GHENT UNIVERSITY

NMO Project – Rubik's cube

Academic year 2023/2024

Àngel Masip Llopis
Matteo Borrelli

Table of contents

1. Introduction

The project was based on solving the Rubik's cube using graph search algorithms. The work focused on writing a Python code that received as input a certain state of a scrambled 2x2x2 Rubik's cube and using search algorithms, including the Breadth-First search algorithm and Bidirectional search, the code provided as output the solved state of the Rubik's Cube and computational parameters that we will analyse later.

Next, we performed a comparative analysis of the two different algorithms and their heuristics to solve the cube by analysing the number of states visited, the number of states generated, the time taken, and the memory used. To have an even more detailed analysis we implemented a heuristic and see if it was possible to obtain an improvement in the parameters. Ultimately, we applied the same search algorithms for the Pyraminx cube and compared the results of the two analyses.

Additionally, to have a visual representation, we created an interface that shows the faces of the Rubik's cube and the movements necessary to arrive at the final solved state, depending on the selected algorithm.

GHENT UNIVERSITY

2. Description of our code for the 2x2 Rubik's Cube

The representation of the Rubik's cube is a string that for every face we have four words, and every word represents a "cubie" that is formed of the colours that are around in a clockwise manner.

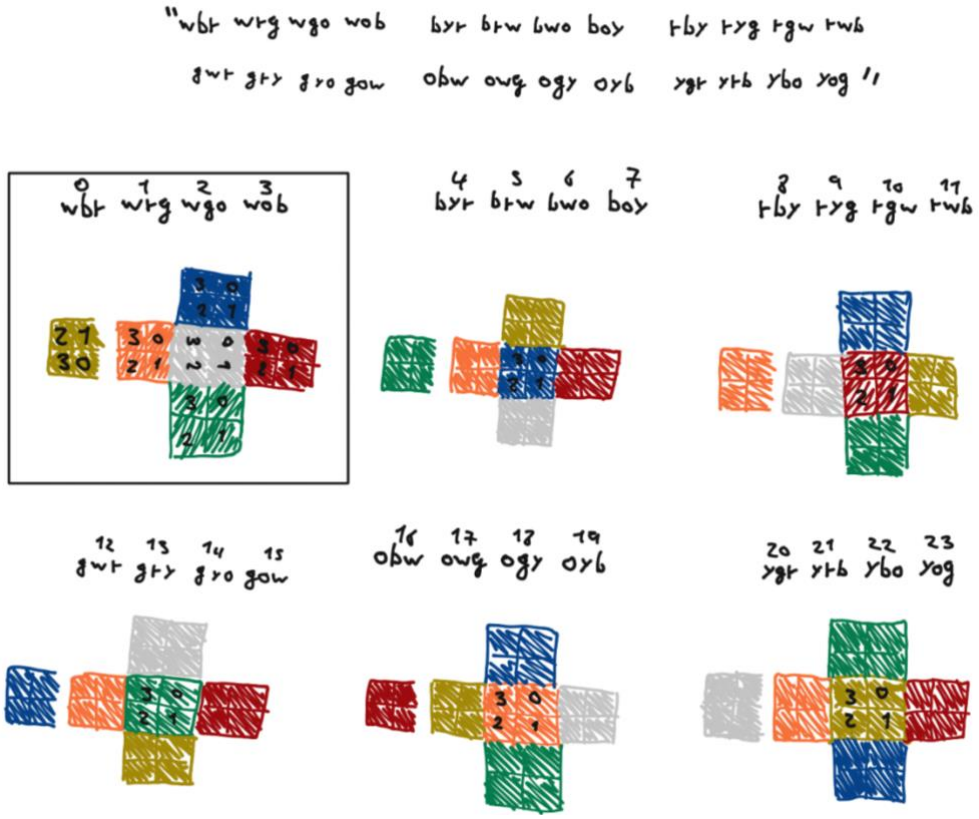

Figure 1: 2x2x2 Rubik's cube representation.

In the preceding discussion, we considered the **BFS algorithm**. However, for the sake of precision, it should be noted that the code in question implements the A* algorithm, an advanced graph search technique that utilises heuristic functions to efficiently find the shortest path from an initial state to a goal state. In this specific implementation, the A* algorithm is adapted to allow for breadth-first search (BFS) capabilities by setting the heuristic (function_h) to zero, indicating that it does not influence the order of node expansion. Furthermore, the cost function (function_g) is set to 1, which ensures that every move has the same cost.

