

BP2: Applying algorithms to solve a Rubik's cube

Martin Sundman (sundman@ruc.dk, 70454)

Mathias Marquar Arhipenko Larsen (mamaar@ruc.dk, 70434)

Oliver Skjellerup Demuth Heinecke (osdh@ruc.dk, 71604)

Rasmus Beyer Andersen (rbeyera@ruc.dk, 71466)

Gruppe: 4

Vejleder: Line Reinhardt (liner@ruc.dk)

June 19, 2022

Contents

1	Abstract	4
2	Introduction	4
3	Relevance	5
4	Requirements	5
5	Problem definition	5
5.1	What is a Rubik's cube	5
5.2	Data structures	6
5.3	Algorithms	7
5.3.1	Depth first search	8
5.3.2	breadth first search	9
5.3.3	best first search (A*)	9
5.3.4	Iterative Deepening Depth first search (IDDFS)	10
5.3.5	Iterative Deepening A* (IDA*)	11
5.4	CFOP	11
5.4.1	Cross	11
5.4.2	F2L	12
5.4.3	OLL	13
5.4.4	PLL	14
6	Decisions about implementation	15
6.1	Different versions	15
6.2	Heuristic	15
7	Implementation	17
7.1	App	17
7.2	Cube	19
7.2.1	Scrambling the cube	20
7.3	Cubie	21
7.4	Open_node	22
7.5	Solver	22
7.5.1	Opening nodes	23
7.5.2	Fitness	24
7.5.3	Heuristic Estimate	25
7.5.4	Arguments for heuristic	26
8	Tests and results	27
8.1	Heuristic preliminary tests	28
8.2	User interface	28
8.3	CFOP test	30
8.3.1	Test 1	30
8.3.2	Test 2	30
8.4	Data	31
8.5	Time	32
8.6	Memory	32
8.7	Nodes	33

9	Discussion	33
9.1	Heuristic	33
9.2	Algorithm	34
9.3	language	34
9.4	Data	35
10	Conclusion	36

1 Abstract

Algorithms can come in a broad aspect of uses, people are in contact with algorithmic work at all time. It is used in cellphones, computers, cars, etc. An algorithm gives a computer or a car a specific set of instructions, that allows it to complete a specific task, whether it is to navigate by GPS in a car or calculate a random path in a game. This report investigates different kinds of algorithms, and how well they work on a complex object, such as a Rubik's cube. The algorithms searches through the 18 different manipulations that can be applied to a Rubik's cube. The final algorithm determines which move is closest to solving the cube based on a heuristic. The algorithms used in this report comes from the theory of depth-first search, breadth-first search and best-first search. The algorithms are IDDFS, A* 1 and A* 2 that we developed to solve the Rubik's cube. Each algorithm works differently and therefore have different areas where they perform better. For these algorithms we tested average number of nodes visited, average time used, average memory used and average number of moves needed to solve the Rubik's cube in order to find the best one.

The data collected shows that the latest heuristic A*2 outperforms the two other algorithms. The simplest way A*2 out performs the other algorithms is by being able to handle more complex scrambles and not run out of memory due to its heuristic. Further, the memory used to solve a cube with A*2 grows slower than a cube solved with A*1.

2 Introduction

In 1974 there was an architectural professor with the name of Ernő Rubik, who designed a sequential move puzzle, that sparked interest in the scientific community [1]. The puzzle was named '*The Rubik's cube*' and prompted a new interest in solving the cube as quickly as possible within the community of "unprofessionals". Those people would later be referred to as the "speedcubers" [2]. A popular way to solve the cube is a collection of speed-cubing algorithms, known as CFOP. CFOP is an acronym, where each letter is an algorithm used for a part of the Rubik's cube. What we know today as CFOP was collected and published by Jessica Fridrich [3], for more detail see Section 5.4.

In parallel with the development of digital computers, more advanced algorithms were developed in order to maximize the limited computing power at the time. These algorithms have served as stepping stones for the different algorithms in use today and are used to optimize almost everything we take for granted. The problems range from sorting containers in a cargo harbor to using Microsoft Office - Word's spell check. It is important to quantify which algorithms are good for which problems due to the wide range of applications algorithms have. This can be done by defining the difficulty of the problem. For the purpose of our report, we have chosen to examine a Rubik's cube, which is a 3-dimensional color puzzle. The Rubik's cube is good for challenging algorithms since if you manipulate one tile on the cube, you end up manipulating a group of tiles. Moreover, the difficulty of the Rubik's cube has been proven to be in the set of NP-hard problems [1], meaning that it is believed that no polynomial-time algorithm will exist, although it has not been disproven either. Considering all this we have derived the problem:

Which algorithms exist that can solve Rubik's cubes, and how can we implement, test and evaluate which is the best?

In order to answer our question, first it is important to know what a Rubik's cube is, see Section 5.1. Further more, a description of algorithms is made in Section 5.3. Knowledge from these two sections were combined when we developed our program in a Java 17.1

object-oriented program. The program executes a Rubik's cube simulation, scrambles the cube and solves it algorithmically. The program is explained in Section 7. Although during development, some issues occurred which have been addressed in Section 6. Finally, the program is tested and a conclusion is drawn in Section 8 and 10 respectively.

3 Relevance

This report looks at the efficiency of different algorithms, to investigate this we will solve a Rubik's cube through simulation. We have chosen to look at CFOP and an A* based algorithm. CFOP will be executed by hand and compared to the results of the algorithm. IDA* is a depth-first search algorithm and was made specifically for solving a Rubik's cube, by Richard Korf [4]. When deciding what algorithm is the '*best*', we first need to decide what variable we are measuring. Either computing time or amount of moves to solve the cube. Both are relevant as short computing time allows for more data to be collected. An optimal solution for any physical Rubik's cube would be the one with the fewest moves required. Therefore we choose to put focus on the minimal number of moves as the more important factor when deciding which algorithm is best.

4 Requirements

The program constructed should be able to do the following:

1. The program should have some form of representation of a Rubik's cube, whether it be text or graphical.
2. The program should have a way for the user to manipulate the Rubik's cube when the algorithm is not running. This includes an exit, stop, scramble, and all the moves that can be applied to a Rubik's cube, these are listed in section 5.1.
3. The program should be able to solve a Rubik's cube with an A* based algorithm.
4. The program should show what moves have been applied to both scramble and solve the cube allowing the actions taken to be mimicked in real life.

5 Problem definition

5.1 What is a Rubik's cube

The Rubik's cube is, as the name suggests, a cube parted into smaller sub-cubes that we call '*cubies*' where the standard Rubik's cube is referred to as a 3x3x3 ("*3 by 3 by 3*") detailing the dimensions of the cube by the number of cubies. Other cubes exist and can range from 1x1x1, as a collector's item to 9x9x9 as an extra challenge to 1x3x4 for a more obscure scrambled cube. A Rubik's cube is considered solved when all cubies of the same colors are placed on the same side of the cube. These colors, much like the size of a cube, can range from cube to cube, but the original Rubik's cube has the colors: *red, green, yellow, blue, white and orange*. To scramble the cubes you can apply "moves" to the cube, which turn a set of the cubies that make up one side of the cube, all moves are shown in Figure

It is important to note that the center tile of every side of the Rubik's cube stay fixed in relation to the other middle tiles, meaning that red will always be on the opposite side of the cube than orange, just as blue is opposite of green, and white is opposite of yellow. It is also important to note that the neighboring colors, just like the opposing colors, also

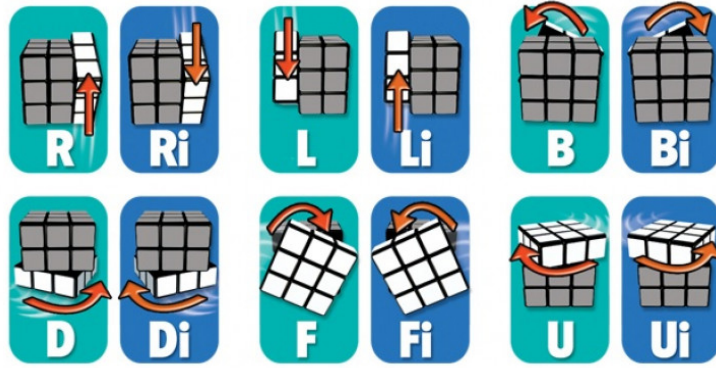


Figure 1: Moves that can be applied to a Rubik's cube

do not change. Therefore, by knowing the colors of two center tiles, you can know the color of every other center tile of the cube. It has been proven that, the most complex scramble on a 3x3x3 cube, at most requires 20 moves to solve it [1]. The importance here is that we need to note which moves are legal because if we want to use the same move boundary we need to allow the same moves. Therefore note, within the 20 move boundary the standard 12 moves as shown on Figure 1 and super turns, i.e. 180° turns, are allowed. Since there are six possible super turns, we end up with 18 moves, including the 12 normal moves as shown in Figure 1.

Meaning that all together we are looking at 18^{20} or $1.2748236 * 10^{25}$ different states the cube can have, meaning that we must test an equal amount of scenarios. For a frame of reference, it has been estimated that there are about $1 * 10^{21}$ stars within the observable universe [5].

5.2 Data structures

We need to somehow represent a Rubik's cube in a data structure. The cube consists of 26 cubies separated in to 3 types: 8 corner cubies, 12 edge cubies and 6 center cubies. To represent the cube the data structure needs to keep track of each cubies data. This includes: type, colors, orientation, and its placement relative to all the other cubies. Their type and color tell how many tiles the cubie has and which colors they are, the orientation tells which direction the tiles point. We have considered and/or tried 3 approaches for representing the cube.

