26. In March 2008, Tomas Rokicki cuts the lower the bound to 25 [4] and then 23 and 22 [2].

In July 2010, Tomas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge proved that God's number for the cube is exactly 20 [5, 8, 9]. They used iterative-deepening-A* (IDA*) algorithm with a lower bound admissible heuristic function based on pattern databases. These pattern databases consist of tables which store the exact number of moves required to solve the cube upto a particular level to make the processing faster by precomputing some results. The algorithm developed can solve one billion positions per second, using 35 CPU years on Google server.

3 Methodology

The Rubik's cube can be modeled as a graph problem where the nodes are a set of all possible configurations and the arcs represent the connection between two configurations that are one twist away from each other. The primary movement on any $3\times3\times1$ subcube of the cube is 90° and 180° rotation in the clockwise and anticlockwise directions. The 180° move in clockwise or anticlockwise direction will result in the same configuration of the cube. Thus, there are three primary movements on each face of the cube, 90° turn in clockwise direction known as F, 180° turn known as F2, and 90° turn in anticlockwise direction F′. Each node of the tree represents a cube configuration. As seen above, any cube face has three possible moves: F, F2, and F′. Considering six faces of a cube and three possible moves for each, each node of the tree will give rise to 18 branches of its own. This problem mapping can be used to find solutions for solving the Rubik's cube.

3.1 Nodes in a Brute Force Approach

Consider a depth of zero at the root node which will contain the scrambled cube. There are six faces of a cube and three possible moves for each face, i.e., F, F2, and F' giving a total of 18 different possible combinations. Hence, number of nodes at depth 1:18.

Each node at depth 1 will now have 15 different combinations possible (Not considering the turns which result in the same node configuration as the one in depth 1).
$\therefore$ at depth 2, total number of nodes present will be $15 \times 18 = 270$ nodes.

Similarly, for depth 3, the number of nodes will be $15^2 \times 18 = 450$ nodes.

$\therefore$ at the depth 20, which is considered to be the God's number, the total number of nodes will be $15^{19} \times 18 = 3.99 \times 10^{23}$.

In general, total number of nodes at any depth will be given by

$$15^{(d-1)} \times 18, \tag{1}$$

where d is the depth of the tree, with depth 0 having 1 node only—the scrambled cube.

Since God's number is 20, it can be claimed that any specific configuration or scramble is at a maximum distance of 20 from the root. It has been proven that the entire state space of cube contains 43 quintillion distinct or unique cubes. Number of nodes generated till level 20:

$$1 + \sum_{n=1}^{20} 18 \times 15^{(n-1)} = 4.275 \times 10^{23}.$$

$\therefore$ this approach produces many more nodes. The repetition factor is

$$\frac{4.275 \times 10^{23}}{4.3 \times 10^{19}} = 9942.628.$$

For any cube, at the first step, i.e., level 0, there are $6 \times 3 = 18$ unique possibilities. At the next level, there are 15 nodes produced as the three moves of the face on the parent node are not considered.

3.2 Breadth-First Search and Depth-Limited Search

BFS allows to search for all the solutions at a particular level, and hence guarantees a shortest path to the solution. Nodes generated in BFS at a particular level d are given by (1).
$\therefore$ Nodes in memory at depth 5: $18 \times 15^4 = 911{,}250$.

BFS continues till a depth of *limitBFS*, i.e., depth 5 and after that DFS is employed upto a certain limit (*limitDFS*), i.e., Depth-Limited Search (DLS).

DLS searches for a solution depth-wise unlike BFS. The memory requirements compared to BFS are considerably less, and hence this approach is used. The number of nodes generated by level 5 BFS is 911,250 which are the roots for DLS. On a

machine with 20 cores and 32 GB memory, 19 threads are used to utilize the cores in order to perform DLS in parallel. The nodes of level 5 are generated into 19 groups. The first 18 groups have 47,960 nodes, and the 19th group has 47,970 nodes. Each group is taken up by a thread which treats these nodes as roots and does DLS traversal [10, 11].