The implementation was designed in such a way that it would be possible to implement a heuristic and compare the performance of the base algorithm and the same algorithm with the heuristic.

The children of a node are generated based on 18 possible moves. Each move modifies the state of the Rubik's Cube, leading to a new node in the search graph. We can see that in the code we have implemented only 6 moves. This is because in order to identify the children of a node, we perform these moves one, two and three times.

Let me explain, in the “getChildren(state)” function, we first identify the six children according to these moves we have implemented (twist of 90 degrees).

Then, if we apply one of these moves three times, we are effectively applying the counterclockwise version of that move (twist of 270 degrees). Similarly, applying this move twice is equivalent to applying the “half” version of that move (twist of 180 degrees). This is the method by which the children of a node can be identified, given that there are 18 possible moves from a state.

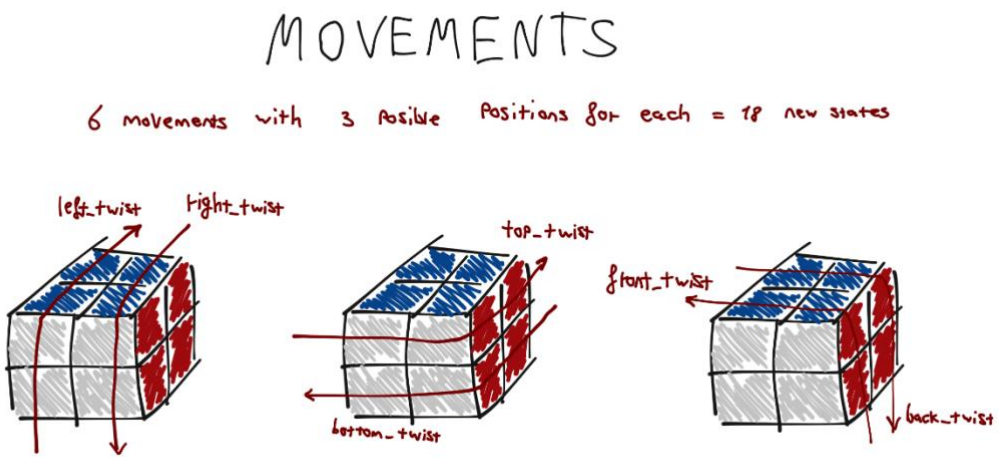

Figure 2: Movements of the cube.

It can be observed that the moves are implemented in a manner analogous to a dictionary. Upon examination of our representation of a state, we can see that these are the positions of the words that must be altered in order to attain the desired state. In other words, the “cubies” that must be modified to achieve the desired state after the application of this move.

In addition to the standard A* implementation, the code incorporates a **bidirectional graph search strategy**. This method initiates two simultaneous searches: one from the initial state forward and the other from the goal state backward. The search concludes when the two searches meet in the middle of the path. This approach can significantly accelerate the search process by effectively reducing the search space that each search front needs to cover. A subsequent analysis will be conducted to ascertain the advantages of utilising this algorithm in comparison to BFS.

The **heuristic function** that we have implemented for the Rubik’s Cube takes as input a string representation of a state, where each segment of the string indicates the colour of a part of the puzzle (e.g., a face of a Rubik’s cube). It then constructs a dictionary where each key is a colour, and the associated value is a list of positions (indices in the string) where that colour appears. For each colour in the current state, it then checks the positions of that colour against its positions in the goal state.

The heuristic cost increases by the number of positions in the current state that do not match any position in the goal state for that same colour.

It can be observed that the number of the heuristic is normalised using the value *3/40. The rationale behind this is to ensure that the heuristic is able to identify the optimal solution. In other words, to ensure that the heuristic is admissible, that is, that it does not overestimate the optimal cost to reach the final state from any given state. The value in question was empirically determined by examining the optimal cost of a state and the value provided by the heuristics.

3. Description of our code for the Pyraminx

The code of the search algorithms is the same as explained before, we of course changed the representation, the moves and the getChildren function. We show now the solved representation for the Pyraminx: '111111111222222222333333333444444444'.

Each number represents a colour, and each position represents a mini triangle.

