CUBIX: A Case Study of Search Algorithms on $N \times N \times N$ Rubik's© Cube

Rohit Doriya, Priyank Mandal, Rohit Kumar, Karan Jain

https://github.com/FreakyAI-star/Cubix

Abstract— Rubik's cube came out in 1980 as a puzzle for kids but since then it has challenged a lot of eminent computer scientists to come up with a good optimal solution for any scramble out of 43,252,003,274,489,856,000 possible unique combinations. Being AI enthusiasts, this number has thrilled us too because it is a huge huge search space to work with, but at the same time contains so many elegant patterns to reduce this search space considerably. Thus, this paper tries to explore various search techniques like BFS, A*, IDA*, etc. that we have learnt from the course along with some intelligent heuristics to search for an optimal solution path of any scramble. This paper also aims to explore several other heuristics and compare the results obtained from different search techniques.

Keywords— Layering algorithm, Kociemba's algorithm, Breadth-First Search (BFS), Better Breadth-First Search (BBFS), A*, Iterative Deepening A* (IDA*), Path-Time trade-off

I. Problem Statement

Our problem statement is to search for an optimal solution path for any $N \times N \times N$ rubik's cube starting from a standard $3 \times 3 \times 3$ rubik's cube using various search algorithms with intelligent heuristics and then compare the results. In this rubik's cube there are 27 possible $1 \times 1 \times 1$ cubies. The six cubies in the middle of each face rotate but remain stationary. These six cubes create a fixed reference framework that prevents the entire cube from rotating. The remaining 20 movable cubies are distributed as follows: 12 are on the edges with two visible faces each, and 8 are on the corners, each with three faces.

Edge cubie only move inside edge positions, while corner cubies only move within corner positions. An edge cubie can be orientated in two distinct ways, while a corner cubie can be oriented in one of three ways in a specific place. By indicating the location and orientation of each of the 8 edge cubies and 12 corner cubies, we can thereby characterise the state of a cube in an entirely unique way. This is shown as a 20-element array, one for each cube, the contents of which encode the position and orientation of the cube as one of 24 distinct values, with 8×3 for corners and 12×2 for edges. Thus, there are $8! \times 3^8! \times 12! \times 2^{12}$ total possibilities. Since there are no permitted moves between any two of the problem space's 12 distinct but isomorphic subgraphs, further constraints limit this by a factor of 12. As a result, there are a total of $8! \times 38 \times 12! \times 2^{12}/12 = 43,252,003,274,489,856,000$ states that can be reached from a particular state.

Initial State: the Initial state for the problem statement would be an scrambled rubik's cube achieved by mixing an unscrambled rubik's cube using any combination of legal moves.

Operator: The operations that can be performed on the rubik's cube are Front, Back, Top, Bottom, Right and Left which represent clockwise rotation of the specific face (Front represents rotation of the front face in clockwise direction) and there primes too, which represents rotation of the face but in anti-clockwise direction.

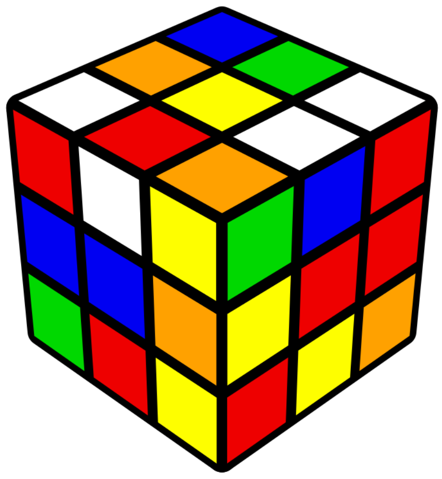


Fig. 1: $3 \times 3 \times 3$ Rubik's cube

State Space: The state space will be containing all the possible combination's that exist in a Rubik's cube which is around $8! \times 38 \times 12! \times 212/12 = 43, 252, 003, 274, 489, 856, 000$ states

Goal: The goal of the problem is to solve the rubiks cube, which is to achieve a single colour on each of its face, such that minimum number of moves are required, and is computed in minimum amount of time.