1. Each tile on the cube is saved in a 54 item array since there are 9 tiles on each side of the 6 sided cube. These tiles are then switched around according to what "moves" are allowed. However, this was slow in run-time and tedious to program.
2. Each cubie of the cube is made as a smaller cube. This cube can have values on its sides to represent colors or null to represent no color. This is then saved in an $3 \times 3 \times 3$ array to keep track of positioning. Since everything is sorted in classes it makes it easier to apply "moves", however there is an issue with the orientation, since the orientation of the edge cubies which have two colors and the corner cubies which have three colors are defined differently.
3. Two arrays, one for corners and one for edges, each cubie has an identifier corners 0-7, edges 0-11, and a representation of correct or incorrect orientation [6].

Approach 1 is not a great structure because cubies are not guaranteed to stay consistent, since tiles are independent and not tied together by more than their placement in the array, as well as moves become more convoluted to code. Approach 2 is easier to code and interpret since it is a one-to-one real to code representation of a cube.

However, with approach number two a lot of memory is used in the algorithm, with many objects and arrays. Because the algorithm opens nodes containing a series of moves we call a path, more information in Sections 7.4 and 7.5. The way the algorithm saves every state of the cube is by storing a path leading to that state. This unfolded a third issue with our program: when running the algorithm, it was not an uncommon occurrence that it would use an excessive amount of memory. This is because when it comes to storing a String, which was the datatype the moves were saved as at first, in the memory of a computer, the computer allocates 16 bytes of memory per string. So every node contains an array which uses up 16bytes for every depth the algorithm is in, since it uses one string for every move it has applied. This multiplied by the amount of moves allowed, 18, to the power of the depth, to see how many nodes have been opened at this depth. Suddenly this piles up and ends up using an enormous amount of memory. To solve this we decided to use an 8bit number instead of a string to symbolize the moves that can be applied to our cube. This ensures that instead of 16Bytes for every element in the path array, the algorithm uses only 1 Byte. By doing so, it seems that we have about halved the memory usage for the algorithm. The decision to change the way moves were stored from String to byte in turn changed the data type of `path` from String array to byte array. This significantly reduced memory use of the algorithm. The third approach could further decrease the memory usage because no objects are made and all data on color and orientation is saved as bytes, thus much less memory intensive.

5.3 Algorithms

To find the solution of a cube it is important to understand that this is a search-problem. This clarifies how a problem like the Rubik's cube can be tackled. To decide which approach to take, consider the case where the program looks through every state the search problem can be in and check if that is the solution state. In the case of the Rubik's cube the number of states, as mentioned in Section 5.1, is quite big. This is where algorithms are applied, to minimize the search time. To simplify advanced problems such as the Rubik's cube, something called a state tree is made. A state tree simply contains all states the search algorithm can run into, connected by the decisions needed to take to get to said state. The states in such a tree are called *nodes* and are symbolized by a circle, and the decisions are called *branches* connecting said circles, see example in Sections 5.3.1, 5.3.2, and 5.3.3.

To cut down on search time something called a heuristic can be applied. The formula for a heuristic search is $f = g + h$ where g is the cost used to reach the current state from the start state, and h is the estimated cost needed to get to the solution state [4]. In some state the cost is defined as a different value depending on what decision might be taken, but in the case of a Rubik's cube, everything has a cost of 1 move. Meaning that g , in our case, is the amount of moves already applied, and h is the estimated amount of moves needed to solve the cube. Note that a heuristic is only applied in some algorithms and therefore is only used in Sections 5.3.3 and 5.3.5.

5.3.1 Depth first search

The first algorithm to evaluate is *depth first search* (DFS). A DFS algorithm will look through the state tree by looking in depth first, hence the name of the algorithm. In Figure 2 it can be seen as an example of how DFS will search through a binary state tree. Where A is the start state of the search problem. Here the algorithm will pick the right most choice until the end of the tree is reached [7]. In Figure 2 it is shown as the arrows going from A , to D . When reaching the bottom of the search tree, the algorithm will then re-pick the latest decision making us change from D to E . If E is not a solution state, the algorithm will then go back and re-pick the next decision. Since Figure 2 is a binary tree, there are no more decisions in node C , therefore the algorithm will go back and pick the decision from node B , making the algorithm jump from state E to state F . This will then repeat until either a solution state is reached, the whole tree has been searched through or the algorithm is manually stopped [7]. Note that in the case where D is a solution state, the algorithm would stop and show that as the solution. Now if F is also a solution state this would be a problem, since F is earlier in the tree, making it a *better* solution. Although since the algorithm would not have tested state F , that solution would not have been found. Therefore, with DFS it is impossible to know if the solution the algorithm has found is the best, or even good.

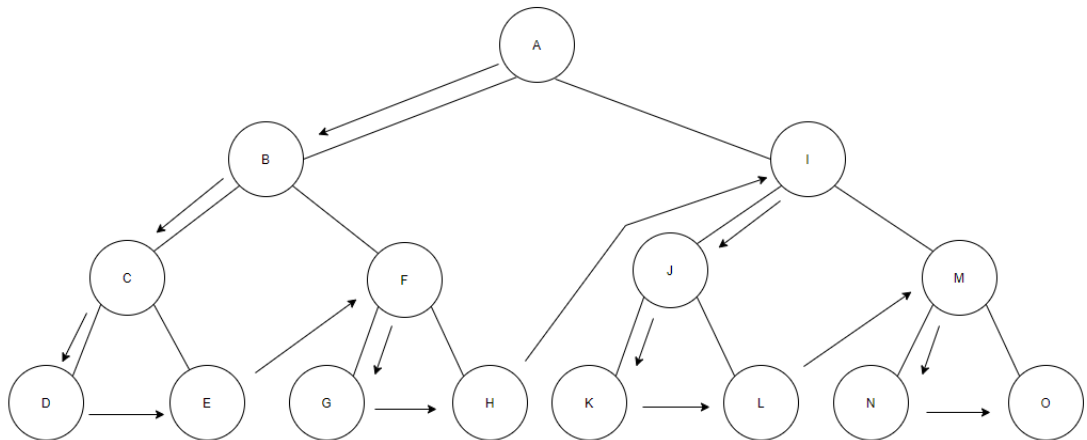


Figure 2: Figure showing an example of how DFS will iterate through a binary state tree. The arrows indicate what path the algorithm will take

5.3.2 breadth first search

To avoid the issue of missing better solutions, another algorithm can be used, such as *breadth first search* (BFS). This algorithm will search through the width of the tree first, meaning that it will search through the earlier depths of it first [8]. In Figure 3 an example of how BFS will search through a binary state tree, where A is the start state. In Figure 3 the algorithm picks the first choice, like DFS, although unlike DFS, BFS will then re-pick if the first choice was not the solution state, meaning that the algorithm will leap from state B to state I . If state I is not the solution state, BFS will go back to the first decision in state B and test C . Further more if C is not the solution state, the algorithm will then continue to the next decision in state B , leading the algorithm to state F , see Figure 3. This will then repeat until a solution state is found, the whole tree has been searched through or the algorithm is stopped manually [8]. This avoids the issue that DFS had, since we know that all nodes in an earlier depth have already been tested and were not the solution state. This means that when a solution state has been found we know that only equivalent or worse solutions can exist.

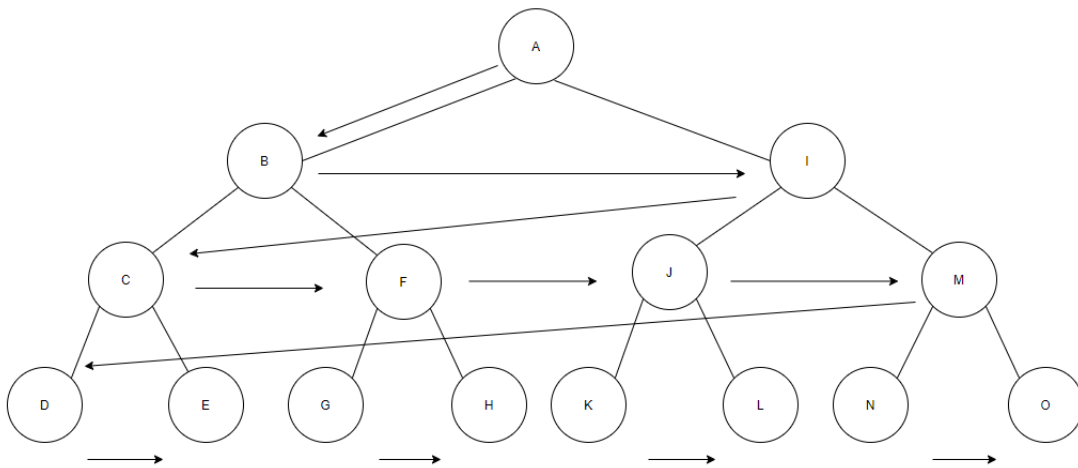


Figure 3: Figure showing an example of how BFS will iterate through a binary state tree. The arrows indicate what path the algorithm will take.

