

How To Solve a Rubik's Cube: Using Search Algorithms

Richard Y. Tao

Macleans College, 2 Macleans Road, Bucklands Beach, Auckland, 2012, New Zealand; richard82288@gmail.com

ABSTRACT: Breadth-first search (BFS) and depth-first search (DFS) are 2 of the most common methods of graph traversal to get from a starting node to a goal node. This paper explores how BFS, DFS, and a combination of the two—Iterative-deepening depth-first search (IDDFS)—compare when solving a Rubik's Cube. To investigate their differences in performance and correctness empirically, a linear representation of the cube, the moves (quarter-turns as functions), and the orders of instructions for each search algorithm were developed in code. I hypothesized that IDDFS would achieve the lowest runtime of the three algorithms while preserving the optimal solution length (H1). Furthermore, BFS and IDDFS would yield optimal solutions more reliably than DFS, measured by the success rate of solutions found by the algorithm (H2). IDDFS beats BFS and DFS in runtime, yielding optimal solutions in the fastest time. The results validate both H1 and H2.

KEYWORDS: Computer Science, Graph Traversal, Search Algorithms, BFS/DFS.

■ Introduction

The Rubik's Cube is the best-selling toy and puzzle of all time, so there has been an aura surrounding its challenge.¹ Traditional solving methods, such as CFOP (Cross, first 2 layers, orienting last layer, permuting last layer), Roux, ZZ, and Petrus, are widely used in the speedcubing world today, but these generate average solutions of around 50 moves.² The Fewest Moves Challenge (FMC) was introduced to the World Cube Association (WCA), the official governing organization for all competitive speedcubing competitions, in 2015, with the challenge being as stated in its name. Competitors have 1 hour, 3 cubes, a pen, and paper to find the shortest solution they can to a given scramble. God's number is the largest number of moves it would take to solve any cube state, assuming the optimal solution. It is known to be 20 moves.³ In a hobby and sport dominated by human interest, how would computers fare with modern-day search algorithms? Is there an optimal method for finding the shortest solution to any scramble? If there is, humans haven't managed to do it yet, but maybe computer search algorithms can. Korf's, Thistlewaite's, and Kociemba's algorithms have produced promising, near-optimal results.⁴

2-dimensional search trees, featuring a starting node, a goal node, and everything in between connected by branches, are also prevalent in the modern world. Google Maps gets you from point A to point B using heuristic functions dependent on factors such as traffic, distance, and time, which all factor into the heuristic cost and can be represented by a set of nodes and branches joining these nodes.⁵ In the context of Rubik's Cube solving, although classic properties of BFS, DFS, and IDDFS are well known, their empirical trade-offs on short scrambles remain under-quantified. Accordingly, we test two hypotheses:

H1: For short scrambles (≤ 7 moves), IDDFS exhibits lower median runtime than BFS and DFS while returning optimal-length solutions

H2: BFS and IDDFS return optimal solutions at a higher rate than DFS across depths 1–7

The roots of this topic come from finding God's number for the Rubik's Cube, where an upper and lower bound were squeezed through a 3-decade process until Tomas Rokicki *et al.* used Google's supercomputers to find the number to be 20.³ Herbert Kociemba contributed to the process through Kociemba's algorithm, which essentially identifies a subset of states that can all be solved in 18 moves.⁶ Every scramble can undergo a series of moves to become a state of this subset, which can subsequently be solved.⁷ Although God's number is 20 and not 18, the most common optimal solution length to any random scramble is 18 moves,⁸ proving Kociemba's algorithm's usefulness. Kociemba's algorithm is near but not optimal,⁶ but by a combination of a known subset and considering heuristics, it can always find a near-optimal solution in a reasonable amount of time.⁹

Korf's algorithm can find the shortest solution to any Rubik's Cube scramble, but it is very computationally intensive and requires significant amounts of resources and time to execute. Subsequent research attempts have either employed new techniques to lessen the run time by about 10%,⁶ or depended on heuristics (IDA*) and known subsets to balance time and cost against finding a near-optimal solution.

My research employs 3 fundamental search algorithms – BFS, DFS, and IDDFS – to solve the cube and analyze their inherent differences in relation to the topic. These 3 methods have been chosen logically: they are the fundamental basis of most search algorithms. Despite many papers detailing the methods used to solve a cube using different search algorithms, there does not seem to be an existing direct comparison between the accuracy and runtime of these methods. Although not done at the higher move counts, which are closer to a true scramble length, due to lack of memory, the pros and flaws of each search algorithm can be easily seen for scrambles less than 8 moves, and also compared against the theoretical search space. Something which I have done that may not have been reflected on as much in other previous literature is tracking the

scrambles used and solutions produced, as paired with certain move orders; this may produce interesting results for the different search algorithms. The comparison of these 3 different search algorithms in combination with certain scrambles can help us to better understand where and when to apply these fundamental methods when solving the Rubik's Cube computationally.