II. Literature Survey

We explored various search techniques used for solving rubik's cube. A common technique used nowadays is Deep Reinforcement learning which learns how to solve increasingly difficult states in reverse from the goal state without any specific domain knowledge [1]. Apart from this there are also studies on improving the heuristics by using a trained neural network as a heuristic distance estimator with a standard forward-search algorithm [2]. But since our focus is mainly on testing the traditional search algorithms taught in the course we studied each of these algorithms individually and then implemented it for the search space of the Rubik's cube [3, 4, 5]. We also went through traditional Rubik's cube solving algorithms like layering method and kociemba's algorithm [6] and korf's algorithm [7] as baseline algorithms for comparing our results.

III. Motivation

Invented by Hungarian Erno Rubik in the late 1970s, the Rubik's Cube is the most famous combination puzzle of its time. The standard version consists of a subcube or his 3x3x3 cube with a different colored sticker on each exposed square of the cube. Each 3x3x1 plane of the cube can be rotated or rotated 90, 180, or 270 degrees with respect to the rest of the cube. In the goal state, all squares on each side of the cube have the same color. The puzzle is jumbled up in a series of random spins and the task is to return the cube to its original target state. The Rubik's Cube is a fascinating and timeless puzzle with trillions of possible states. As some cube-geek turned AI-enthusiast, and armed with a handful of human-centric algorithms for solving the cube, We decided to try our hands at programming a Rubik's Cube solver. No modern computer can solve the cube with a simple brute-force algorithm. A 3x3 Rubik's Cube has 43,252,003,274,489,856,000 permutations with a branching factor of about 13, which is too many and too many states to traverse. Inventing a programmatic solver requires a pinch of efficient data structures, a pinch of group theory, and a dash of clever algorithms.

IV. Methodology

Our methodology is quite straight forward. We first try to explore the search tree composed of all possible rubik's cube combinations one by one using Breadth first search (BFS) where we traverse all nodes at each depth before moving forward in search of goal node. Next, we improve the performance of our previous search algorithm by not traversing the nodes which are redundant (i.e., nodes which lead to the same position as some previously explored node). This way we can reduce the amount of search tree to be explored and get results in a shorter time. We called this improvement of BFS algorithm as Better BFS (BBFS). Building upon this, we next try to make our algorithms intelligent in choosing nodes which lead to goal state by using heuristics. This lead us to use A* algorithm.

We have used three different heuristics for A* algorithm: *Euclidean distance heuristic*, *Hamming distance heuristic* and *Manhattan distance heuristic*. Each of these heuristics are explored in the subsections below. After this we move on to improve this search exploration further using Iterative Deepening A* in which the depth of search tree is increased iteratively until a goal state is reached. Thus IDA* algorithm has been our best attempt to find the optimal solution. By optimal solution we mean to address a solution path which is small but at the same time takes a reasonable amount of time to compute. We call this trade-off between solution path and solution length as path-time trade-off.

a. BFS (Breadth first search)

The breadth-first search algorithm is a graph traversal technique that chooses a random initial node (source or root node) and starts traversing the graph layer by layer, visiting and exploring all nodes and their respective child nodes. Breadth-First Search algorithm follows a simple, level-based approach to solve a problem. In deciding children we give input 'all', which here means to give children with respect to all the moves possible in Cube. This may include some redundant moves (and thus States too).

Algorithm 1 Breadth First Search

```
Function solve(Start_Node):
    Goal_Node = Solved_Cube()
    depth = 0
    if Start_Node == Goal_Node:
        return [Start Node]

    seen = Directory()
    seen.Insert(Start_Node)

    fringe = Directory()
    fringe.Insert(Start_Node)

    while True:
        depth += 1
        new_fringe = Directory()
        for i in fringe:
            for j in i.children('all'):
                if j == Goal_State :
                      return path
                if j not in fringe and j not
    in seen:
                    new_fringe.Append(i)
                    seen.Append(i)
        fringe = new_fringe
```

b. BBFS (Better Breadth first search)

The Better Breadth First Search works exactly like the Breadth First Search but with just one difference and that is it evaluates much less options. Thus the branching factor for this tree is reduced. The difference lie in selecting the children for a particular node, here rather than considering all the children we just consider limit moves and thus less number of children. We just choose the children formed by 3 basic moves of the Rubik's cube, front, up, right. Rest all the moves can be implemented by using these moves repeatedly and by changing the position of cube.