3.3 HashMap

The depth of the back tree is represented by *limitHash*, and the root node is the solved cube. These nodes are stored in a HashMap where the key is the cost of misplaced facelets in the cube structure a.k.a. cost. The HashMap stores all the nodes from level 0 to level *limitHash* so that all the solutions within the range *limitDFS* + *limitHash* are found. For instance, if *limitDFS* is 5 and *limitHash* is 3, then the solutions at any of the levels 6, 7, and 8 must be reachable. Nodes in the back tree are generated using BFS for the following reasons.

- Time required for traversal using BFS up to depth 5 is which is faster than DFS.
- The memory required by BFS till depth 5 is 125.14M bytes which is within the bounds.

Total nodes in HashMap (for level = 5):

$$1 + 18 \times 15^0 + 18 \times 15^1 + 18 \times 15^2 + 18 \times 15^3 + 18 \times 15^4 = 976{,}339.$$

3.4 Bidirectional Search

Now consider two trees, the front tree which has the scrambled cube as the root node and the other is the back tree, with a solved cube as the root.

The back tree is first expanded to give 18 nodes owing to the 18 possible moves. These 18 nodes are compared with the root of the front tree since it has not expanded yet. If a match is found, the solution is found and the inverse of the move of the matching node gives the solution to the cube. However, if a match is not found, the front tree is expanded and compared with nodes at depth 1 of the back tree. If the match is found, it means that we have reached the solution. Thus, the moves required to solve the cube are given by the moves of the front tree and inverse moves on the back tree. If the match is not found, the back tree is then expanded and a comparison is done between the expanded nodes of the back tree and the existing nodes of the front tree. In this way, the comparison continues for each new depth expanded.

Considering a bidirectional search tree, the maximum depth of back tree is 10 and that of front tree will be a depth of 10.

Thus, the maximum total number of nodes become

$$15^9 \times 18 + 15^9 \times 18 = 1.383 \times 10^{12} \text{ nodes.}$$

Memory requirement :

$$M \times 1.383 \times 10^{12} \text{ bytes.}$$

This is much less as compared to $M \times 3.99 \times 10^{23}$ bytes required for BFS approach.

4 Mathematical Proof

We have used BFS in our approach for its completeness and optimal property. Let us prove that BFS finds the smallest possible solution for any input scramble. First, let us have some notations for clarity.

Consider there are n vertices, *numbered 1 to n* in the graph created for by considering input cube as root up to the *limitBFS* specified. The BFS starts at vertex r denoting the input scramble, which forms the root of the BFS front tree, and a total of l vertices are reachable from r (and hence processed during breadth-first search).

Now, for a vertex v, we define

- $dist[v]$ to be the minimal distance from root r to any vertex v in the tree,
- $level[v]$ to be the level of any vertex v in the tree,
- $pos[v]$ to be the position number p, where $1 \leq p \leq l$, and v was the pth vertex to be inserted into the queue during breadth-first search traversal.

We wish to prove that for any vertex v, *level[v] = dist[v]*. We use the method of induction for $pos[v]$ for all v to achieve this. Thus, we prove by induction on $p = 1$ to l that for v with $pos[v] = p$, we have

(A) dist[v] = level[v]

(B) For any vertex w, if dist[w] < dist[v], then pos[w] < pos[v].

Now, for the case $p = 1$, it is obviously true.

We must now prove (A) and (B) for all p, where $1 < p \leq l$, assuming (A) and (B) hold for all $p^0 < p$.

To prove (A) for any p, let $pos[v] = p$, and let v be the child of v^0 in the breadth-first search tree.

Using contradiction, let us suppose that there is a path having length $len < level[v]$ for, $r \sim w \rightarrow v$ (Denotes a path from r to v, through w)

We have

$$pos[v^0] < pos[v]. \tag{2}$$

Now, in BFS, we know that a child is placed in the queue only when the parent is removed from it. So, we can say v is placed in the queue only when v^0 is dequeued from it.

So we have

$$dist[w] < dist[v^0]. \tag{3}$$

Let us apply the induction hypothesis (A) at $pos[v^0]$ using (2). Thus, we obtain

$$dist[v^0] = level[v^0] = level[v] - 1 > len - 1 \geq dist[w^0].$$

Using (2), we can apply hypothesis (B) for $pos[v^0]$, and together with (3), we obtain (4) as

$$pos[w] < pos[v^0]. \tag{4}$$

But now let us consider any time instant during the execution of breadth-first search when the node w was dequeued from the queue.
Since there is an edge $w \rightarrow v$, the breadth-first search traversal would currently visit v, if it already has not earlier. Thus, the parent of v has p at max $pos[w]$. But, using (4) it has to be strictly less than $pos[v]$, and hence, v^0 cannot be the parent of v, as we had earlier assumed. Thus, it is a contradiction.

