Università degli Studi di Cagliari

LM INGEGNERIA DELLE TECNOLOGIE PER INTERNET

UNIVERSITÀ DEGLI STUDI DI CAGLIARI

ARTIFICIAL INTELLIGENCE PROJECT

A Comparative Study of Search Algorithms for Rubik's Cube Solvers

Authors:
Edoardo Pilia (ID: 70/91/00138)
Alessia Congia (ID: 70/91/00160)

Abstract

The Rubik's Cube is a classic of computational search theory. It is a finite, deterministic, discrete state-space search problem, offering a rich environment for implementing and testing various search algorithms. The objective is to find a sequence of moves (a path) that transforms any given scrambled state of the $3 \times 3 \times 3$ cube into the solved state.

1 Introduction and Problem Definition

The Rubik's Cube, invented in 1974 by Hungarian architect Ernő Rubik, is a globally recognized and structurally complex combinatorial puzzle. This $3 \times 3 \times 3$ cube is composed of 54 smaller squares (*sticker*) distributed across six faces, each defined by a color (typically Red, Yellow, Green, Blue, White, and Orange).

Figure 1: Rubik's Cube.

This puzzle can be treated as a search problem.
Generally, a search problem requires, first, a precise formalization. This process fundamentally involves two primary steps:

- **Goal State Formulation**: identifying the target configuration that defines the problem as solved or terminated.
- **Problem Formulation**: determining the set of legal actions (or operators) and delineating the complete state space of the problem.

In Rubik's Cube case, the goal state corresponds to the unique solved state, where all nine cubes on each face display a single, uniform color. This transformation is achieved by applying a sequence of face rotations, which act as the operators in the search space.

The complexity of the Rubik's Cube is derived from the immense number of achievable configurations. In fact, the puzzle's state space comprises approximately 43 quintillion ($4,3 \times 10^{19}$) distinct permutations.
Despite its vastness, this state space is both finite and deterministic. In 2010, Rokicki et al. proved that any configuration of the $3 \times 3 \times 3$ Rubik's Cube can be solved in a maximum of 20 moves or less. This number, known as *"God's Number"* establishes the maximum depth for an optimal solution path [1].

2 Cube Representation and Implementation

Independently by the search algorithm implemented for the solution, it is necessary to define a data structure that is both universally usable and capable of accurately modeling the cube's states and its dynamic transitions.
Given the objectives of minimizing the memory consumption while facilitating rapid, correct state transitions, a robust and consistent structural design is proposed.

This section details the technical design choices made for implementing the $3 \times 3 \times 3$ Rubik's Cube within the Python environment. We will focus on the specific data structures utilized for encoding the cube's configuration—arrangement of individual faces and the resulting compound cube structure—followed by the rigorous implementation of the six fundamental face rotations that define the problem's legal actions.
Finally, we establish the necessary node structure required to encapsulate the state within the search graph, enabling systematic and efficient state space exploration.

2.1 Cube Data Structure

The internal representation of the Rubik's Cube must guarantee an efficient, unambiguous, and compact encoding of any possible configuration. To achieve this, the implementation is structured around two core classes: **face**, which models an individual 3×3 face, and **cube**, which aggregates six faces into a complete cube configuration. This design ensures modularity, ease of access to face data, and precise control over state transitions. We will now present the data structure implementation more in detail.

Each face of the cube is represented as a 3×3 matrix of symbols, where each symbol corresponds to a color and is identified by the variables `R`, `W`, `G`, `B`, `Y`, `O`. This matrix-based representation preserves the geometric structure of the physical puzzle and allows direct manipulation of rows, columns, and faces.

The color of a face is implicitly defined by the value of its central sticker, which is immutable, stored in the attribute `self.color`. In this way the central sticker determines the identity of each face.
The class `face` provides controlled access to its internal structure through methods such as `get_row` and `get_column`, which extract full rows or columns from the matrix when needed for interactions with adjacent faces.
Symmetrically, the methods `switch_row` and `switch_column` allow replacing a specific row or column with another one while enforcing an important physical constraint: the central row and central column cannot be modified, since in the real puzzle no legal move alters the middle layer of a single face in isolation.
Another operation implemented within the face class is the `rotate` method, which performs a rotation of the face itself. The rotation is modeled by selectively copying the outer rows and columns of the matrix and rearranging them according to the chosen direction: a clockwise rotation when `backward=False` and a counter-clockwise one when `backward=True`. This method ensures that only the external sticker of the face are permuted, while the central sticker remains fixed.