Algorithm 2 Better Breadth First Search

```
Function solve(Start_Node):
    Goal_Node = Solved_Cube()
    depth = 0
    if Start_Node == Goal_Node:
        return [Start Node]

    seen = Directory()
    seen.Insert(Start_Node)

    fringe = Directory()
    fringe.Insert(Start_Node)

    while True:
        depth += 1
        new_fringe = Directory()
        for i in fringe:
            for j in i.children('limit'):
                if j == Goal_State :
                      return path
                if j not in fringe and j not
    in seen:
                      new_fringe.Append(i)
                      seen.Append(i)
        fringe = new_fringe
```

c. Heuristics

Euclidean Distance Heuristic - In order to apply this heuristic to our search algorithm we have to decide the reference state from which we have to check the distance. The best possible way should be to calculate the distances between the same color faces of cubies to the same color center cubie and try to minimize it. We applied it in a different fashion as we calculated a score in accordance with the number of cubes of the same color on one side of the cube. The more cubies with the same color the more the score is. We subtract this from the maximum possible score and then minimize this difference.

$$ED(p,q) = \sqrt{\sum_{i=1}^{n} (x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

Manhattan Distance Heuristic - For each cube, calculate the minimum number of moves required to position and orient the cube correctly, and sum these values across all cubes. A better heuristic is to take the maximum sum of the corner block Manhattan distances divided by 4 and the maximum sum of the edge blocks divided by 4.

$$MD(p,q) = |x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|$$

Hamming Distance Heuristic - As per coding theory, The Hamming distance between $x, y \in \{0,1\}$ is defined as the number of coordinates in which x and y differ. We can apply these heuristics to Rubik's cube by calculating the number of cubies or faces which were not in their correct position. We start a score equal to the total number of cubie faces and then use a nested loop to iterate for both the current state and the goal state. Whenever the color of the cubie face doesn't match we decrement the score by one. We try to maximize the score. In the Goal state, the score is equal to the total number of cubie faces.

d. A* Algorithm

A* is an informed search algorithm, which uses a heuristic function to find the next node with the smallest cost and iterates over it until a satisfying state has not been reached. after extending a node, it has to decide which node should be extended next which is done using a cost functions represented as: -

$$f(n) = g(n) + h(n)$$

where g(n) is the cost of the path from the start node to the current node while h(n) is the heuristic function that estimates the cost to reach the goal state.

A* algorithm is implemented using a priority queue to perform repeated expansion of the nodes which costs the least.

Algorithm 3 A* Algorithm

```
OBJECT State(cube, cost):
  self.cost = cost
  self.cube = cube

Function A*(Cube, Heuristic):
  fringe = PRIORITY_QUEUE()
  start  = STATE(cube,0)
  explored = SET()
  fringe.push(start)

  while !fringe.empty():
    current = fringe.pop()
    if current.isSolved():
      return path to reach state
    if current is in explored:
      continue
    else:
      for i in current.children():
        if i not in explored:
          fringe.push(STATE(i,current.cost +
  Heuristic(i) ))
      explored.add(current)
```

e. IDA* Algorithm

IDA* Algorithm, As described by its author Richard Korf, is a depth-first search algorithm, that looks for increasingly longer solutions in a series of iterations, using a lower-bound heuristic to prune branches once a lower bound on their length exceeds the current iterations bound. The algorithm just improves on the Depth first iterative deepening, by adding the concepts of A* algorithm where the heuristic function is given as :-

$$f(n) = g(n) + h(n)$$

where g(n) is the cost of the path from the start node to the current node while h(n) is the heuristic function that estimates the cost to reach the goal state. Here the function stops searching a path if the heuristic function f(n) > CUTOFF. In the beginning, The cutoff is set to the cost at initial state, whereas for the next iteration the cutoff is set to the minimum cost of all values that exceed the current cutoff.

The memory usage of this algorithm is less than the A* algorithm, but unlike A* it might be visiting some nodes multiple times.