5.3.3 best first search (A^*)

There is an issue with BFS and DFS in Sections 5.3.2 and 5.3.1 respectively. This issue is that both algorithms can end up searching through the whole state tree if the solution state is found at the end of the tree, which in the examples shown in Figures 3 and 2 would be in state O . For a more targeted search, algorithms with heuristics were developed. A heuristic estimates how far from the goal state any given state is, assigning a value representing the distance. One of such algorithms is A^* , which uses said heuristic search to judge what decision is 'best' [9]. In A^* the main objective of the algorithm is to find a solution state with the lowest heuristic value possible. This would mean that both g and h , in $f = g + h$, should be as low as possible [9]. In Figure 4 it can be seen how A^* would search through a binary state tree, with the heuristic of each node written above said node. Since A^* needs a low heuristic, in the first decision it can pick between $B, h = 6$ and $J, h = 8$, making the algorithm chose B . These two nodes are referred to as open nodes, since those are the nodes that are open for decision for the algorithm. When a node has been chosen, it is then closed. Meaning that on the next decision, the algorithm will be able to pick between: $C, h = 5$, $F, h = 4$, and $J, h = 8$. By leaving earlier choices open for the algorithm, it is allowed to go down a different branch, if the current branch turns out to not be as good as first estimated; i.e. a local minimum [9]. This also means that in the next case where the algorithm has

to pick a node at the state F , the algorithm would pick C , since its choices are: $C, h = 5$, $G, h = 7$, $H, h = 9$, and $J, h = 8$. This will then continue until the algorithm has found the goal state or is manually turned off. Since the efficiency of A* is heavily dependant on its heuristic it is important to note that if the heuristic is inefficient, an A* algorithm can also end up searching through the whole tree. Although this should not be the case. The issue with A* is that when running this algorithm on bigger state trees, the algorithm ends up having a lot of open nodes, meaning that it requires a lot of memory to be able to save all the choices that it has.

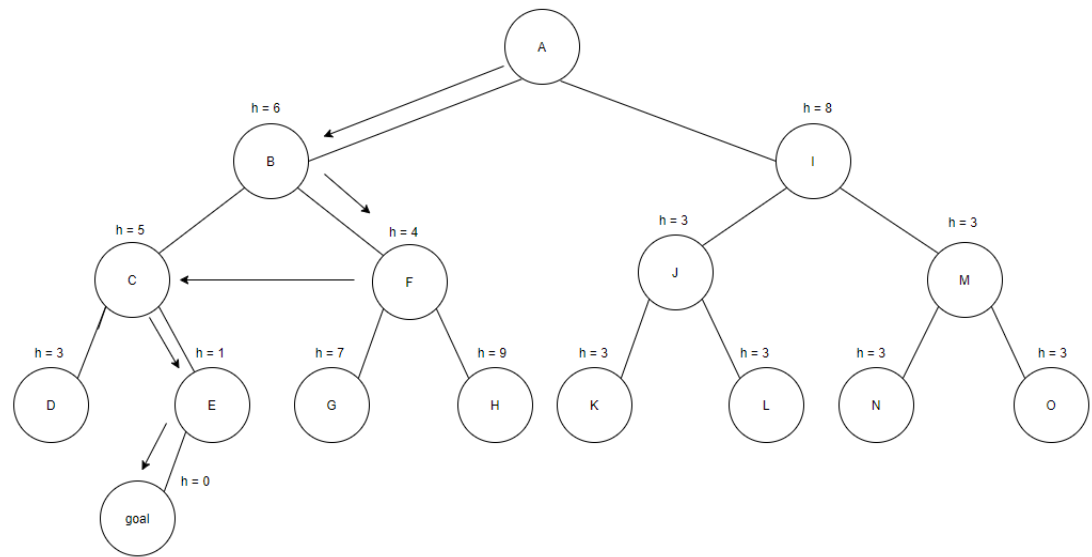


Figure 4: Figure showing an example of how A* will iterate through a binary state tree. The arrows indicate what path the algorithm will take.

5.3.4 Iterative Deepening Depth first search (IDDFS)

To work against the issue of memory, an algorithm that iterates through depths has been created. The algorithm known as *iterative deepening depth first search* (IDDFS) [4] is an algorithm that has this trait. By using iterative deepening, IDDFS achieves a kind of pseudo breadth first. In Figure 5, the search of IDDFS through a binary state tree is shown. Here the different colors indicate a depth iteration and the start state is A . To start of with the algorithm acts like a normal DFS, although it ignores nodes that are deeper than the depth threshold, which is 1 at first. This is seen by the blue arrows. When the tree has been searched through are the given threshold, the threshold is expanded by one. This can be seen by the yellow arrows. When the tree has been searched through at threshold 2 and the solution has not been found, the threshold will once again be increased by one. Shown as the red arrows. By searching through the tree with IDDFS you eliminate the possible issue of a better solution being at an earlier depth, that DFS has.

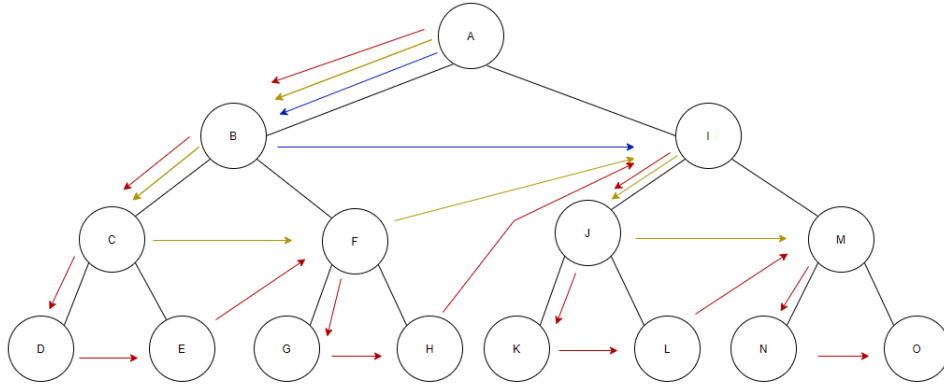


Figure 5: Figure showing an example of how IDDFS will iterate through a binary state tree. The arrows indicate what path the algorithm will take. Here the colors blue indicates a threshold of 1, yellow a threshold of 2, and red a threshold of 3

5.3.5 Iterative Deepening A* (IDA*)

Finally comes the algorithm called *iterative deepening A** (IDA*). This algorithm was also called the *korf* algorithm, since it was developed by Richard Korf and was made specifically to solve the Rubik's cube [4]. This algorithm was made to iterate through depths, like IDDFS, but also apply a heuristic like the A* algorithm. This allows it to find the solution state of the cube at the time complexity of A*, but only use the memory complexity of DFS, since all open nodes that have been opened in previous depths, are closed. The difference with IDA* from a normal A* is that when it reaches the bottom of the tree, the algorithm does not iterate back to an earlier step, the threshold is simply expanded while also closing all the previously open nodes [4]. To find a heuristic that can efficiently find the solution of a cube Korf proposed that by using a database that has every state a cube can be in, this database can output the number of moves any given state the cube is in, is from the solution state[10].

5.4 CFOP

CFOP as mentioned in the introduction, Section 2, is a collection of algorithms that can systematically solve a Rubik's cube. The name CFOP is an acronym for Cross-F2L-OLL-PLL which is the order to solve the cube. The four steps will be explained in further detail below.

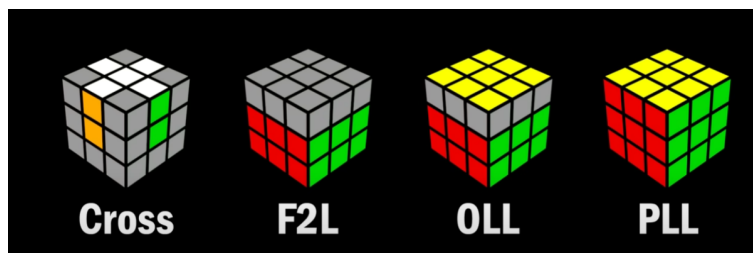


Figure 6: Order in which the cube is solved in CFOP [11]

5.4.1 Cross

When solving a Rubik's cube the very first thing you want to achieve is the cross as shown in Figure 6. There are two different situations in this step that needs to be accounted for. In the case of a cube where the white side is where we want to create our cross, the first situation would be where the color of a middle cubie corresponds to the color of the centerpiece of one

side as shown in Figure 7. Here the first step is to align the two corresponding colors with each other and then turn the side so the white tiles connect. In the example in Figure 7 the actions illustrated are **D** and then **180° R**. Where the red face is considered as the front of the cube.



Figure 7: The making of one side of the cross [12]

The other main scenario possible for making the cross is when the color of the cubie and the centerpiece does not correspond, where you would have to make a slight alteration by doing a **Fi** and one **R**. This can be seen in Figure 8



Figure 8: The incorrect making of one side of the cross [12]

5.4.2 F2L

The method F2L is used to solve the two bottom layers of the cube. It fills in the missing spots between the cross' "arms". There are many ways to optimize this, with something called *'look ahead'*, where as the name suggests you anticipate your next move prior to making the first move. In this section, we will only be looking at beginner F2L, which is the most bare bones version of F2L there is. First, since the cross has been made on the white side, we look for a corner cubie that contains a white tile. This cubie contains two other colored tiles, that correspond to two sides of the cube. From that, you now know the location and orientation of the correct position of the cubie. Next, we find a cubie containing those two exact colors, for example, green and red. If you orient the cube so that the cubie with the white side is on the bottom left of your vision, as shown in Figure 9. From here we place the cubie with the green and red color on the top layer of cubies and rotate it so it is to the right or left of the corner cubie we are working with the **U** or **Ui** operation. The position of the two-colored cubie is dependent on its orientation of it. If the green is facing up and the red is facing sideways, the cubie must be placed above the red center tile and the other way around if the colors of the cubie are switched. If we consider the case where the red is facing the solver, the following sequence is to be applied **{U, R, Ui, Ri, F, Ri, Fi, R}**. This will make the cube go from the left picture to the right picture in Figure 9

Another scenario that the cube can be found in is where the cubie with two colors needs to go down and to the left, whereas in the example earlier in Figure 9 the cubie needed to go down to the right. Meaning in the new case the red and green cubie would be turned so that the red color faces up and green faces the side of the cube. The way to handle this is like the former algorithm, just mirrored. Meaning that when the cubie is above the corresponding color the algorithm would be **{Ui, Li, U, L, Fi, L, F, Li}**. Knowing this we can now solve the first two layers of the cube.