■ Methods

The first step is representing the cube. A cube net was used, with a complete list of 54 numbers representing each facelet on the puzzle. There are 6 faces on a Rubik's Cube, hence 12 unique 90° turns (e.g., U for 'up' face clockwise, and U' for 'up' face counterclockwise). Each of the 12 possible quarter turns was then set as a function, each shifting 20 facelets in a certain way (1 turn moves 21 facelets, less the centerpiece, which is unaffected). In a graph with nodes representing the states of the cube and edges connecting these states via quarter-turn moves, BFS is a sweeping algorithm, exploring every possible move at each depth, then moving on to the next. This ensures the shortest solution is found each time due to the algorithm's completeness. With big O notation, where $O(\cdot)$ represents the most amount of time required in the worst case scenario to reach the goal node, the time complexity (big O) of BFS is $O(bd)$, where b represents the branching factor of our graph (how many children from each node, in this case, 12), and d being the depth of the goal node; the space complexity is $O(bd)$ as every node explored at each level is kept in memory.¹⁰ BFS employs a FIFO (first in, first out) queue, meaning each depth is processed from left to right in a designated move order, guaranteeing the shortest solution each time. It should also be noted that the general order of action for each algorithm is: check if goal state → explore next node → repeat, in case the initial state is indeed already the goal state.

DFS employs LIFO (last in, first out) with a stack, adding to the stack by exploring the deepest paths first, then returning to the last depth level after hitting a dead end. It has a time complexity of $O(bd)$ and space complexity of $O(bd)$.¹¹ This space complexity is much less than BFS, increasing linearly rather than exponentially with depth, as DFS only remembers the path it took to get to a node, along with all the daughters of the nodes along that path. Immediate inverse moves are pruned trivially, thus saving time visiting unnecessary paths and preventing endless loops. For experimental reasons, the depth limit was set to 7, meaning that our DFS algorithm would only return solutions for scrambles less than or equal to 7 moves in length. A higher depth limit would also likely increase solve times for lower move counts, so it is only fair to set it to be the same throughout the entire experiment.

IDDFS is a combination of BFS up to a certain depth, but expanding its depth limit by 1 after each level, as the name suggests. It is effectively DFS being iterated at increasing depths (as the name suggests). The time complexity of IDDFS is the same as for BFS and DFS individually ($O(bd)$), as in the worst-case scenario, every node will have to be visited before finding the goal node. Immediate inverse moves are pruned just like in DFS, which plays out much better for time: due

to IDDFS's complete, breadth-first character, the branching factor is reduced from 12 to 11, greatly reducing its realistic search space. Its space complexity is the same as DFS ($O(bd)$) due to the algorithm's intrinsic depth-first nature. Effectively, IDDFS is an extension of DFS, which guarantees the completeness of BFS and the ability to find the optimal solution each time while maintaining the superior space complexity of DFS, making it theoretically the ideal search algorithm for solving the Rubik's Cube in our case.

A fixed moveset is also specified for every search algorithm to define the search space and thus the order of execution with FIFO and LIFO queues. The move set is U, U', D, D', L, L', R, R', F, F', B, B'. It is arbitrary, but it may create varying search times for the greedy DFS depending on what moves are in the solution.

Here is the link to the code, which is contained in the GitHub repository:

https://github.com/richard82288-del/rubiks_cube_search_algo

■ Results and Discussion

Results:

https://docs.google.com/spreadsheets/d/1iqReiS-QxsAv-I5cNupY_3d03f4BrLx1C4o7KK-S7tzc/edit?usp=sharing

All 3 algorithms employed in this experiment are uniform-cost (non-cost-associated), as each move on a Rubik's Cube objectively gives an equal probability of reaching the goal state when the solution is unknown. While BFS gives the optimal solution each time as expected, DFS sometimes adds unnecessary moves to the solution, highlighting its greed for depth. This often results in an equivalent 'U3' instead of 'U', or adding an 'F', 'F''; as its search order prioritizes clockwise moves before anticlockwise moves, DFS occasionally will add 2 moves to otherwise optimal solutions even in the <6 move range. However, given lucky move sets often involving sequences of U's, DFS may occasionally be faster than expected, leading to a rather unpredictable time-to-move ratio, although rarely outperforming BFS or IDDFS in time or solution length (solution length is rather redundant when analyzing shorter scrambles, as there is usually only 1 solution from an algorithmic perspective).