Algorithm 4 IDA* Algorithm

```
OBJECT State(cube,cost):
  self.cost = cost
  self.cube = cube

Function Search(path,path_cost,bound,Heuristic
    ):
  current = path[-1]
  min_val = INF
  f = path_cost + Heuristic(current)
  if f > bound:
    return False, path, f
  if current.isSolved():
    return True, path, f
  for i in current.children():
    if i not in path:
      found, path, val_min = Search(path + [i
    ], f, bound, Heuristic)
      if found:
        return True, path, val_min
      if val_min < min_val:
        min_val = val_min
  return False, path, min_val

Function IDA*(Cube, Heuristic):
  bound = Heuristic(Cube)
  path  = [cube]
  while True:
    found, path, min_val = Search(path, 0,
    bound, Heuristic)
    if found:
      return path
    if min_val == INF:
      return []
    bound = min_val
```

V. Experiment Settings

To make a reasonable comparison between the algorithms which we have implemented based on our path-time trade-off it is necessary to decide some baseline algorithms. Henceforth, we have chosen two algorithms: *Layering algorithm* and *Kociemba's algorithm*. The layering algorithm simply solves the cube by matching all the colors of each of the three layers one by one. This is also the traditional way to solve a rubik's cube. Secondly, the kociemba's algorithm is a two phase algorithm. In phase 1, the algorithm looks for maneuvers which will transform a scrambled cube to G1 (G1 is a subset of possible moves obtained by turning the faces of a solved cube and not using the moves R, R', L, L', F, F', B and B'). That is, the orientations of corners and edges have to be constrained and the edges of the UD-slice have to be transferred into that slice. In phase 2 we restore the cube. There are many different possibilities for maneuvers in phase 1. The algorithm tries different phase 1 maneuvers to find a shortest solution [6].

We have made the comparision of all algorithms based on two simple parameters: *average solution length* and *average algorithm runtime*. This will enable us get a clear picture of the path-time trade-off.

VI. Results

After successfully setting up the experimental settings, we run the experiments on each of the algorithms for the various scramble lengths. We can observe that the A* class of algorithms with various heuristics perform similar to each other in terms of average runtime and have minimal difference. The A* class of algorithms lie in between the performance ordering with the BFS, BBFS and IDA* performing worse than A* in every scramble length and both kociemba and layering performing better than A* in every scramble length. Now on the other hand the top 2 best average runtime algorithms perform the worst in terms of the average solution length. Refer to experiment tables. Refer to figure 4 and figure 5.

From the graphical representation of the results, we can observe the abnormal entries for BFS being in the neighborhood of upper 100 secs for scramble length 6 in average

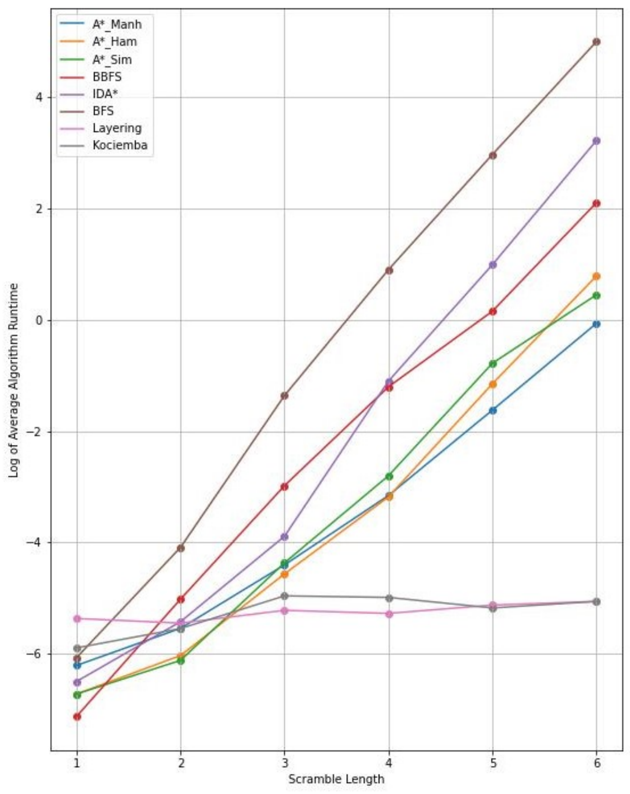


Fig. 2: Comparison of different algorithms on the basis of average algorithm runtime.