Let us now prove (B). Suppose,

$$\begin{aligned} pos[v] &= p, \\ dist[w] &< dist[v]. \end{aligned} \tag{5}$$

Using contradiction, let us assume that $pos[w] \geq pos[v]$. By (5), we can surely say that $w \neq v$, and thus, $pos[w] \neq pos[v]$. Hence, we get

$$pos[w] > pos[v]. \tag{6}$$

Now, let v^0 be the parent of vertex v and let $r \sim w^0 \rightarrow w$ (Denotes a path from r to w, through w^0) be the shortest path from root r to vertex w. This implies that the path $r \rightarrow w^0$ is the shortest of any of the paths from r to w^0. Thus, we can conclude

$$dist[w^0] = dist[w] - 1. \tag{7}$$

We have

$$pos[v^0] < pos[v]. \tag{8}$$

Also, by (A) at $pos[v]$, which was proved above, the path $r \sim v^0 \rightarrow v$ is a shortest path, and therefore the tree path $r \sim v^0$ also has to be the shortest path. Thus, we may conclude

$$dist[v^0] = dist[v] - 1. \tag{9}$$

Now, using (5), (7) and (9), we get

$$dist[w^0] < dist[v^0]. \tag{10}$$

Using the induction hypothesis (B) and using (8), at $pos[v^0]$, which according to (8) is less than $pos[v]$, we can say

$$pos[w^0] < pos[v^0]. \tag{11}$$

Now, vertex v was queued only when v^0 was removed, and vertex w entered the queue when either w^0 was removed or at some previous time. Thus, by using (11), we know that w would enter the queue prior to v. This contradicts (6). Hence, we have proved that the breadth-first search traversal would always find the most optimal solution for any input scramble.

So as the God's number is 20, we can say that using an exhaustive approach such as BFS, we can find the solution definitely within 20 moves if and only if we can curtail the exponential growth in memory effectively. Now let us verify using an example (Refer Fig. 1a).

Let G be the goal state. Now,

$$\begin{aligned} &\therefore pos[G] = 7 \\ &\because parent(G) = D \\ &\therefore pos[D] = 4. \end{aligned} \tag{12}$$

Here, the bold edges represent the shortest path and the arrows represent the actual path. Consider, for hypothesis (A), $dist[G] = level[G]$ (since level and distance both are 2). Thus, hypothesis (A) is satisfied.

Consider for hypothesis (B), we have $dist[D] < dist[G]$. Thus, we must also have $pos[D] < pos[G]$. This is true from (12). Here, we can say that for no node in the search space the hypothesis (A) or (B) will be violated. Thus, breadth-first search