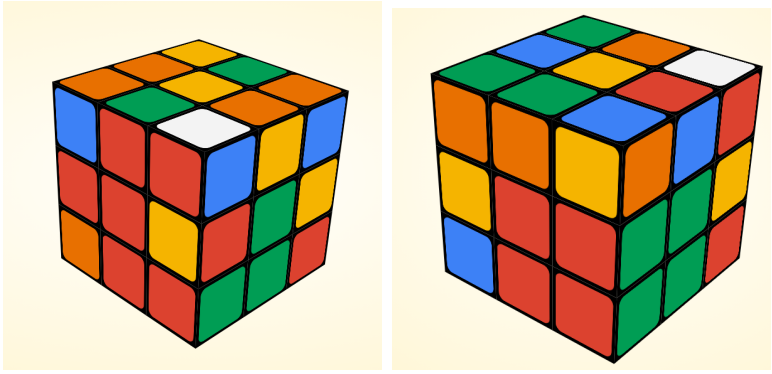


Figure 9: Filling the cross, F2L. Here the green and red cubie has been moved from the top layer to in-between the red and green center tiles

5.4.3 OLL

OLL, also known as Orientation of the Last Layer is a step that orients the cubies correctly. The goal of OLL is to place every color that is supposed to be on every top tile on top. In the case we have been running until now, we want the top to be yellow. OLL contains many of algorithms and a lot of cases and every cube is a different case.











2 LOOK OLL	Edge Orientation		
	1. Opposite		$F (R U R' U') F'$
	2. Adjacent		$f (R U R' U') f'$
	3. None		$[F (R U R' U') F'] [f (R U R' U') f']$
	Corner Orientation		
	1. Sune		$(R U R') U (R U2 R')$
	2. Anti-Sune		$(R' U' R) U' (R' U2 R)$
	3. Car		$F (R U R' U') (R U R' U') (R U R' U') F'$
	4. Blinker		$[f (R U R' U') f'] [F (R U R' U') F'] (R U2) (R2 U') (R2 U') (R2 U2 R) \text{ (faster)}$
	5. Headlights		$(R2 D) (R' U2) (R D') (R' U2 R')$
	6. Chameleon		$(r U R' U') (r' F R F')$
	7. Bowtie		$F' (r U R' U') (r' F R)$

Figure 10: OLL algorithms for different cases [12]

In Figure 10 we see the different cases and what to do when in-counteracting them. The cube in Figure 10 is shown from above while the gray tiles symbolizes any color that is not yellow and the small yellow marking on the sides of the gray tiles symbolize where the yellow tile is on that cubie. Note that in Figure 10 an inverted move is denoted with a "''", where our notation is noted with an "i". This makes it so that the moves $\{R_i, L_i, F_i, U_i, D_i, B_i\}$ will be denoted $\{R', L', F', U', D', B'\}$ Some of these cases do not lead to a fully solved OLL step such as the three first cases, who might lead to each other in some cases.

5.4.4 PLL

Finally, PLL or Permutation of the Last Layer is the step where we place the cubies at their correct location and after performing OLL the orientation of the cubies should all be correct, after this step the cube should be solved. Very much like OLL, PLL has many cases and is shown in the same way as OLL in Figure 11.

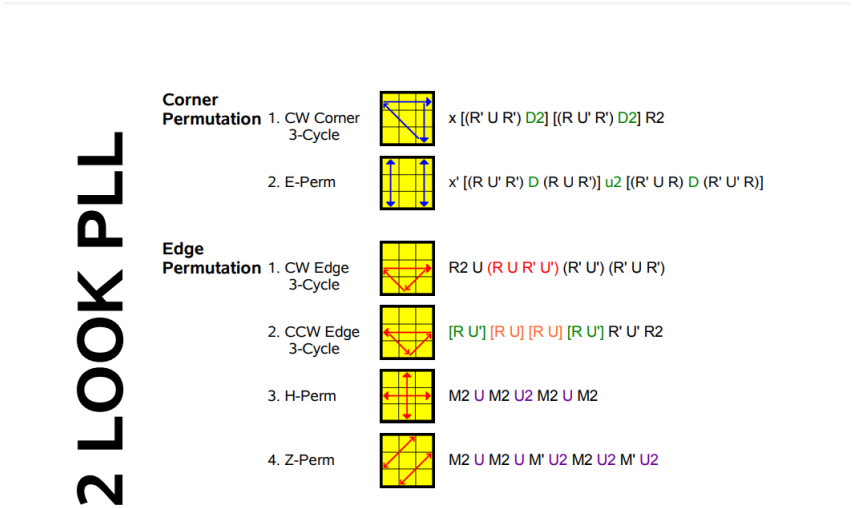


Figure 11: PLL algorithms for different cases [12].

In Figure 11 we see arrows showing which cubies switch places, with one major difference from OLL. In PLL a move that this report does not allow is used, M2. M2 is the move where the middle layer on the cube is moved 180°, luckily that move can be rewritten to {2x 2L, 2R}. Here is another unfamiliar move, but it simply means to rotate the cube. This can be seen in Figure 12, where the cube shown at the top is the starting position and the cubes below have been affected by the given move.

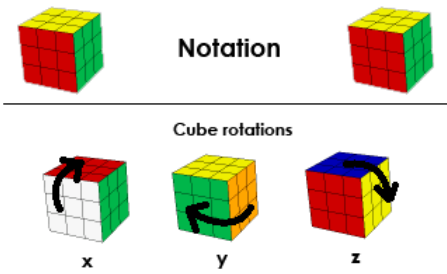


Figure 12: Rotation of cube in the x, y, and z direction [12].

6 Decisions about implementation

During this project we have had many issues and many discussions that needed solving. Because of that our program and our algorithm has grown and changed throughout the time of writing this report. In this section we hope to explain these issues and how we came up with a solution to the given issue.

6.1 Different versions

In Section 5.2 we explained that we tried two different data structures and that there were upsides and downsides to them. The main difference in them were how we represented the cube in the program. In version no. 1 we used an array that was 54 long, where each element in the array represented one tile on the cube. This method quickly showed it self to be bothersome, because of the way changes are applied to the cube. A way to imagine it is to think of a normal Rubik's cube, but instead of turning one side, you would take off all of the stickers for that side and glue them back on to the Rubik's cube again in the new positions. Because of this 'sticker change' way of manipulating the cube, we often ran into small errors, making debugging the program very cumbersome. To counter this issue, we changed the cube to be described with objects.

Which brings us to our next structure. The next structure we decided to use is where the cube is represented as a 3D-array (3x3x3) containing 27 elements, where every element in this array represents a cubie. Since a cubie can have a color on every side on a cubie, we can define the colors to be on the right position of any given cubie, making the cube. This solved the issue we had before of making mistakes with positions, because all of the sudden the colors are not bound to a position in the cube array, they are bound to a side of a cubie. Later when the heuristic needed to be implemented, we ran into our next big obstacle. How is the algorithm supposed to know, what state of a cube is better than another? We came up with the solution that the fitness of a state should be decided by the amount of cubes oriented correctly + the amount of cubies that are on their correct position. The issue with this is that corner cubies and edge cubies both have different orientations. As explained in Section 7.5.3, a corner cubie has three different orientations it can be in, while an edge cubie only has two. Because of this we ended up changing our structure to a third and final structure. The final structure ended up with splitting the cubie into two different forms being, `Edge_cubie` and `Corner_Cubie` while still with the same array just containing two sub classes of the `Cubie`.

Three different algorithms where developed. First an IDDFS, see section 5.3.4, which brute forces its way to the solution by checking each combination of moves until it finds a solution. The second and third algorithms are both A*, referred to as A*1 and A*2 respectively, the only difference between the two is their heuristics. A*1 was the first attempt at a working heuristic, it simply looked at how many cubies were in their correct position and correct orientation. This however, only checks what level of solved a given state of the cube is. Allowing the program to run into local minimums which can cause a lot of computer time. A*2 instead gives an estimate of how many moves are required to solve the cube from the given state of the cube. The estimate is calculated as the sum of moves needed to get each cubie to their correct position divided by 5, this proved to be a better approach, for in depth explanation see Section 8.

6.2 Heuristic

The issue with the heuristic described above is that it did not estimate the fitness of a given state very well. In fact this was almost no better than having no heuristic at all; meaning

searching through the full tree of states the cube can be in. This suggests that our heuristic needs to be changed. In '*Analyzing the Performance of Pattern Database Heuristics*'[10] it is explained that by using a database containing all states a cube can be in. The algorithm can estimate how many steps the current state of the cube is away from being solved. Regrettably we have no such database and therefore a more crude approach to our heuristic is needed. In [13] however, it was suggested that for a more simple heuristic, it is also possible to calculate the distance, in moves, for every cubie independently of the other cubies, for that cubie to be on its correct orientation and position. Finally to add up this for all the cubies and divide by 5 to get our final heuristic. The division of 5 is explained in 7.5 This means that our heuristic should be 0 when the cube is solved, making our fitness the same as the amount of moves used to reach this solved state.