The class `cube` models the full Rubik's Cube as a structured container composed of six instances of the face class, each representing one face of the puzzle. These faces are stored in a fixed internal order, which serves as a convention for our project implementation that simplifies state access, manipulation, and comparison across different cube configurations.
This class also encapsulates a collection of methods responsible for implementing all fundamental cube operations, which will be discussed more in detail in the following subsection.
We also implemented a `print` method to display the current state of the cube in a human-readable way. Finally, the class includes the static method `create_target`, which constructs the goal state of the puzzle. This method provides a clear and uniquely identifiable reference for evaluating the correctness of the solution.

2.2 Move Generation

In the standard formulation of the Rubik's Cube, the set of legal actions comprises 24 moves. Each of the six faces can be rotated clockwise or counter-clockwise by $90°$ or $180°$. However, a $180\circ$ clockwise rotation equals to a $180\circ$ counter-clockwise rotation, then the number of the moves r can be reduced to 18. These operations define the full **branching factor** of the search space.
In the implemented system, the rotation logic is encapsulated within the class cube.
A key design decision is that every move—regardless of which face has been rotated—is internally expressed as a **rotation relative to the red face**. This approach yields an important advantage: it ensures that rotational logic remains centralized and consistent, avoiding redundant implementations.

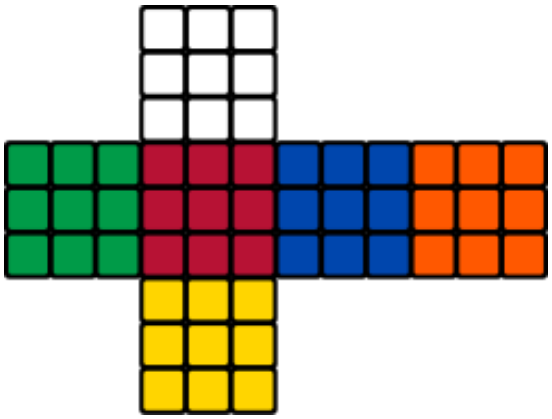

Figure 2: Rubik's Cube reference that we use for project implementation.

The full set of 18 legal actions is generated by three Python methods which utilize boolean flags to parameterize every aspect of the move:

- `rotate_red_row(self, backward, up, double)`: this method handles horizontal rotations (if `up=True` this action corresponds to a white face rotation, while if `up=False` corresponds to a yellow face rotation).
- `rotate_red_column(self, backward, right, double)`: this method handles vertical rotations (if `right=True` this action corresponds to a blue face rotation, while if `right=False` corresponds to a green face rotation).
- `rotate_face(self, backward, red, double=False)`: this method handles rotations of the front or back faces (if `red=True` this action corresponds to a red face rotation, while if `red=False` corresponds to a orange face rotation).

Backward and double are two boolean variables which represent respectively the direction of the rotation (clockwise if `backward=False` and counter-clockwise if `backward=True`) and the angle of rotation (90° rotation if `double=False` and 180° rotation if `double=True`).
A potential issue arises from the possibility of overlapping configurations, where identical resulting states are achieved starting from the same initial configuration but via different action. This issue will be resolved during the execution of the search algorithms by discarding identical and redundant configurations.
The function `execute_function_set` serves as the **successor function** to the search problem, central to generating the search space from any given state. This method systematically executes the complete set of 18 legal moves on the current configuration.
First, the function initializes a list of all 18 moves, parameterized by the three core rotation functions and their respective boolean flags. This ensures that the full branching factor of the search space is explored.
Second, it implements a basic check to prevent trivial cycles by comparing the newly generated state with the state of the immediate parent node. If the states are identical,

the move is discarded. This happens for each node expansion, then the number of new nodes generated by an expansion will be the branching factor minus one.
Subsequently, for every valid successor state, the function creates a reference to the current node, designating it as the parent node, thus enabling path reconstruction.
Finally, the function returns a set of these newly generated nodes. This set contains all the valid successor states and it is immediately available for insertion into the queue of the implemented search algorithms.