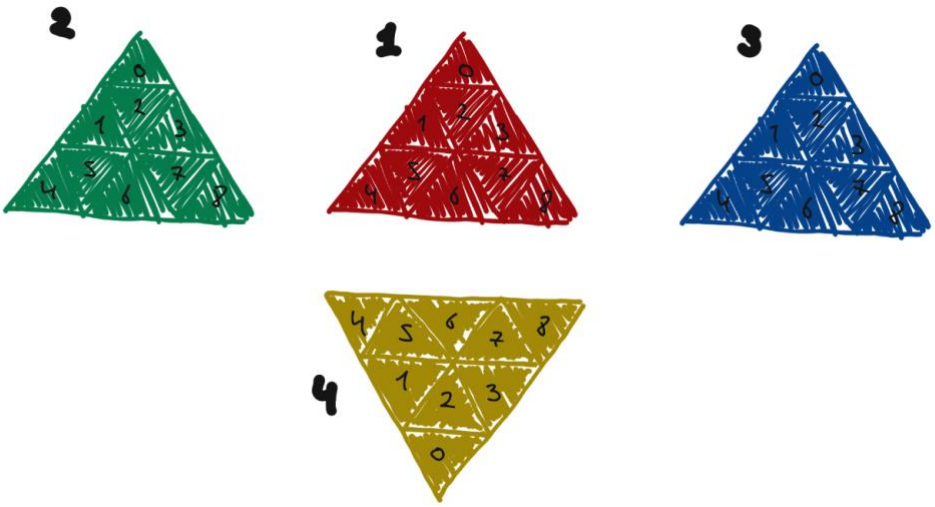

Figure 3: Pyraminx cube representation.

Here we can see the movements:

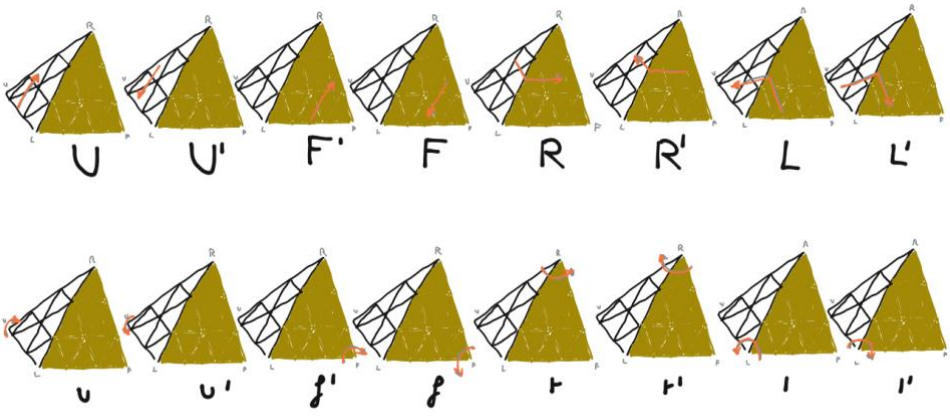

Figure 4: Movements of the Pyraminx cube.

4. Analysis and results for Rubik's Cube

We now move on to the analysis phase of the results obtained from the various resolution tests we have performed.

Our representation, as mentioned, includes 18 movements (half turns in addition to quarter turns) and therefore the maximum reachable cost is equal to 11. The cost refers to the number of turns with which it is possible to solve the scrambled cube. However, we did not find any state with cost 11 in the attempts we did. Indeed, the percentage of states with cost 11 is equal to 0.072%[1]. The parameters present in the table are the following:

- **Graph cost:** the graph cost refers to the number of turns it took to get to the final solved state.
- **Graph counter:** this variable is used to count the number of nodes explored during the graph search. It represents the total of nodes that have been visited and evaluated during the search process.
- **Time graph:** is the time taken by the algorithm to arrive at the final solution state
- **Node counter:** this variable counts the total number of nodes generated or expanded during the graph search. It includes both the nodes that have been explored and those that are currently in the open list (heap).
- **Max counter:** keeps track of the maximum number of nodes that have been explored or are currently present in the priority queue (heap) during the execution of the graph search algorithm. It is updated whenever the sum of explored nodes and nodes in the priority queue exceeds the current value of max counter.
  This counter is useful for determining the peak number of nodes processed at any point during the execution of the algorithm. In addition, max counter indirectly measures memory usage by tracking the peak number of nodes explored or present in the priority queue during the execution of a graph search algorithm. Since each node in the search process typically consumes memory to store state information and links to child nodes, an increase in max counter indicates a higher demand for memory resources. However, it's important to note that max counter doesn't provide a direct measurement of memory usage but serves as an indicator of the maximum load on memory during the algorithm's execution.

After running the code several times to solve the cube, the highest cost we obtained is 9. To perform a comparative analysis between the BFS and Bidirectional Search algorithms, we used costs 5 and 6, since already for a cost of 7 it became too heavy for the BFS algorithm. Therefore, with the tables below, we will analyze the obtained values of the above parameters for costs 5, 6 and 9.