7 Implementation

The program is object oriented programming built in Java 17. To make a program that can solve a Rubik’s cube in object-oriented programming, first, we need to separate different functionalities into different classes. The classes chosen are: Main, App, Cube, Cubie, Open_node, and Solver One method was also taken from the internet, runGC [14], which calculates the memory usage of the program. Also, note that the class Cubie is a superclass and is extended by two subclasses; Edge_cubie and Corner_cubie. Below you can see an UML diagram showing the connections between the classes:

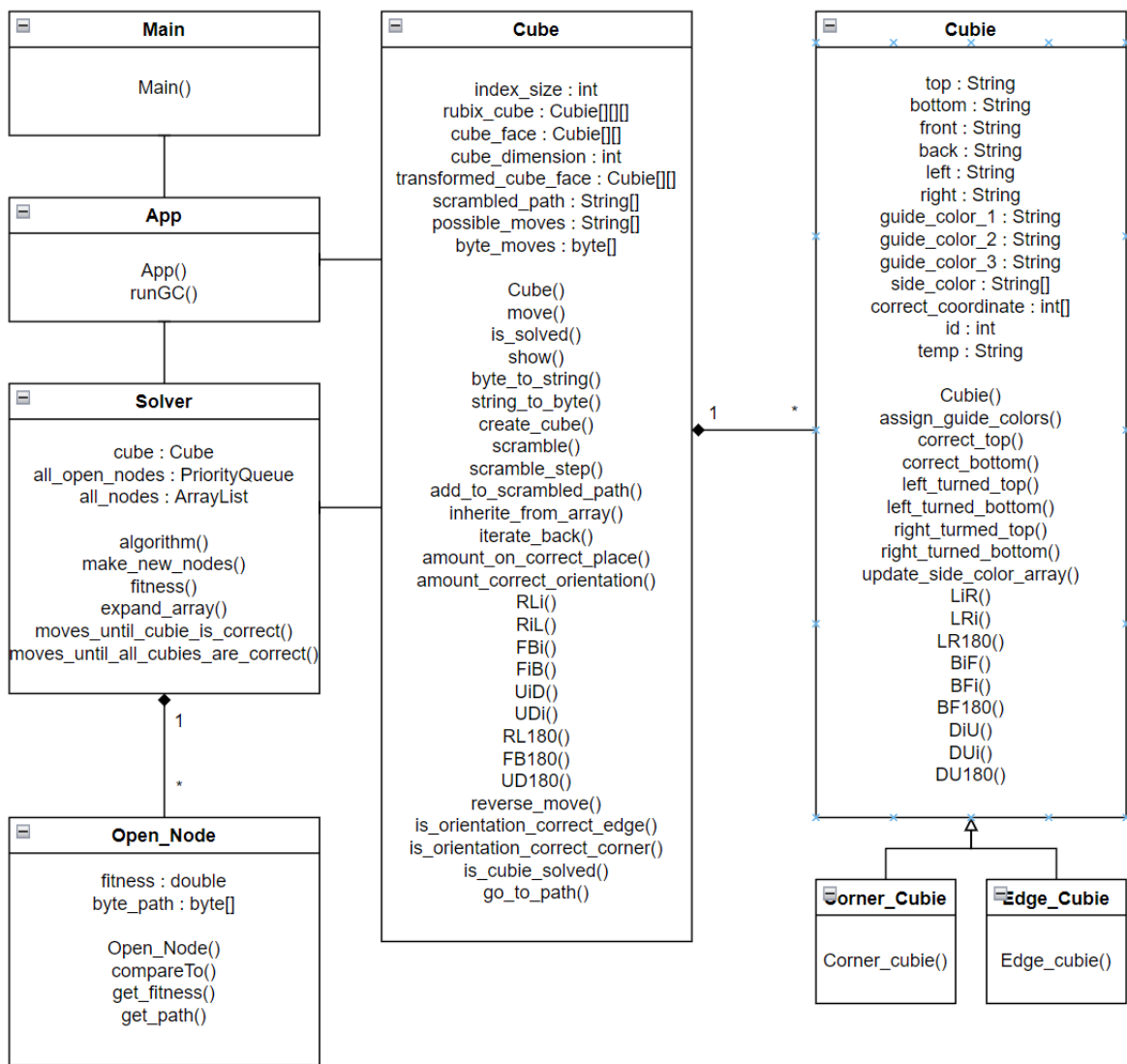


Figure 13: UML diagram

7.1 App

The App class contains the main loop and nothing else apart from a few helper methods. If you go through the constructor of App, which can be seen in the figure below, we start by seeing the making of the cube and solver object along with a few variables. After the initiations, the While loop begins and continues until a user fed command tells it to stop. If the *solve* command is given then **solver.algorithm()** is called, the method is explained in section 9.2. briefly explained **solver.algorithm()** applies our algorithm to the cube to solve it. To monitor how much memory is used in solving the Rubik’s cube we call a method runGC, which is one of the previously mentioned helping methods in App, and finally to end the solving the program returns the Boolean value **automatic = false**. If **automatic** is set to **false** from the beginning we enter the else statement that contains the different user inputs we have allowed. This user input is saved in the variable, **command** and is later

compared to the allowed inputs, among those being: "help", "stop", "solve", "scramble", and any move on the `possible_moves`. To read the input correctly we use the package `Local` to *trim* any white spaces the user might have put around the command and to set all of the letters in the input to *lowercase*. If the command "stop" is received, the Boolean `running` will be set to `false`, meaning the program will stop, while if the command "scramble" is called, the user will be asked how many moves will be applied to the move and print the moves applied. "help" is there to give the user a birds-eye view of the available commands, showing all commands including every move that can be applied. Finally, if the method "solve" is called the Boolean `automatic` will be set to `true`, making it so that on the next iteration of the `While` loop the cube will be solved. Note that if the input does not get recognized by our system the program will output "Not a valid command!".

```

1 public App() {
2     Scanner sc = new Scanner(System.in);
3     String command;
4     boolean running = true;
5     boolean automatic = false;
6     cube = new Cube();
7     solver = new Solver(cube);
8     // main while-loop
9     while (running) {
10        System.out.println("----- ... -----");
11        cube.show();
12        if (automatic) {
13            System.out.println("Solution moves:");
14            System.out.println(Arrays.toString(solver.IDA_step(new
15                Open_Node(20, new String[]{}), 20)));
16            runGC();
17            automatic = false;
18        } else {
19            System.out.print("Write a command: ");
20            command = sc.nextLine();
21            if (string_in_array(Cube.possible_moves, command)) {
22                cube.move(command);
23                System.out.println("is it solved: " + cube.
24                    is_solved());
25            } else if (command.trim().toLowerCase(Locale.ROOT).
26                equals("stop")) {
27                running = false;
28                System.out.println("Loop ended");
29            } else if (command.trim().toLowerCase(Locale.ROOT).
30                equals("solve")) {
31                automatic = true;
32            } else if (command.trim().toLowerCase(Locale.ROOT).
33                equals("scramble")) {
34                System.out.println("how many scrambles? ");
35                int scramble_amount = sc.nextInt();
36                System.out.println("scramble moves:");
37                System.out.println(Arrays.toString(cube.scramble(
38                    scramble_amount)));
39            } else if (command.trim().toLowerCase(Locale.ROOT).
40                equals("help")) {
41                System.out.println("List of commands:");

```

```

35         System.out.println("stop, help, scramble, solve, R,
                                Ri, R180, L, Li, L180, F, Fi, F180, B, Bi, B180
                                , U, Ui, U180, D, Di, D180");
36     } else {
37         System.out.println("Not a valid command!");
38     }
39 }
40 }
41 }

```

7.2 Cube

In the Cube class we maintain all the methods used to draw and manipulate the cube. Because there are 18 manipulations that we can apply to the cube this class is naturally long. To start by taking a look in the constructor of Cube, shown below this text segment, we first define the size of the cube, a helping variable called "index_size", what a cube face is and a helping variable for a cube face. Finally we construct the cube with `create_cube()`.

```

43 public Cube() {
44
45     cube_dimension = 3;
46     index_Size = cube_dimension - 1;
47     cube_face = new Cubie[cube_dimension][cube_dimension];
48     transformed_cube_face = new Cubie[cube_dimension][cube_dimension];
49     create_cube();
50 }

```

Now to take a closer look at `create_cube()` method, code shown below. The first lines of the method are used to define variables. Then the 3-dimensional Cubie array which represents the Cube is initiated. Each Cubie is assigned their colors when they are initialized. This is done in the triple nested for-loop where we define the top, bottom, right, left, front, and back side as either a color ranging through {red, orange, yellow, white, blue, green}. Finally in the bottom of the method contains an if-else-statement where the different cubies are assigned if they are either an `Edge_Cubie` or a `Corner_Cubie`. Finally in the end we have an ID that goes up by one for each cubie. Keep in mind that the code below is only an outline and is missing some components to be correct in syntax. The real code is in the attachments.

```

51 public void create_cube() {
52     String top, bottom, front, back, left, right;
53     int[] coord;
54     int id = 0;
55     rubix_Cube = new Cubie[cube_dimension][cube_dimension][
        cube_dimension];
56
57     for (int index1 = 0; index1 <= index_Size; index1++){
58         if (index1 == 0){
59             bottom = "WHITE";
60         } else if (index1 == index_Size){
61             top = "YELLOW";
62         }
63         for (int index2 = 0; index2 <= index_Size; index2++){
64             if (index2 == 0){