2.3 State Representation

The class `cube_node` defines the structural representation of a Rubik's Cube state within the search space. Each node encapsulates the complete set of information required for algorithmic state exploration, path reconstruction, and—when applicable—evaluation under informed search strategies. In particular, each node stores:

- **current cube configuration** `self.current`, which is represented by an instance of the class cube. This object encodes the full physical state associated with the node in the search tree.
- **reference to the parent node** `self.parent`, that enables reconstruction of the solution path by backtracking once the goal state is reached.
- **depth of the node** `self.depth`, corresponding to the number of moves from the root state to the current node. The depth of the node is useful in all search algorithms and is also used in informed search strategies to contribute in the calculation of the node function evaluation (the function $g(n)$ corresponds to the depth node).
- **value for the node evaluation function** $f(n)$ `self.function`, used when the node participates in informed strategies such as A*. This field stores the result of applying an evaluation function of the form $f(n) = g(n) + h(n)$, where the last term is the heuristic function.

To support heuristic computation, the class provides the method `get_face_distance`, which computes the Manhattan distance over the cube configuration. This method evaluates how far each sticker is from its target face, and the method `cube_heuristic` aggregates the distances into a scalar value (divided by eight to make it comparable to the depth) that reflects the degree of disorder of the state.
A crucial aspect of the design is that the heuristic must be admissible, that is, it must never overestimate the true minimum number of moves required to reach the goal. Admissibility guaranties optimality for informed strategies.
In charge of this, the Manhattan heuristic is a consistent approximation of how close a configuration is to the goal state, guiding the search process efficiently through the enormous state space of the puzzle.

3 Analysis and Implementation of Searching Algorithms

This section presents the analysis and implementation of the three search strategies adopted for solving the Rubik's Cube within the defined problem framework.
We examine two uninformed search methods, **Breadth-First Search (BFS)** and **Depth-First Search (DFS)**. These algorithms provide fundamental baselines for systematic exploration.
Subsequently, we introduce an informed search strategy, the **A* algorithm**, which integrates a problem-specific heuristic to guide the exploration toward more promising configurations.
For each of the three strategies, we detail the design decisions, data structures, and operational logic adopted in the implementation.

3.1 Uninformed Search: Breadth-First Search Algorithm

Breadth-First Search is one of the fundamental uninformed search strategies. Starting from the initial configuration, BFS expands all nodes at depth d before considering any node at depth $d+1$. This guarantees completeness: if a solution exists, BFS will eventually find it. Typically, BFS strategy is not optimal, unless the path cost is a non-decreasing function of depth, which is the case of the Rubik's cube. However, this approach is characterized by a very high memory consumption, since it must store all frontier nodes at each depth level. Given the combinatorial nature of the puzzle, memory management becomes a critical component of the algorithm's implementation.

The algorithm is represented by the function `elaborate` which initializes two main data structures:

- `expanded`, a set containing cube configurations already expanded;
- `visited`, a set ensuring that no configuration is inserted into the queue more than once.

The BFS loop proceeds while the queue is non-empty. At each iteration, the algorithm dequeues the shallowest available node using `queue.popleft()`.
If the dequeued node has not yet been expanded, it is inserted into `expanded` set to prevent re-expansion.
A goal-test immediately checks whether the current configuration matches the target state. If so, the function returns the target and the number of expanded nodes. The reconstruction is possible by calling recursively the parent node.
Otherwise, the algorithm generates all successor nodes by invoking `execute_function_set` function, which applies the full set of legal cube moves to the current state. For each successor, the implementation ensures that its configuration has not yet been visited; in that case, the node is enqueued and recorded in the `visited` set.

If the queue becomes empty without encountering the target, the function returns `None`. Since BFS is complete, such a condition suggests that the input cube configuration was invalid.

3.2 Uninformed Search: Depth-First Search Algorithm

Depth-First Search is an uninformed search strategy that explores the state space by always expanding the deepest unvisited node available. Unlike BFS, which performs a level-order expansion, DFS follows a single path as far as possible before backtracking. This approach strongly reduces memory usage, since at any time DFS stores only a single path from the root plus the frontier of unexplored siblings.
DFS algorithm is complete unless there are infinite or cyclic spaces, and unless a mechanism such as depth limit is introduced. Considering the particularity of the Rubik's cube, having a depth limit equal to the God's number can guarantee completeness. However, we decided for a smaller limit (which is 6). This transforms the procedure into a form of **Depth-Limited Search (DLS)**, ensuring termination in smaller time while enabling controlled exploration of deeper layers, avoiding extremely high elaboration time. Furthermore, DFS is not optimal: the first solution found is not guaranteed to be the shortest.
The DFS routine implemented in the function `elaborate` reflects these design considerations.