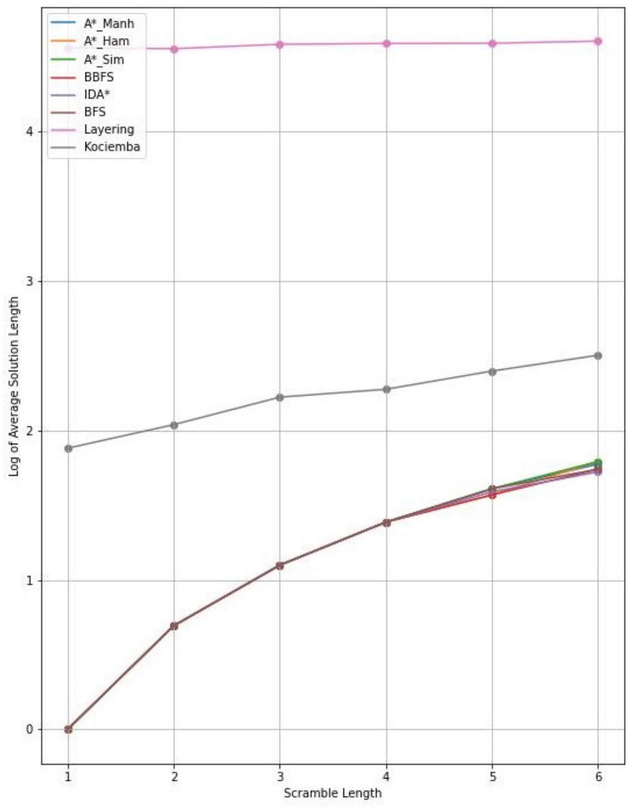
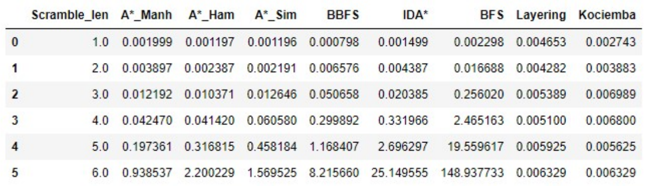


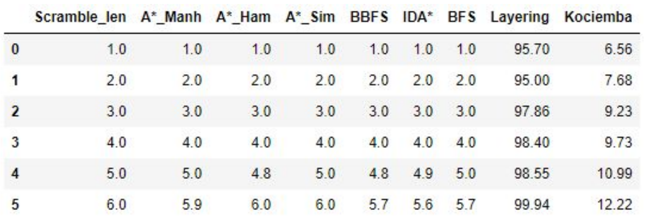
Fig. 3: Comparison of different algorithms on the basis of average length of solution path.

runtime and entries of layering being in the neighborhood of 100 moves for every scramble length for the average solution length, so in order to avoid visualisation errors we incorpo-



	Scramble_len	A*_Manh	A*_Ham	A*_Sim	BBFS	IDA*	BFS	Layering	Kociemba
0	1.0	0.001999	0.001197	0.001196	0.000798	0.001499	0.002298	0.004653	0.002743
1	2.0	0.003897	0.002387	0.002191	0.006576	0.004387	0.016688	0.004282	0.003883
2	3.0	0.012192	0.010371	0.012646	0.050658	0.020385	0.256020	0.005389	0.006989
3	4.0	0.042470	0.041420	0.060580	0.299892	0.331966	2.465163	0.005100	0.006800
4	5.0	0.197361	0.316815	0.458184	1.168407	2.696297	19.559617	0.005925	0.005625
5	6.0	0.938537	2.200229	1.569525	8.215660	25.149555	148.937733	0.006329	0.006329

Fig. 4: Comparison of different algorithms on the basis of average algorithm runtime.



	Scramble_len	A*_Manh	A*_Ham	A*_Sim	BBFS	IDA*	BFS	Layering	Kociemba
0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	95.70	6.56
1	2.0	2.0	2.0	2.0	2.0	2.0	2.0	95.00	7.68
2	3.0	3.0	3.0	3.0	3.0	3.0	3.0	97.86	9.23
3	4.0	4.0	4.0	4.0	4.0	4.0	4.0	98.40	9.73
4	5.0	5.0	4.8	5.0	4.8	4.9	5.0	98.55	10.99
5	6.0	5.9	6.0	6.0	5.7	5.6	5.7	99.94	12.22

Fig. 5: Comparison of different algorithms on the basis of average length of solution path.

rated the use of log function on each entry to plot out a better representation of the results. Refer to figure 2 and figure 3.