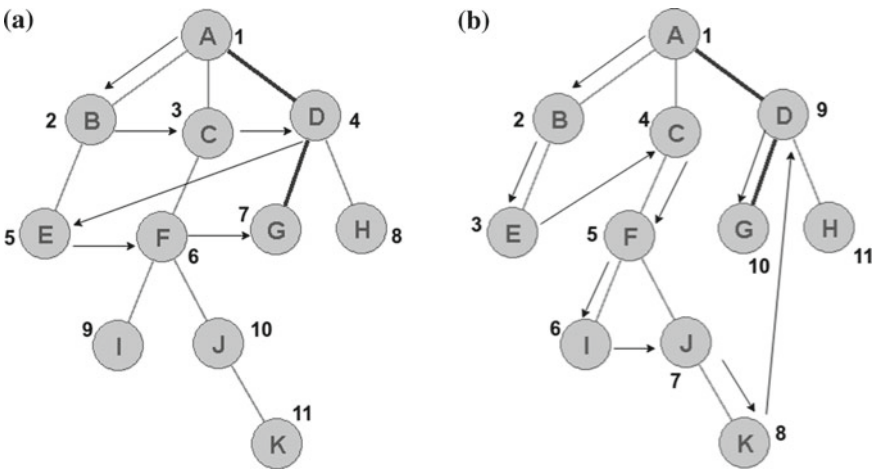

Fig. 1 **a** Breadth-first search. **b** Depth-first search

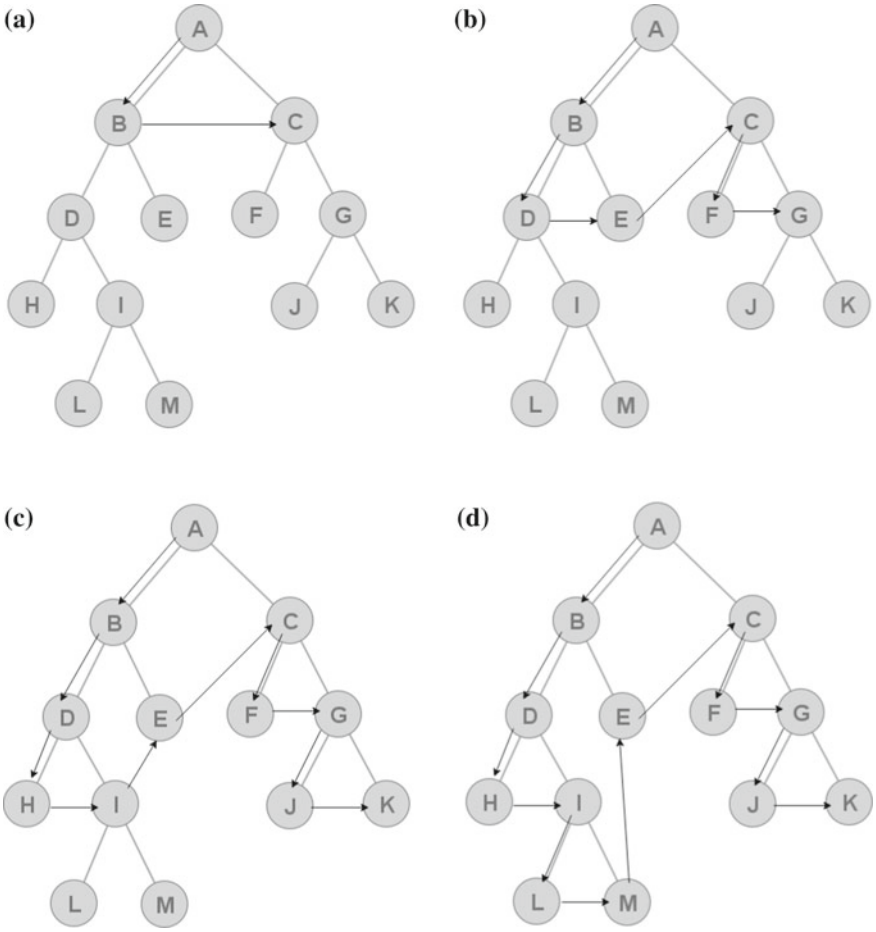

Fig. 2 Depth-limited search traversal. **a** Depth bound 1. **b** Depth bound 2. **c** Depth bound 3. **d** Depth bound 4

will definitely find the solution (if it exists) by traversing all the nodes up to the point of solution.

After the BFS traversal is completed, the DFS traversal of the search space starts by considering each node of BFS at level *limitBFS* as root. Now let us consider the same hypothesis for depth-first search traversal (Refer Fig. 1b).

Here, the bold edges represent the shortest path and the arrows represent the actual path. Let G be the goal state. Thus, (12) is satisfied here as well. Consider, for hypothesis (A), $dist[G] = level[G]$ (since level and distance both are 2). Thus, hypothesis (A) is satisfied.

Consider for hypothesis (B), we have $dist[G] = 2 < dist[J] = 3$. Thus, we must also have $pos[G] < pos[J]$, but this is not true. From Fig. 1b, we can verify that for depth-first search, $pos[G] = 10 > pos[J] = 7$.

Thus, we can say that depth-first search traversal, even though finds the correct solution, is not optimal as the hypothesis (A) or (B) will be violated.