The main structure implemented by the algorithm is `expanded`, a set containing cube configurations already expanded, ensuring fast duplicate detection and preventing unnecessary regenerations of identical states.
The search loop proceeds while the queue is non-empty. At each iteration, the algorithm removes the first node in the queue via `queue.popleft()`. Although a queue is used, DFS behavior is achieved by subsequently pushing new nodes to the front of the queue (`appendleft`), effectively turning it into a Last In First Out structure.
A node is eligible for expansion only if its cube configuration has not yet been expanded, and its depth does not exceed the predefined depth limit.
Once a node is added to `expanded` set, a goal-test checks whether it matches the target. If the solved configuration is encountered, the function returns the target node and the number of expanded nodes.
Otherwise, if the conditions are met, successors are generated through `execute_function_set`, which applies the full set of legal cube moves. Unlike BFS, which appends successors to the end of the queue, DFS inserts each new node at the front. This ensures that the most recently generated node is expanded next, reproducing the depth-first behavior.
If the queue is exhausted without finding a solution, the function returns `None`. Considering that DFS is complete unless is bounded, such an outcome is plausible even when the initial configuration is valid, especially with strict depth limits.

3.3 Informed Search: A* Algorithm

A* represents the most advanced of the implemented strategies, using heuristic evaluation to guide the exploration of the state space efficiently. The algorithm assigns each node a score determined by the evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the depth (cost from the initial configuration) and $h(n)$ is the heuristic estimate of the remaining distance to the target. The heuristic therefore plays a central role: if it is admissible, A* is both complete and optimal, guaranteeing that the solution found is the shortest possible one.
The designed algorithm relies on the custom heuristic function implemented in `cube_heuristic`, which provides an informed estimate of how far the current cube configuration is from the goal state. This estimate determines the priority of nodes inside the priority queue, implemented via Python's `heapq` module.
A design choice in this implementation is the use of a depth limit equal to 21, motivated by the *"God's Number"*. This limit helps reducing the search space while maintaining correctness.
The algorithm updates two main structures:

- priority `queue`, a min-heap that always extracts the node with minimal $f(n)$ value, ensuring heuristic-driven expansion;
- `expanded`, a set containing cube configurations already expanded;

Each iteration extracts the best node from the queue, checks whether it exceeds the depth limit, and evaluates its heuristic. If the configuration matches the target, the algorithm terminates and returns the target node and the number of expanded nodes.
Otherwise, the function `execute_function_set` is invoked to generate all successor states obtained from applying the available cube moves. Each of these newly generated nodes is then pushed into the priority queue with a priority given by $g(n) + h(n)$.
Finally, if the goal is not reached, the function returns `None`. Given that A* is complete and optimal under the depth and heuristic constraints, failure to find the solution typically indicates an invalid cube configuration.

4 Performance Evaluation

The objective of a search algorithm is to ensure both effectiveness and efficiency.
Effectiveness is guaranteed if the algorithm is able to reach the solution, if one exists (**completeness**), and if it reaches the solution by following the minimum path cost (**optimality**). **Efficiency**, on the other hand, concerns algorithm performances in terms of **computational complexity**.

The table below shows the theoretically expected level of computational complexity for each algorithm, where b represents the branching factor, d corresponds to the solution

depth, and m is the maximum depth of the search tree in the worst case (used for DFS strategy).

Algorithm	Time Complexity	Spatial Complexity
BFS	$O(b^{d+1})$	$O(b^{d+1})$
DFS	$O(b^m)$	$O(bm)$
A*	$O(b^d)$	$O(b^d)$

Table 1: Search Algorithm Comparison

In order to test our search algorithms, we have set up a unique `main` script in which is required to select the desired strategy by inserting number `1`, `2` or `3`, that correspond respectively with BFS, DFS and A* algorithms. Cube initial configuration is constructed starting from target configuration and applying in it some legal move. Then, the queue list is initialized using this configuration as the root node and the proper `elaborate` function is called. The `execute_algorithm` function returns three structures: `current_node`, that is the final node which will contain the solved configuration if the strategy performs correctly, `iteration`, which represents the number of total expanded nodes, and `queue`, that is the list containing nodes waiting in the queue.

In this next section, simulation results will be presented in order to provide evidence about the optimality of the algorithms and computational complexity, highlighting time and memory consumption in terms of expanded nodes and number of nodes waiting in the queue for each algorithms.