BFS crashes the system at 7 move scrambles, as my 8GB of RAM is not enough to sustain its large but complete space complexity. Its requirement for great operational capacity or memory, especially for a problem such as the Rubik's Cube, which has a branching factor of 12 (in our case, ignoring double moves), makes it unfeasible for slightly longer and more realistic scrambles.

The DFS stack significantly mitigates BFS's memory problem, narrowing down its path to a solution. This does, however, often lead DFS down the wrong path, often taking longer to find depth 1 solutions than for higher depth scrambles. Its inability to accept a dead end on a puzzle with no such thing as the Rubik's Cube highlights its weakness with greed for depth, making it very unreliable for solving a Rubik's Cube, although

much more likely to find a solution at greater depths than BFS due to its tiny memory footprint.

IDDFS typically provides the optimal solution with the fastest time out of all three algorithms, optimal due to its complete nature and faster than BFS because of its small memory.

Table 1: Number of nodes (states) in the search space for a given scramble length. The search space increases exponentially, as can be seen in the table. Note the branching factor is 11 to remove immediate inverse moves, and the numbers are still rough as there are looping paths, e.g., U U U instead of U', which appears in DFS solutions rather frequently.

# of moves in scramble	N	1	2	3	4	5	6	7	8
Search space (# of states)	$\frac{n(1-r^n)}{1-r}$	12	144	1,596	17,568	193,260	2,125,872	23,384,604	257,230,656

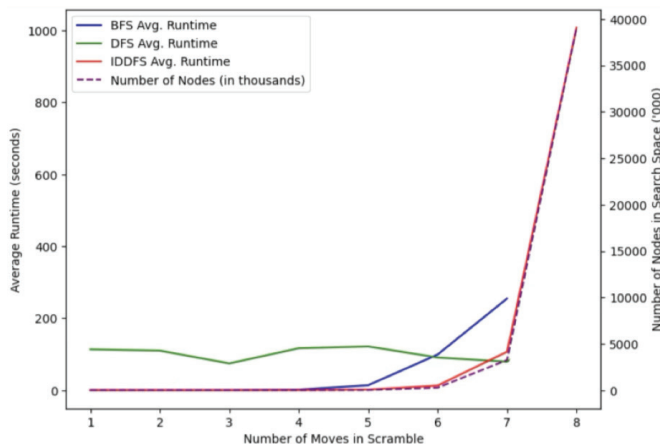


Figure 1: Average runtimes across the number of moves in the scramble for BFS, DFS, and IDDFS. The time taken for IDDFS to find a solution for n moves closely follows the trend for the number of nodes in the search space. BFS runs out of space at 8 moves, and DFS seems to take a similar amount of time regardless of move count.

As can be seen in Figure 1 above, comparing the number of moves in the scramble with the number of nodes in the search space (Table 1) and average runtime for each search algorithm, the number of nodes in the search space increases exponentially. IDDFS also has a trend similar to this increase in the number of nodes, as its systematic iterative deepening at each level is dependent on the length of the scramble (and solution), therefore searching every node at each depth with linear memory, so it does not store every path. DFS takes about the same time to return a solution regardless of scramble length, which makes sense due to its nature of searching deep solutions and potentially missing shorter ones. Note that the depth limit for DFS is set to be 7 moves due to computational constraints. The average time taken by DFS to return no solution for 8 moves is also, in essence, the time taken for it to visit all nodes up to 7 moves, which is actually faster than the average time taken by BFS to generate a solution for 7 moves, but still slower than IDDFS (Figure 1). In further experiments, god's number, the fact that any state of the cube can be solved in 20 moves,³ may be used to set the depth limit, but this may change the runtimes for lower move counts for DFS, as its search space is greatly expanded. The more efficient method for setting DFS's depth limit would be to use the scramble length, but this is unrealistic for solving unknown scrambles.

Although IDDFS is typically the fastest, DFS's reliance on luck is evident with one scramble "U, U", where it found the solution "U, U" in by far the fastest time for any 2-mover (Results). This is expected as U is the first move in the set move order, and it just so happens that in this case, the first path explored by DFS is the solution, while both BFS and IDDFS have to sweep the first depth. The runtime for DFS seems to be random, as, due to its inherent nature, DFS is largely dependent on the specific moves that make up the solution.