So, let us now justify why using the depth-first search is important even though it is incomplete and not optimal. The main advantage of depth-first search is that its space complexity is $O(b \times m)$, where b is the branching factor and m is the limit/depth. This is a big advantage as we are dealing with a search space of the order of 10^{19}. But in order to use it we need our implementation to make sure that the completeness property remains. The main drawback of DFS is its tendency to get stuck in infinite loops. Thus, we use a variant of depth-first search known as depth-limited search and ensure that traversal may not get lost in parts of the search space that have no goal state and never return. Consider Fig. 2.

This shows that the traversal is always within the specified bound or limit and hence we make use of depth-limited search in our implementation.

5 Implementation

5.1 Working for BFS + DLS Approach

This approach taken to solve the Rubik's cube can be said to consist of the following searches together forming the algorithm (Refer Fig. 3a, b):

- Breadth-first search,
- Parallel depth-limited search, and
- Matching phase.

- Given the input scramble, the algorithm will first perform breadth-first search up to a certain limit *limitBFS*.
- At each expansion of node, a cost of the cube generated is calculated. A cost of zero indicates that the cube is solved. If a cube with zero cost is found within the limits of BFS, it indicates that solution is found.
- Concurrently, the solved cube is expanded up to a *limitHash* level and a HashMap is created using cost of each cube as the key. This HashMap created is used during the matching phase.
- If the solution is not found in *limitBFS*, then a parallel DLS is used. Making use of concurrency, various nodes are selected from the *limitBFS* level and concurrently various nodes are expanded up to the limit *limitDFS* as set.
- If the DLS also does not find a solution within the *limitDFS* set, all nodes in the last level, i.e., the *limitDFS* level is compared with those in the HashMap. The cost of each node is taken as key and all possible nodes found in the HashMap are compared.