[1] https://en.wikipedia.org/wiki/Pocket_Cube

Table 1: Results from input of cost 5

Parameter	**BFS**	**BFS + Heuristic**	**BiBFS**	**BiBFS + Heuristic**
graphf_cost	5	5	5	5
graphf_counter	24018	746	809	335
time_graphf	2.23s	0.24s	0.06s	0.11s
node_counter	202044	8958	9811	4387
max_counter	178026	8212	9004	4054

For cost 5, we can see that the "Bidirectional + Heuristics" has the smallest number of explored and generated states. This underscores the impact of heuristics in reducing the number of states visited and generated in the execution of the algorithm. Furthermore, it is also the more efficient algorithm in terms of memory, having a max_counter value smaller than half of that generated for simple bidirectional and for "BFS + Heuristics". Although the regular "BFS" has a max counter value four times larger than that of the bidirectional + heuristic, this indirectly confirms the latter algorithm as the least memory-heavy used. Nevertheless, it is not the fastest algorithm since the normal bidirectional search takes a few thousandths of a second less than bidirectional with heuristic (0.06s rather than 0.11s). The heuristic in question is evidently an inefficient means of calculation.

Table 2: Results from input of cost 6

Parameter	**BFS**	**BFS + Heuristic**	**BiBFS**	**BiBFS + Heuristic**
graphf_cost	6	6	6	6
graphf_counter	1049068	5025	3770	1780
time_graphf	111.13s	1.51s	0.26s	0.55s
node_counter	6254415	53797	42141	21390
max_counter	5205347	48764	38373	19612

When comparing Breadth-First Search (BFS) with Bidirectional BFS (BiBFS), both algorithms have the same cost, suggesting that they both arrive at optimal solutions. However, BiBFS shows a significant improvement in efficiency, as evidenced by its significantly lower node count and faster execution time. With a counter of 3,770 nodes and a runtime of just 0.26 seconds, BiBFS clearly outperforms BFS, which explores over a million nodes and takes over 111 seconds. This superior performance also translates into better memory efficiency, as evidenced by the lower maximum count of BiBFS.

Comparing BFS with and without heuristics, the latter shows remarkable efficiency gains. While both versions achieve optimal solutions, BFS with heuristics shows a much lower number of nodes and significantly reduced execution time. The heuristically enhanced BFS explores only 5,025 nodes and takes only 1.51 seconds, in stark contrast to the regular BFS which takes 111.13 seconds and explores over a million nodes. These improvements highlight the effectiveness of heuristic guidance in improving search efficiency.

Finally, a comparison of BiBFS with and without heuristics shows that while both versions achieve similar costs, the heuristically enhanced BiBFS is more efficient in terms of node count and memory usage. Despite taking slightly longer (0.55 seconds versus 0.26 seconds), BiBFS with heuristics explores fewer nodes and has a lower maximum node count, highlighting the benefits of heuristic-based search strategies in optimising resource usage.

Tables 3: Results from input of cost 9

Parameter	**BiBFS**	**BiBFS + Heuristic**
graphf_cost	9	9
graphf_counter	66360	56803
time_graphf	5.14s	15.29s
node_counter	516167	512756
max_counter	449809	455773

For cost table 9, we ran only the simple bidirectional search algorithms and with heuristics, since, as mentioned earlier, solving with the BFS algorithm and BFS with heuristics would have taken too long and probably will result in a MemoryError.

The results confirm bidirectional search with heuristics the best algorithm in terms of fewer states generated and visited. For this cost, we have a lower max counter value for BiBFS, though slightly, than for bidirectional with heuristics, meaning less memory usage.

We note, however, that the difference in the values of graph_counter and, in particular, node_counter is very small between the two algorithms used. While the difference in execution time is significant, the bidirectional search algorithm takes 5.14s compared to 15.29s for the algorithm with heuristics. The heuristic in question is insufficiently complex to address the problem at hand, and its computational efficiency is suboptimal.

5. Analysis and results for Pyraminx Cube

Let us now analyze the results obtained for the Pyraminx cube. The results refer to a cost 7 because a larger depth is intractable for BFS. The larger cost found with the Pyraminx is 14, but we didn't put the results because are similar as the ones above for comparing BiBFS and BiBFS with heuristic.