```

```

65         back = "GREEN";
66     } else if (index2 == index_Size){
67         front = "BLUE";
68     }
69     for (int index3 = 0; index3 <= index_Size; index3++) {
70         if (index3 == 0) {
71             left = "ORANGE";
72         } else if (index3 == index_Size) {
73             right = "RED";
74         }
75         coord = new int[]{index1, index2, index3};
76         if ((amount_of_1s(coord) >= 1)) {
77             rubix_Cube[index1][index2][index3] = new Edge_Cubie
78                 (top, bottom, front, back, left, right, id,
79                 coord);
80         } else {
81             rubix_Cube[index1][index2][index3] = new
82                 Corner_Cubie(top, bottom, front, back, left,
83                 right, id, coord);
84         }
85         id++;
86     }
87 }
88 }
89 }
90 }

```

7.2.1 Scrambling the cube

The cube is scrambled by randomly selecting moves and applying them to the cube. The process of scrambling is done with 2 methods `scramble()` and `scramble_step()`. We chose to separate the process because we wanted to assure that the scramble was good. By good we mean the same move is not repeated in succession, and a move is not followed by an inverse of the move. This ensures the cube is properly scrambled. When `scramble()` is called it is given a number of scramble moves to apply. First, it checks if the cube has been previously scrambled or not if it has, then the last move is assigned to the variable `previous_move`, which is used in `scramble_step()` to ensure a good scramble as described before. `scramble()` has a for loop that iterates a number of times equal to the scramble moves the method is told to apply. In the loop first the random move is decided by `scramble_step()` then it is applied to the cube, the move is stored in an array called `scrambled_path()`, then the new move is saved as `previous_move`.

```

86 public String[] scramble(int amount_scrambles){
87     String previous_move= "";
88     if (scrambled_path.length > 0){
89         previous_move = scrambled_path[scrambled_path.length - 1];
90     }
91     for (int move = 0; move < amount_scrambles; move++){
92         String current_move = scramble_step(previous_move);
93         move(current_move);
94         add_to_scrambled_path(current_move);
95         previous_move = current_move;
96     }
97     return scrambled_path;

```

```
98 }
```

`scramble_step()` selects a move randomly using `java.util.Random` then the selected move is checked for whether it is the same move as the previous one or if it is the inverse of the previous one. The move is allowed and returned to `scramble()` if it is neither. If the move is the same or inverse then `scramble_step()` calls it self, meaning the method is recursive, repeating until it finds an allowed move.

```
99 public String scramble_step(String previous_move){
100     Random r = new Random();
101     int which_move = r.nextInt(possible_moves.length);
102     String the_move = possible_moves[which_move];
103     if (!Objects.equals(the_move, previous_move) && !Objects.equals(
104         the_move, reverse_move(previous_move))) {
105         return possible_moves[which_move];
106     }
107     else {
108         return scramble_step(previous_move);
109     }
110 }
```

To see if the cube is solved we have the method seen below, to return an either `true` or `false` value, code shown below. This method simply checks if all the orientation and positions are correct of every cubie and returns `true` if they are all correct and `false` if even one is off.

```
110 public boolean is_cubie_solved(Cubie cubie, int z, int y, int x) {
111     int orientation = 1;
112     boolean position = Arrays.equals(cubie.correct_coordinate, new int
113         []{z, y, x});
114
115     if (cubie.getClass() == Edge_Cubie.class) {
116         orientation = is_orientation_correct_edge(cubie, z,y,x);
117     } else if (cubie.getClass() == Corner_Cubie.class){
118         orientation = is_orientation_correct_corner(z, cubie);
119     }
120     return position && orientation == 0;
121 }
```

7.3 Cubie

The `Cubie` class is a superclass containing two subclasses, `Corner_Cubie` and `edge_Cubie`. The distinction between edge- and corner-cubie is used for dealing with orientation, as edges can have two different orientations and corners can have three. These two classes are there to define whether or not the cubie uses a guide color. The guide color gets assigned in `Cubie` and is only necessary for `Corner_Cubie`. In the subclasses, the different sides of the cubies are defined by assigning them a color. The superclass contains all the methods used to manipulate the cubies and evaluate the cubie. This includes the method shown down below: `assign_guide_color()`. This method simply goes through every side of the cubie and figures out if there is a centerpiece color that is equal to its own color near it and define that as its guide color. If we take the corner cubie {red, yellow, blue}, if this cubie is then next to the yellow center tile, it will the define yellow as its guide color. This works of course with every color. This is to figure out what the orientation of any given corner cubie is. In

the same way, the edge cubies have an orientation which are used in the heuristic in Section 7.5.

```

121 public void assign_guide_colors() {
122     for (int color = 0; color < side_color.length; color++){
123         if (side_color[color] != null && color < 2){
124             guide_color_1 = side_color[color];
125         } else if (side_color[color] != null && color < 4){
126             guide_color_2 = side_color[color];
127         } else if (side_color[color] != null && color < 6){
128             guide_color_3 = side_color[color];
129         }
130     }
131 }

```

7.4 Open_node

This class is used to keep track of all the data of every node or possible state the cube can be in. Here we first have a constructor that just defines the fitness and path of the node. Where the fitness is calculated in the **Solver**, Section 7.5. Secondly, path is the way we identify the different states. Every state of the cube has a unique and arbitrary set of moves you need to apply to it to get to that state. The objects made from this class will be stored in a heap, in java this is called a **PriorityQueue**. The method `compareTo()` compares the fitness of this node with another, which lets the heap order the nodes depending on their fitness, so the node with the lowest fitness is first, since the lower the fitness the better the state of the cube. The **PriorityQueue** is used is the class **Solver** explained in Section 7.5. Closing off the class with two methods, one to output the fitness of a given node, `get_fitness()`, and one to output the path of a given node, `get_path()`.

```

132 public class Open_Node implements Comparable<Open_Node>{
133     double fitness;
134     byte[] path;
135     public Open_Node(double fitness, byte[] path){
136         this.fitness = fitness;
137         this.path = path;
138     }
139     @Override
140     public int compareTo(Open_Node o) {
141         if (this.fitness<o.get_fitnes())
142             return -1;
143         if (this.fitness>o.get_fitnes())
144             return 1;
145         return 0;
146     }
147     public double get_fitnes(){ return fitness;}
148     public byte[] get_path(){ return path;}
149 }

```

7.5 Solver

The Solver class handles the algorithm. The algorithm is defined by the `algorithm()` method. First, the moves used to scramble the cube have to be saved in the same format as the path the nodes store, i.e. from string array to byte array. The algorithm works

by opening 18 nodes for each node it looks at, one for each possible move. To get the algorithm started, an arbitrary node is constructed, with no path and some fitness, this is the initial state as shown on Figure 4. From the initial state, the 1st depth is opened with a call of the `make_open_nodes()` method, description in Section 7.5.1. Then `algorithm()` applies a while loop which runs until it finds a solution within the allowed length, i.e. 20. The first thing that happens in the while loop is the first node in the `PriorityQueue` is assigned to `best_node` and removed from the queue. The path, i.e. the moves which takes the cube from the scrambled state to the state of the node, is saved as a variable to be used further down in the method and to print when the solution is found. Then comes the part where the loop prepares for the next iteration. First the cube is put back to its solved state, then it is brought to the scrambled state, and lastly the state of `best_node`. This is done to make sure that no moves are unintentionally applied to the cube. Then the next batch of child nodes are opened from `best_node`, data is collected, and the loop starts over. When a solution is found the loop ends and the path of the solution is returned.

```

150 public byte[] algorithm(int max_depth){
151     byte[] byte_scramble_path = new byte[cube.scrambled_path.length];
152     for (int i = 0; i < cube.scrambled_path.length; i++) {
153         byte_scramble_path[i] = cube.string_to_byte(cube.scrambled_path
154             [i]);
155     }
156     byte[] move_list = {};
157     Open_Node best_node = new Open_Node(20, new byte[]{});
158     make_new_nodes(best_node);
159
160     while (best_node.get_path().length < max_depth && !cube.is_solved()
161         ){
162         best_node = all_open_nodes.poll();
163         assert best_node != null;
164         move_list = best_node.get_path();
165
166         cube.create_cube();
167         cube.go_to_path(byte_scramble_path);
168         cube.go_to_path(move_list);
169         make_new_nodes(best_node);
170     }
171     return move_list;
172 }

```

In the end the algorithm we implemented was A* since it proved easier to implement with our data structure. The way the cube and nodes work fits better with A*, because states of the cube are stored as moves and reached by manipulating the cube. Our approach does not benefit from IDA* being memory light, instead using A* makes use of all the information our approach necessarily stores. In Section 9.2 we discuss further how we could have done it differently.