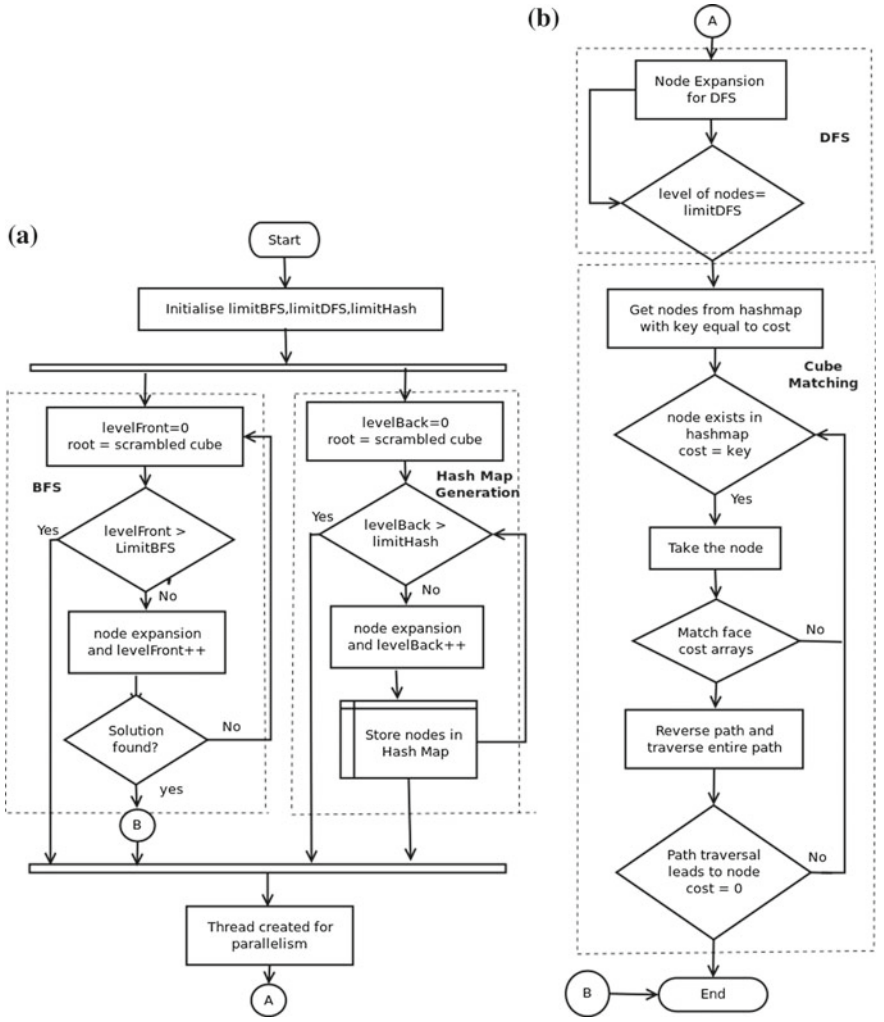

Fig. 3 **a** Implementation flow-part 1. **b** Implementation flow-part 2

– For each of the nodes matching the node cost, all the face costs are matched. If a match is found, then the reverse path of the node is traveled. This leads to the solved cube that is the one with cost zero.

Figure 4 shows an example of how the algorithm works for a scrambled cube whose solution can be obtained in five moves.

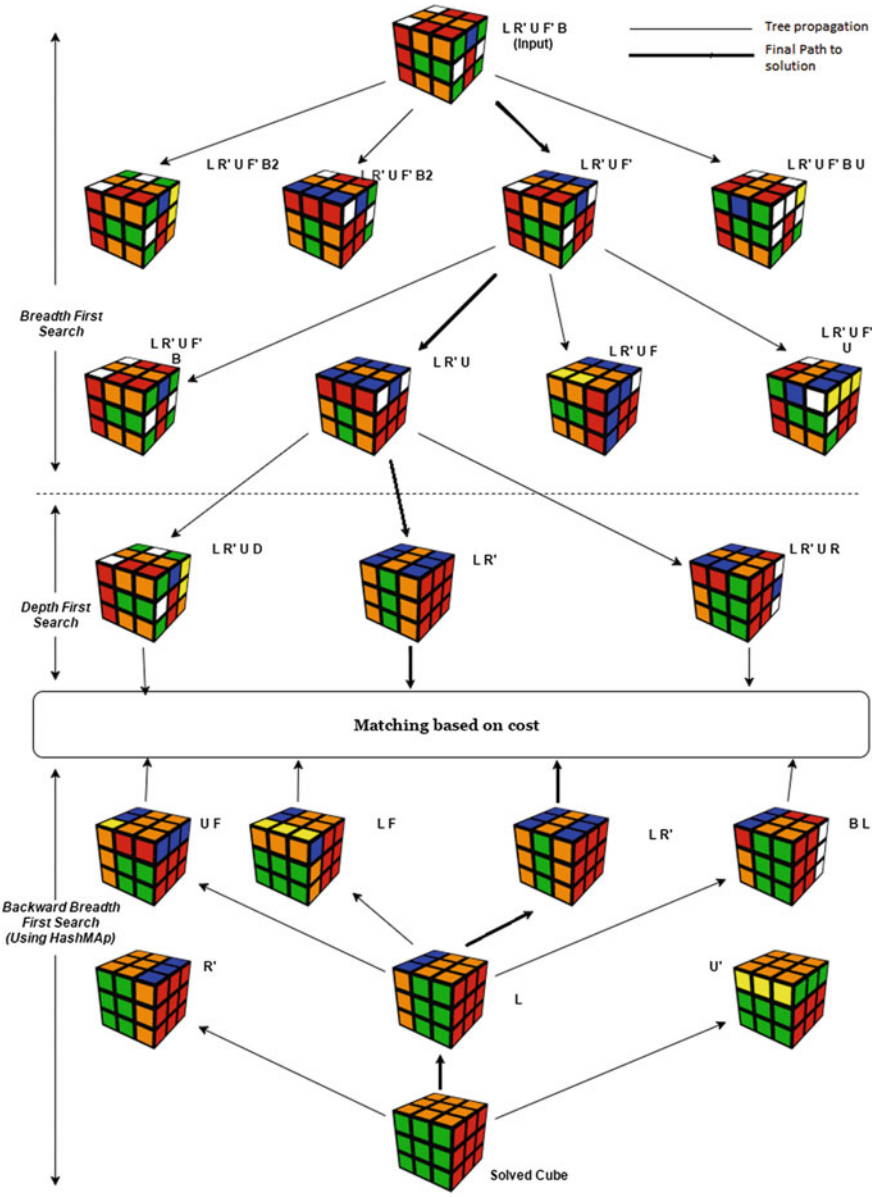

Fig. 4 Implementation flow example. Input scrambled cube: $LR'UF'B$

5.2 Matching Phase

We have created a HashMap by creating a back tree up to a limit *limitHash*. These nodes are stored in a HashMap where the key is the cost of misplaced facelets in the cube structure a.k.a cost. For any level l, the number of nodes t is the sum of all the nodes generated from root r to level l.