Parameter	**BFS**	**BFS + Heuristic**	**BiBFS**	**BiBFS + Heuristic**
Graph Cost	7	7	7	7
Graph Counter	247,784	9,093	3,248	1,331
Node Counter	1,428,352	70,361	22,646	10,952
Max Counter	1,206,842	61,268	19,400	9,623
Computation Time	16.11s	0.645s	0.200s	0.113s

Both "BFS + Heuristic" and "BiBFS + Heuristic" show dramatically lower graph and node counts compared to their non-heuristic counterparts. This suggests that heuristic methods significantly reduce the number of nodes expanded, leading to more efficient search processes.
The computation time for heuristic methods is significantly less than that for basic methods. For example, "BFS" took about 16 seconds, while "BFS + Heuristic" took less than 1 second. This difference highlights the impact of heuristic optimisation in reducing the time complexity of graph searches.
"BiBFS + Heuristic" has the lowest computation time of all methods, indicating a highly efficient bidirectional search combined with heuristic optimisations.
The dramatic reduction in node and graph counters in heuristic methods suggests that fewer resources (memory and processing power) are required. For example, "BiBFS + Heuristic" and "BFS + Heuristic" used significantly less resources than "BFS" and "BiBFS".

Comparing BFS with BiBFS, we can see that BFS has a graph counter of 247,784, BiBFS records a significantly lower count of 3,248, illustrating that BiBFS explores far fewer nodes to arrive at a solution. This efficiency is attributed to the bi-directional approach of BiBFS, which reduces the search space by simultaneously initiating exploration from both the start and end states and meeting in the middle. Similarly, the node counter is significantly higher for BFS at 1,428,352 compared to only 22,646 for BiBFS. This metric indicates that BFS performs a more exhaustive search, resulting in lower memory efficiency. Furthermore, the maximum number of nodes stored in memory at any one time (max counter) is also lower in BiBFS (19,400) compared to BFS (1,206,842), indicating better memory management in BiBFS. Computationally, BFS takes significantly longer, taking 16.11 seconds to complete, while BiBFS takes only 0.20 seconds, demonstrating BiBFS's superior efficiency and speed in finding solutions, an advantage in time-sensitive applications. We can see that BiBFS uses over 60 times less memory than BFS and is 80 times faster.

6. Comparing the Pyraminx and the 2x2x2 Rubik's Cube

In both cases BiBFS outperforms bfs in a magnitude order of 20 or 60 times, we clarly see that the heuristic function implemented for the Pyraminx is way more appropriated than the one implemented for the Rubik's Cube. The God number (the maximum number of twists in an optimal solution) for both puzzles is 11 (counting half turns and quarter turns in the Cube and without counting the moves of the tips for the Pyraminx).
The Pyraminx puzzle has 360 possible positions for the six edges and 32 ways to flip them due to parity constraints. Multiplying this by 3 for the axial pieces results in 75,582,720 possible configurations. For the 2x2x2 Rubik's cube every permutation of the eight corners is possible, which accounts for 8! (40,320) positions. Additionally, seven of these corners can rotate independently, providing 3^7 (2,187) different orientations. The orientation of the cube in space doesn't add any new configurations, so there are 24 fewer. This is because there are 24 different ways to orient the cube, with each face being the base and each face having 4 possible rotations. So, there are 40,320*2,187/24=3,674,160 possible configurations.
We can see that the search space is bigger for the Pyraminx, but it gets reduced to 933,120 if we do not take into consideration the rotations of the tips, that's why it can be more challenging for a BFS algorithm than for a human, but we could change the code and have more clever strategies to reduce this search space.

7. Conclusion

In Summary, the project aimed to summarise the use of graph search algorithms in solving the Rubik's cube, with a particular focus on the 2x2x2 Rubik's cube and then the Pyraminx cube.
Further analysis compared these algorithms and their heuristics, evaluating metrics such as the number of states visited, generated, time taken, and memory usage. Additionally, we developed the requisite representation and movement specifications for both puzzles. Results demonstrated the efficacy of heuristics in reducing computational load and enhancing search efficiency. Bidirectional search consistently outperformed BFS, with and without heuristics, showcasing superior efficiency in terms of node exploration, execution time, and memory usage.
Comparing the Pyraminx with the 2x2x2 Rubik's Cube, while both benefitted from Bidirectional BFS, the Pyraminx exhibited a larger search space due to its increased number of possible configurations. At the end, we saw that the heuristic for the pyraminx is better and more efficient than the one for the Rubik's Cube.
Overall, this project not only provided insights into solving Rubik's cube puzzles using graph search algorithms but also shed light on the importance of a good implementation of heuristic functions and that the implementation of BiBFS has led to a significant enhancement in BFS.