7.5.1 Opening nodes

`make_open_nodes()` receives a parent node and opens child nodes. A `foreach`-loop which makes a new node for each possible move. First, the path from the parent is copied to a new array which is one element longer, and the child's move is added to the path. Then the move is applied to the cube so the fitness of that state can be evaluated, the fitness calculation is

described in Section 7.5.2. The fitness of the state and the path to the state is saved in a new `Open_Node`, which is added to the `PriorityQueue` and to an `ArrayList` which contains all nodes opened during the run of the program, the `ArrayList` is used to collect data. Lastly, the cube is iterated back to the parent state by applying the inverse move to the cube, and the loop goes on.

```

172 public void make_new_nodes(Open_Node node) {
173     for (byte move : cube.byte_moves) {
174         byte[] new_path = new byte[node.get_path().length + 1];
175         duplicate_array(node.get_path(), new_path);
176         new_path[new_path.length - 1] = move;
177
178         cube.move(cube.byte_to_string(move));
179         Open_Node new_node = new Open_Node(fitness(new_path), new_path)
180             ;
181         all_nodes.add(new_node);
182         all_open_nodes.offer(new_node);
183         cube.iterate_back(cube.byte_to_string(move));
184     }
185 }

```

7.5.2 Fitness

`fitness()` calculates the fitness of the state using the formula $f = g + h$ detailed in Section 5.3.3. g is calculated as the length of the nodes path, i.e. how many states away from the initial state this node is. h is calculated as how many moves each cubie needs in order to be at their correct place and oriented correctly, `moves_until_all_cubies_are_correct()` is described in Section 7.5.3. Since the heuristic calculates the moves needed to solve individual cubies, the sum is greater than what is actually needed, because one move affects eight cubies. Therefore, we decided to divide the sum by a factor. We looked at scientific papers to get an idea of what the factor should be. In an article about pattern databases written by Richard Korf[4], he explains that due to *the Manhattan distance* he divides his heuristic by four. To evaluate his decision, we conducted some preliminary tests, looking at the amount of nodes opened by the algorithm, see Section 8.1. Through these tests it was concluded that dividing by five gave the best results, see Section 8.1. Lastly the calculated fitness is returned.

```

185 public double fitness(byte[] path) {
186     int g = path.length;
187     double h = moves_until_all_cubies_are_correct() / 5;
188
189     double f = g + h;
190     return f;
191 }

```


7.5.3 Heuristic Estimate

The heuristic consist of two methods `moves_until_all_cubies_are_correct()` which goes through all cubies and sums up the result from calling `moves_until_cubie_is_correct()` on each of them. `moves_until_cubie_is_correct()` returns an interger between zero and three. zero is if the cubie is in its correct place with correct orientation, because then zero moves are necessary to get the cubie where it needs to be. If the cubie is an edge cubie then it can be in three different incorrect states:

- correct orientation but one move away form its correct place (a move which does not change its orientation), which require one move to get to the right state.
- at its correct place but with incorrect orientation, which requires three moves to get to the right state.
- any other state can always get to the right state in exactly two moves (we found out by checking on a real cube).

If the cubie is a corner cubie then it has two different incorrect states:

- correct orientation but one move away form its correct place (a move which does not change its orientation), which require one move to get to the right state.
- any other state can always get to the right state in exactly two moves (we found out by checking on a real cube).

Depending on the state of the cubie the above-mentioned values are returned and summed together to get the value for the heuristic. When the heuristic is zero, every cubie are on their correct place with correct orientation.

```
192 public double moves_until_cubie_is_correct(int z, int y, int x){
193     Cubie this_cubie = cube.rubix_Cube[z][y][x];
194
195     if (!cube.is_cubie_solved(this_cubie, z, y, x)){
196         if (this_cubie.getClass() == Edge_Cubie.class){
197             if (cube.is_orientation_correct_edge(this_cubie, z, y, x)
198                 == 0){
199                 return 1;
200             } else if (Arrays.equals(this_cubie.correct_coordinate, new
201                 int[]{z, y, x}) && !(cube.is_orientation_correct_edge(
202                     this_cubie, z, y, x) == 0)){
203                 return 3;
204             } else {
205                 return 2;
206             }
207         } else if (this_cubie.getClass() == Corner_Cubie.class){
208             if (cube.is_orientation_correct_corner(z, this_cubie) == 0)
209                 {
210                     return 1;
211                 } else {
212                     return 2;
213                 }
214         }
215     }
216     return 0;
217 }
```

7.5.4 Arguments for heuristic

To evaluate the cube the heuristic of independent moves to correct position and orientation was made. To explain how the heuristic was created we need to understand that there are two types of cubies; Edge_cubie and Corner_cubie. These two have four and only four different cases they can be in. Where the corner and edge cubie truth tables are Tables 1 and 2 respectively.

Position	Orientation	Moves
T	T	0
F	T	1
F	F	2
T	F	2

Table 1: Truth table showing amount of moves needed to correct a corner cubie

To understand how this table was made, we need to go through every scenario. Here it is seen that the simplest case is if both position and orientation is correct, where as intuition might suggest, zero moves are required to place and orient the cubie correctly. The next case to look at is if the position is incorrect, but the orientation is correct. In this case, consider the way orientation is calculated. With this in mind it can be seen that no matter which guide color is used, the correct position is always one move away. Next let us consider the situation where both orientation and position is incorrect. Observe that no matter what position and orientation a corner cubie might have, it will always be one move away from being oriented correctly, since this is one of the previous mentioned cases, the amount of moves needed to solve the cubie is two. Finally, the case of correct position and incorrect orientation. In this case case we see that no matter the orientation of a corner cubie, we can always achieve incorrect position and correct orientation within one move. This state has been explained earlier, making the total moves needed to correct this case two.

Position	Orientation	Moves
T	T	0
F	T	1
F	F	2
T	F	3

Table 2: Truth table showing amount of moves needed to correct a edge cubie

Considering the truth table for the edge cubie instead of the corner cubie, it can be seen that all but one of the cases are the same, the arguments for these cases are also the same. The one case to differ is where the position is correct and the orientation is incorrect. In the edge cubies case it is not possible to place it in a state of correct orientation first, therefore we need to break the position off, making the new state incorrect position and incorrect orientation. From this state, like the corner cubie, a state of correct orientation can be reached within one move followed by a second move to correct the orientation. This makes the odd cases needed moves to solve 3.

8 Tests and results

To evaluate our algorithm we have gathered a range of data. This can help us see what part of the algorithm needs tweaking. Below are the hardware specification on the test computer we used to gather the data in this section. To know what is relevant to look at regarding

OS	CPU	RAM	GPU
Windows 10 64bit	AMD ryzen 5 2600x	32Gb	AMD radeon RX 5700 XT

Table 3: Test computers hardware specifications

data collection, we can ask the question: what does the different parts of the algorithm affect on the data gathered? Naturally to answer that question with confidence we need to gather some data and compare it to other versions of the program, which negates the point of the question. Although we can still draw a hypothesis, such as:

1. Adding a heuristic to our program, rather than brute forcing the cube, will cause the algorithm to search through fewer nodes, in turn use less time and memory.
2. A better heuristic will cause the algorithm to search through fewer nodes, and in turn use less time and memory.
3. A heuristic looking at the cubies orientation and position will look through more nodes when one move moves 8 cubies, compared to a heuristic that looks at every cubie independently and calculates the amount of moves for them to be on their correct position and orientation.

With these hypotheses we can now test for memory usage, time usage and how many nodes are used in the different versions of our program. More definitively we have three versions, version 1 (brute force), version 2 (heuristic 1) and version 3 (heuristic 2). To ensure that the scramble used to test each program did not favor any program by chance we will run the test multiple times for each program and depth and gather the average, minimum and maximum of every piece of data we have an interest in.

8.1 Heuristic preliminary tests

The preliminary test were conducted to find the best factor to divide the heuristic by. Values ranging from 3-8 were tested on the same scramble. The only solution was the inverse of the scramble meaning each test had to find the same point. As seen in Table 4 simply dividing by some factor makes the estimate better, but dividing by 5 gives the best results.

Scramble: [U180, F180, R, B, L, Ui, Fi, L180]

Solution: [L180, F, U, Li, Bi, Ri, F180, U180]

factor	nodes	time	solve moves
1	26082846	53190ms or 53 seconds	(could not solve)
3	2095992	4594ms	8
3.5	821880	1739ms	8
3.75	645084	1374ms	8
4	484416	1062ms	8
4.25	457794	992ms	8
4.5	389466	829ms	8
5	294624	692ms	8
5.25	379044	824ms	8
5.5	370872	820ms	8
6	458496	973ms	8
7	536580	1170ms	8
8	2012310	4186ms	8

Table 4: Test computers hardware specifications

8.2 User interface

To gather an overview for the reader, here is a quick visualization of how the program looks when it is executed. Initially the program shows a Rubik’s cube that has been unfolded, with the blue side pointing towards you, yellow pointing up, and the red side pointing to the right. See Figure 14.

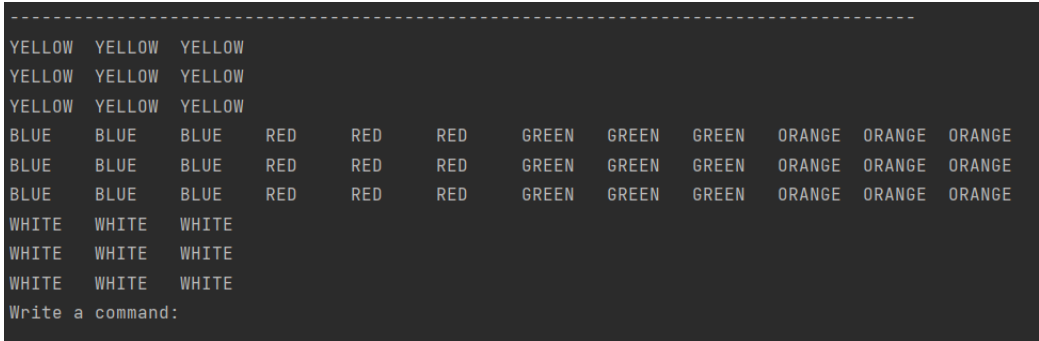


Figure 14: Initial screen when the program is run.