Performance in average runtime:

$Layering > Kociemba > A^*_{Manh} > A^*_{Sim} > A^*_{Ham} > BBFS > IDA^* > BFS$

Performance in average solution length:

$IDA^* = BBFS > BFS > A^*_{Ham} > A^*_{Manh} > A^*_{Sim} > Kociemba > Layering$

VII. Conclusion

a. Interpretation and Limitations

Our chosen metric for deciding upon which algorithm performs better was the path-time tradeoff, where in we consider the most optimal algorithm which computes the solution in a reasonable time with being considerable on the number of steps taken. Taking this metric into consideration we can see that every algorithm that we have tested upon has its own advantages and disadvantages, to list out some of them BFS has the property to find the shortest path there is to the goal state but at the same time it needs to explore all the nodes that come before it and this has its memory and time limitations on the other hand the Layering algorithm computes the solution of the current state of the cube in the neighborhood of 0.005 secs almost always but it lacks in the number of steps it takes as they are in the neighborhood of 100's, these two algorithms shows the extremes of the spectrum. The trend continues with all the algorithms and the answer to the question of "most optimal algorithm" was satisfied by the Kociemba Algorithm as it almost guarantees an average runtime of 0.005 secs across a scramble lengths and the solution length lies in the range of lower 15's. This fits best with our metric for deciding the optimal algorithm as it attempts to offer best of both worlds.

b. Future scope

The future scope for solving this problem can be by improving upon those algorithms which have shown promising results but have a potential to fine tuned further. Taking the A* class of algorithms into consideration they have a slightly varied performance based on the choice of the heuristic and this can be further improved with a better huerestics that provided the algorithm with better insights of where and how far is the goal state. Another improvement in this problem domain is the state termed as "SuperFlip", it is a state wherein the corners of the cube are solved but the edge pieces are in their correct position but flipped. The interesting thing about this state that there is a certain order of moves that solves this state in minimum time and this order of moves is true even if you start anywhere in the ordering given that you complete the ordering in a circular pattern. So instead of search for the final goal state that lies deep within the search space we can search for this intermediate state and reduce the search times for the heuristic algorithms and then apply the ordering of the moves to solve and obtain the goal state. Such advancement will definitely help us solve this megalithic of a problem.

References

[1] A. S. Forest Agostinelli, Stephen McAleer, "Solving the Rubik's cube with deep reinforcement learning and search," *Springer Nature*, 2019.

[2] R. Brunetto and O. Trunda, "Deep heuristic-learning in the rubik's cube domain: An experimental evaluation." in *ITAT*, 2017, pp. 57–64.

[3] Wikipedia. (2022) Breadth-first search. https://en.wikipedia.org/wiki/Breadth-first_search (11/02/2022).

[4] ——. (2022) A* search algorithm. https://en.wikipedia.org/wiki/A*_search_algorithm (10/02/2022).

[5] ——. (2022) Iterative deepening search. https://en.wikipedia.org/wiki/Iterative_deepening_A* (07/10/2022).

[6] Kociemba.org. (2022) The two-phase algorithm coordinates. http://kociemba.org/math/twophase.htm (11/02/2022).

[7] R. E. Korf, "Finding optimal solutions to rubik's cube using pattern databases," in *AAAI/IAAI*, 1997, pp. 700–705.

Roles of each individual

Rohit Doriya (B20AI034) - Worked upon proposing the problem statement, implementation and methodology of the Kociemba algorithm. Co-developed the entire pipline of the algorithms.

Karan Jain (B20AI016) - Karan did a major job in researching and implementing the BFS techniques and its better version which showed a significance improvement than BFS. he also helped to research different heuristics and implemented them.

Rohit Kumar (B20AI035) - Rohit contributed majorly for developing the A* and IDA* method to solve and helped us to see a broader view of various algorithms. He also help in development of different heuristics.

Priyank Mandal (B20AI055) Implementation and methodology of the Layering algorithm. Co-developed the entire pipline of the algorithms, provided interpretation of the results and conclusion.