Conclusion

BFS works well for solving a Rubik's Cube when there is plenty of RAM available, and especially for shorter scrambles where it is able to guarantee the optimal solution in a reasonable amount of time. DFS has the advantage of a small memory footprint, but its reliance on a known and set depth limit makes it both unreliable and unrealistic for solving a Rubik's Cube. Even with God's number of 20, DFS may produce unnecessarily long solutions or fail to find a solution even after a long time, as it is luck-based. However, when the set depth level is similar to the scramble/solution length, DFS may even be faster than IDDFS.

A combination of BFS and DFS, in essence, we can conclude that IDDFS is the best search algorithm of the three for solving a Rubik's Cube. With the optimality of BFS and the scalability of DFS, IDDFS's iterative deepening can guarantee the optimal solution in a reasonable amount of time using a reasonable space complexity.

Given enough RAM, further experimentation can be done to discover the limits of IDDFS with respect to Rubik's Cube solving and how it compares with IDA* in terms of time taken to find the optimal solution. Setting DFS's depth limit to god's number, 20, would also be worthwhile to see its results. I have not done so for the sake of time, and the main goal is to compare these 3 search algorithms rather than delving deep into 1.

The Rubik's Cube is just one example of the applications of a search algorithm, which I happen to be very interested in. A better understanding of the risks and benefits involved with each of these fundamental search algorithms in graph theory helps us to make better decisions when it comes to similar real-life situations where there are legitimate consequences.

Acknowledgments

I would like to acknowledge Qi Xuan Khoo for being my mentor for this research article.

References

1. Daniels, J. S., Moreau, D., & Macnamara, B. N. Learning and Transfer in Problem-Solving Progressions. *Journal of Intelligence*, 10(4), 85. <https://doi.org/10.3390/jintelligence10040085>. 2022.
2. Duberg, D., & Tidestrom, J. Comparison of Rubik's Cube Solving Methods Made for Humans. <https://www.diva-portal.org/smash/get/diva2:812006/FULLTEXT01.pdf>. 2015.
3. Rokicki, T., Kociemba, H., Davidson, M., & Dethridge, J. The diameter of the Rubik's Cube group is twenty. *siam REVIEW*, 56(4), 645-670. <https://tomas.rokicky.com/rubik20.pdf>. 2014.

4. Chen, J. Different Algorithms to Solve a Rubik's Cube (J. Phys.: Conf. Ser. 2386 012018). <https://iopscience.iop.org/article/10.1088/1742-6596/2386/1/012018/pdf>. 2022.
5. Mehta, H., Kanani, P., & Lande, P. Google Maps (International Journal of Computer Applications (0975 – 8887) Volume 178 – No. 8). https://www.researchgate.net/profile/Pratik-Kanani/publication/333117435_Google_Maps/links/5eb6cc7da6fdcc1f1dcb10aa/Google-Maps.pdf. 2019, May.
6. Grumser, T. Engineering Optimal Solvers for Rubik's Cubes (Doctoral dissertation, Karlsruher Institut für Technologie (KIT)). <https://ae.iti.kit.edu/download/grumser-tan.pdf>. 2022.
7. Rokicki, T. Twenty-Five Moves Suffice for Rubik's Cube (No. arXiv:0803.3435). arXiv. <https://doi.org/10.48550/arXiv.0803.3435>. 2008.
8. Korf, R. E. Finding optimal solutions to Rubik's Cube using pattern databases. In AAAI/IAAI (pp. 700-705). <https://cdn.aaai.org/AAAI/1997/AAAI97-109.pdf>. 1997, July.
9. Kaur, H. Algorithms for solving the Rubik's Cube (KTH Royal Institute of Technology). <https://www.diva-portal.org/smash/get/diva2:816583/FULLTEXT01.pdf>. 2015, May 27.
10. Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS). GeeksforGeeks. <https://www.geeksforgeeks.org/dsa/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/>
11. Lozano-Pérez, T., & Kaelbling, L. P. Notes on the Complexity of Search (MIT 6.034 Artificial Intelligence, Fall 2003). Massachusetts Institute of Technology. <http://www.ai.mit.edu/courses/6.034b/searchcomplex.pdf>. 2003, September 23.

■ Author

Richard Tao is a Rubik's puzzle enthusiast who is a senior at Macleans College in Auckland, New Zealand. He is the founder of Cubing East Auckland, a non-profit organization promoting speedcubing in the local community. Richard plans to go to college in the United States one day, majoring in mathematics.