Thus, for level 3, we get total nodes t as

$$t = 1 + 18 + 18 \times 15 + 18 \times 15^2 = 4339.$$

For level 4, we get total nodes t as

$$t = 1 + 18 + 18 \times 15 + 18 \times 15^2 + 18 \times 15^3 = 65{,}089.$$

Thus, for level 5, we get total nodes t as

$$t = 1 + 18 + 18 \times 15 + 18 \times 15^2 + 18 \times 15^3 + 18 \times 154 = 976{,}339.$$

Thus, for level 6, we get total nodes t as

$$t = 1 + 18 + 18 \times 15 + 18 \times 15^2 + 18 \times 15^3 + 18 \times 15^4 + 18 \times 15^5 = 14{,}645{,}089.$$

On observing the numbers, we can clearly state that they follow a geometric progression. Now, we know that the total number of nodes for any geometric progression can be found using the expression:

$$1 + \sum_{k=1}^{l} a \cdot r^{(k-1)} = 1 + a \times \frac{r^l - 1}{r - 1}. \tag{13}$$

Here $a = 18$, $r = 15$.

$$\begin{aligned} \therefore 1 + \sum_{k=1}^{l} a \cdot r^{(k-1)} &= 1 + \sum_{k=1}^{l} 18 \times 15^{(k-1)} \\ &= 1 + 18 \times \frac{15^l - 1}{15 - 1}. \end{aligned} \tag{14}$$

Thus, for any level l, we can find the total number of nodes in the HashMap using the above formula.

For level $l = 5$ from (14), we get total nodes $t = 65{,}089$. Now consider that a vertex v having cost c enters the matching phase, so now it traverses through all the 65,089 nodes to find nodes which have their costs as cost c. This is not efficient as the space and time complexities are too high.

Table 1 Nodes at each level in the HashMap

Cost of node (Key)	Level 3	Level 4	Level 5	Level 6
0	1	55	163	1027
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	528
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	96
9	0	0	0	384
10	0	0	0	384
11	0	0	0	0
12	72	336	2418	9162
13	0	0	192	480
14	24	72	432	4608
15	0	96	384	2016
16	144	264	2904	16,200
17	0	0	816	3504
18	48	144	2544	13,872
19	0	384	2208	18,048
20	24	984	4200	40,944
21	0	480	1776	28,464
22	192	1920	11,568	90,768
23	96	1152	8448	74,160
24	546	1746	15,558	138,564
25	48	1968	17,568	144,816
26	168	1656	19,440	191,088
27	48	1776	19,968	209,424
28	528	2784	34,704	333,408
29	96	2304	32,112	346,752
30	576	4176	48,114	517,680
31	672	2592	48,288	590,352
32	1056	9144	71,304	820,968
33	0	5184	61,104	861,024
34	0	5424	77,184	1,078,992
35	0	3456	66,576	1,089,552
36	0	5424	80,880	1,260,432
37	0	5472	88,656	1,299,840
38	0	3456	76,128	1,302,192

(continued)

Table 1 (continued)

Cost of node (Key)	Level 3	Level 4	Level 5	Level 6
39	0	864	63,360	1,138,032
40	0	1776	49,008	1,064,784
41	0	0	32,880	774,720
42	0	0	25,008	636,864
43	0	0	8208	312,096
44	0	0	2208	184,176
45	0	0	0	31,584
46	0	0	0	11,664
47	0	0	0	384
48	0	0	0	1056
Total:	4339	65,089	976,339	14,645,089

But, if we use a HashMap, then the vertex v must only traverse through a subset of total nodes t which have same cost as cost c. Table 1 shows the number of values across each cost as will be created in a HashMap.