4.1 Results

Tables below show the outcomes of our experimentation. We evaluated the algorithms by using different known cases, starting from one move missing from the solution to a maximum of eight moves. This last case was only experimented using A*, since the computational and memory requirements for this scenario using BFS and DFS were too high for our resources. However, the most notable scenarios were with three and five moves, which were tested three times using different starting configurations. All the tests were executed using the same workstation in similar execution environment.
Before showing test results, it's important to introduce some consideration we have done about time and spatial complexity calculation. All the search strategies have a time complexity equals to the total number of expanded nodes. In BFS, due to systematical exploration of the search space, the space complexity equals the time complexity. Also in A* the space complexity corresponds to the time complexity. DFS strategy introduces a strong reduction of memory consumption against BFS. However, two scenarios can occur: in the best case scenario, the solution is found in the first branch and the space complexity is defined as the sum of the depth of the solution and the length of the

queue; in the other case, the space complexity is defined as $1 + b * m$, where m is the maximum depth found (in our case, the depth limit) and b is the branching factor. Considering a branching factor of 17 (24 minus six redundant moves minus the opposite of the last move) and a depth limit of 6, the space complexity is 103. In our simulations, only the second scenario occurs.

Algorithm	**Expanded Nodes**	**Queue**	**Path length**	**Time C.**	**Spatial C.**
BFS	17	204	1	17	17
DFS	90080	71	5	90080	103
A*	2	34	1	2	2

Table 2: Test 1: One Move

Algorithm	**Expanded Nodes**	**Queue**	**Path length**	**Time C.**	**Space C.**
BFS	125	1543	2	125	125
DFS	14756	71	5	14756	103
A*	3	50	2	3	3

Table 3: Test 2: Two Moves

Algorithm	**Expanded Nodes**	**Queue**	**Path length**	**Time C.**	**Space C.**
BFS	1634	20131	3	1634	1634
DFS	124793	44	5	124793	103
A*	6	98	3	6	6

Table 4: Test 3: Three Moves - A

Algorithm	Expanded Nodes	Queue	Path length	Time C.	Space C.
BFS	2051	25250	3	2051	2051
DFS	19817	71	4	19817	103
A*	4	66	3	4	4

Table 5: Test 4: Three Moves - B

Algorithm	Expanded Nodes	Queue	Path length	Time C.	Space C.
BFS	3326	40930	3	3326	3326
DFS	19510	40	5	19510	103
A*	6	98	3	6	6

Table 6: Test 5: Three Moves - C

Algorithm	Expanded Nodes	Queue	Path length	Time C.	Space C.
BFS	558750	6837636	5	558750	558750
DFS	562668	46	5	562668	103
A*	65	1015	5	65	65

Table 7: Test 6: Five Moves - A

Algorithm	Expanded Nodes	Queue	Path length	Time C.	Space C.
BFS	461056	5643934	5	461056	461056
DFS	103398	44	5	103398	103
A*	42	658	5	42	42

Table 8: Test 7: Five Moves - B

Algorithm	Expanded Nodes	Queue	Path length	Time C.	Space C.
BFS	581995	7121890	5	581995	581995
DFS	547780	30	5	547780	103
A*	22	352	5	22	22

Table 9: Test 8: Five Moves - C

Algorithm	Expanded Nodes	Queue	Path length	Time C.	Space C.
A*	16132	252880	8	16132	16132

Table 10: Test 9: Eight Moves

Conclusions

Results show that between this algorithms, A* outperforms the others in both time and space complexity, and in finding an optimal solution. However, if more than eight moves are missing, the algorithm starts to suffer. The only test we perform with 10 moves were still elaborating after more than 10 minutes with an extremely high memory usage (¿22 GB RAM). This proves that this heuristic is admissible for partially completed cubes. About BFS, the memory constraints made impossible to perform tests with more than five moves. Anyway, considering that the path cost is a function of the depth while solving the Rubik's Cube, BFS guarantees to obtain an optimal solution. About DFS, the depth limit at 6 was implemented to avoid extremely high elaboration time and memory usage. This can be seen in the first tests, expanding a very high number of nodes and finding a far-from-optimal solution. The gap between limited DFS and BFS is reduced while approaching the limit. During the five moves tests, DFS performed similar to BFS in terms of time complexity.

References

[1] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge. God's Number is 20. Available: `https://www.cube20.org/`, 2010. [Accessed: 15 December 2025]. pages 3