6 Result

The program was run on a processor having 32 GB RAM with 20 processing cores. Using multiple threads to take advantage of multiple processors, we can achieve much higher efficiency. In our implementation, the generation of the breadth-first search tree and generation of HashMap are independent tasks and thus can be concurrently carried out. This doubles the speed up. Within the breadth-first search tree, traversal parallelism is difficult to achieve as the subtasks for BFS are dependent on each other. But for depth-limited search, all the threads can work concurrently on the list of nodes as there is no interdependency.

Speed up S is of two subcategories: [12]

- Speed Up in Latency ($S_{LATENCY}$).
- Speed Up in Throughput ($S_{THROUGHPUT}$).

Now, let $L1$ and $L2$ be the latencies of architectures without concurrency and with concurrency, respectively.

$$\therefore S_{LATENCY} = \frac{L1}{L2}. \tag{15}$$

And, now suppose $Q1, k1, A1$ and $Q2, k2, A2$ be the throughput, execution density (the number of stages in an instruction pipeline), and execution capacity (the number of processors for a parallel architecture) of architectures without concurrency and with concurrency, respectively.

$$\therefore S_{THROUGHPUT} = \frac{Q2}{Q1} = [\frac{k2 \times A2}{k1 \times A1}] \times S_{LATENCY}. \tag{16}$$

Now in our implementation the number of stages in the pipeline for both the architectures is the same, i.e., $k1 = k2 = 1$. Also, we have used 19 unique CPU cores in parallel. Thus, using this information and from (16), we get

$$\therefore S_{THROUGHPUT} = \frac{Q2}{Q1} = 19 \times S_{LATENCY}. \tag{17}$$

Now, suppose to find a solution for a scramble at level 7, we take 15 mins in architecture 1 and 4 mins in architecture 2, then according to (15),

$$S_{LATENCY} = \frac{15}{4}. \tag{18}$$

Referring to (17) and (18), we get

$$\therefore S_{THROUGHPUT} = 19 \times \frac{15}{4} = 71.25. \tag{19}$$

Thus, concurrent processing is very useful in our implementation. The earlier speed up is mainly for the DFS part, let us see the speed up for the entire system. Using Amdahl's law [7] for calculating the latency, let $S_{LATENCY(total)}$ be the latency of the entire system, p is the proportion of execution time that the part benefiting from improved resources originally occupied, and s is the speed up of that part.

$$\begin{aligned} S_{LATENCY(total)} &= \frac{1}{(1-p) + \frac{p}{s}} \\ &= \frac{1}{1 - 0.66 + \frac{0.66}{(\frac{15}{4})}} \\ &= \frac{1}{0.34 + 0.176} \\ &= 1.938. \end{aligned}$$

Thus, the speed up of the entire system has significantly improved.

References

1. Van Grol, R.: The quest for gods number. Math Horiz. **18**(2), 10–13 (2010)
2. Rokicki, T.: Twenty-two moves suffice for rubik's cube. Mathem. Intelligencer **32**(1), 33–40 (2010)
3. Gymrek, M., Li, J.: The mathematics of the Rubiks cube. http://web.mit.edu/sp.268/www/rubik.pdf (2009)

4. Rokicki, T.: Twenty-five moves suffice for Rubik's cube. arXiv preprint arXiv:0803.3435 (2008)
5. Rokicki, T., Kociemba, H., Davidson, M., Dethridge, J.: The diameter of the Rubik's cube group is twenty. SIAM Rev. **56**(4), 645–670 (2014)
6. Betsch, G.: Adventures in group theory: Rubiks cube, Merlins machine, & other mathematical toys. Mathem. Intelligencer **27**(2), 92–92 (2005)
7. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, Spring Joint Computer Conference. AFIPS '67 (Spring), pp. 483–485. ACM (1967)
8. Korf, R.E.: Finding optimal solutions to Rubik's cube using pattern databases. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence. AAAI'97/IAAI'97, pp. 700–705 (1997)
9. Rokicki, T., Kociemba, H., Davidson, M., Dethridge, J.: God's number is 20. http://www.cube20.org
10. Russel, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall (2002)
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. Tata McGraw-Hill (2002)
12. Martin, M., Roth, A.: Performance and Benchmarking. https://www.cis.upenn.edu/~milom/cis501-Fall12/lectures/04_performance.pdf (2012)