The program will then prompt the user to write a command in the console to execute the next piece of code. As mentioned in Section 7.1, the program can take a range of commands, where to manipulate the cube, you simply write any move of your choice in as the command. Another command has been made to make the scrambling of the cube easier; '*scramble*'. When this command is called the program will prompt the user to input another value, being the amount of random moves they want applied to the cube. see Figure 15.

```
Write a command: scramble
how many scrambles?
7
```

Figure 15: The console while the scramble command is being run.

After scrambling the cube, whether that be manually or with the built in '*scramble*' command, if the user wants to have the cube solved, the command '*solve*' can be written, which will solve the Rubik's cube. Once the solve command has been run, the program will show the line shown in Figure 16. The figure contains, first, a visualization of the cube, secondly, the amount of memory the program has used before running the '*solve*' command, and finally a title for when the solution to the cube has been found.

```
Write a command: solve
-----
WHITE  WHITE  RED
WHITE  YELLOW BLUE
YELLOW YELLOW BLUE
ORANGE ORANGE ORANGE WHITE RED YELLOW BLUE GREEN RED BLUE BLUE GREEN
ORANGE BLUE RED YELLOW RED YELLOW BLUE GREEN BLUE ORANGE ORANGE GREEN
GREEN YELLOW RED YELLOW ORANGE YELLOW BLUE RED GREEN ORANGE RED RED
WHITE GREEN GREEN
WHITE WHITE WHITE
WHITE GREEN ORANGE
before solving
memory Used: 36mb
Solution moves:
```

Figure 16: Screen while the program is solving the Rubik's cube.

Figure 17 depicts what the interface looks like after the cube have been solved. Firstly, the time to solve the cube is displayed in *ms* and if more relevant in '*seconds*'. After *time to compute*, the list of moves needed to solve the scrambled cube we started with. Finally, before the program shows the state of the cube, it will show how much *memory is used* and how many nodes were opened to solve the cube. After showing the cube in its solved state, the program again prompts the user to input a command, where the user can start the process all over again.

```
Solution moves:
time to compute: 153ms or 0 seconds
[Ri, Di, Ri, Li]
after solver
memory Used: 17mb
12528
-----
YELLOW YELLOW YELLOW
YELLOW YELLOW YELLOW
YELLOW YELLOW YELLOW
BLUE BLUE BLUE RED RED RED GREEN GREEN GREEN ORANGE ORANGE ORANGE
BLUE BLUE BLUE RED RED RED GREEN GREEN GREEN ORANGE ORANGE ORANGE
BLUE BLUE BLUE RED RED RED GREEN GREEN GREEN ORANGE ORANGE ORANGE
WHITE WHITE WHITE
WHITE WHITE WHITE
WHITE WHITE WHITE
Write a command: |
```

Figure 17: Initial screen when the program is run.

8.3 CFOP test

To compare CFOP to the computer, two tests has been run where the scramble was known. Here the values for the two tests are saved. Although, since CFOP is done by hand, the time value can vary majorly from solve to solve. Further more, it is not possible to monitor memory on the human executing CFOP, making the value useless.

8.3.1 Test 1

In test 1 the scrambles: [F180, R, F, D, R180, F180, D, L180, R] have been applied. The result from solving this scramble can be seen in Table 5.

Algorithm	Moves	Time
A*2	[Ri, L180, Di, F180, R180, Di, Fi, Ri, F180] (9 moves)	2262ms or 2 seconds
CFOP	96 moves	52.4 seconds

Table 5: Test for CFOP algorithm vs A*2, containing time to solve, and moves applied to solve. Specific moves in CFOP has not been recorded

8.3.2 Test 2

In test 1 the scrambles: [Bi, L, F, F180, R180, U, B180, R180, Di] have been applied. The result from solving this scramble can be seen in Table 6

Algorithm	Moves	Time
A*2	[D, R180, B180, Ui, R180, F, Li, B] (8 moves)	4328ms or 4 seconds
CFOP	105 moves	55.3 seconds

Table 6: Test for CFOP algorithm vs A*2, containing time to solve, and moves applied to solve. Specific moves in CFOP has not been recorded

Through these tests it is obvious to see that at a scramble of 9 moves, A*2 is far faster. Although, with CFOP it is possible to solve any cube, whereas with A*2 whenever the scramble becomes more advanced than 10 scrambles, the algorithm will run out of memory.

8.4 Data

To test the program, a data collection program has been implemented. This program will save and write the relevant data to three separate CSV files that can be graphed later. To make sure that the data most accurately represents the time complexity and memory complexity of the cube. The program has taken a set amount of repeats for every scramble amount. This predetermined number has been set to 50, to ensure that an appropriate average is found. In Figure 7 are the values for the first version of the program. These values have been collected through out 50 runs of random scrambles.

Scrambles	Average nodes	Average memory	Average time	Average moves needed to solve
1	0.0	10.66	0.2	1.0
2	0.0	32.8	0.32	1.92
3	0.0	257.86	2.2	2.82
4	0.0	1684.8	29.24	3.62
5	0.0	12510.6	429.36	4.46
6	0.0	14899.73	6107.71	5.35

Table 7: Average values of each depth after 50 runs for first version of the program. Name: Brute force, IDDFS

Below in Figure 8 the values collected for the second version of the program. This version has been named the '*old heuristic*' since it contains an outdated heuristic. These data points have been collected over 50 runs of random scrambles.

Scrambles	Average nodes	Average memory	Average time	Average moves needed to solve
1	18.0	48.64	0.12	1.0
2	35.64	98.04	0.26	1.94
3	6446.16	5456.06	18.52	2.78
4	180435.6	56565.42	416.48	3.8
5	6558384.38	56380.31	16126.94	5.38

Table 8: Average values of each depth after 50 runs for the second version of the program. Name: old heuristic, A*1

Below in Table 9 the average values of the third version of the program can be seen. These values have been collected throughout 50 runs of random scrambles.

Scrambles	Average nodes	Average memory	Average time	Average moves needed to solve
1	18.0	56.18	0.26	1.0
2	37.8	127.36	0.32	1.94
3	253.08	482.22	1.22	2.74
4	4436.28	4748.56	16.4	3.7
5	20278.44	11961.56	60.08	4.34
6	615992.4	45242.94	1795.74	5.7
7	566368.2	177081.68	1675.84	6.26
8	958138.92	270885.48	2817.5	6.66
9	7573788.64	5180.75	19416.18	7.75

Table 9: Average values of each depth after 50 runs for the third program. Name: New heuristic, A*2

8.5 Time

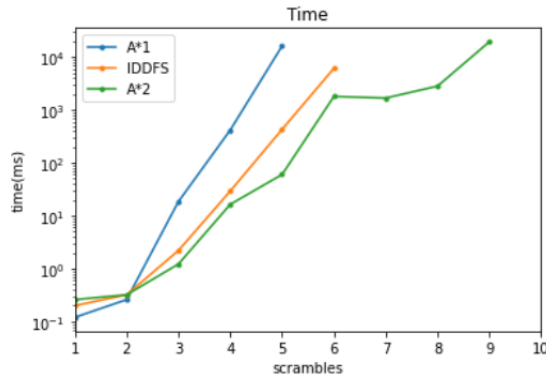


Figure 18: Graph showing the average time used to solve in progression over scrambles

As shown on the graph the three algorithms are around equally fast at solving the cube on 2 scrambles. When more scrambles are applied A*1 is the algorithm using the most time solving the cube. This is because of its heuristic not being as good as the newer ones. It is shown that A*1 does not go beyond scramble 5 do to too many nodes open, explanation in Section 8.7. The same goes for IDDFS that only reaches scramble 6, but IDDFS does not use as much time as A*1. The graph also shows that A*2 is uses a lot less time than the other two. A*2 uses as much time on scramble 9 as A*1 does on scramble 5. A*2 does have a better heuristic and therefore it makes sense to see it getting a lot further using a lot less time.

8.6 Memory

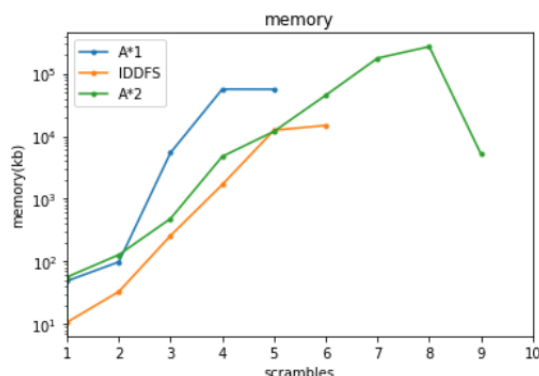


Figure 19: Graph showing average memory usage for solving a cube in progressing

As seen on the memory graph, IDDFS is using the least amount of memory, followed by A*2 and then A*1, which can be seen on the Graph 19. The reason IDDFS has the least memory usage is due to its examination of the move list, and the way it paths though the tree, see Section 6.1. A*2 uses only slightly more memory than the IDDFS, which would indicate that IDDFS is not working as intended. Reasons for this could be the way it searches through the tree. IDDFS's larger than expected memory use could be caused by the big stack of recursions which take up a lot of memory. IDDFS is implemented in a way where the recursion stack never drops a call when it is finished, it remains in the stack as the next state is checked with a new recursive call on top of it. Meaning that when ever the algorithm returns to a previously visited state, it still adds to the return stack instead of saving memory. A*2 memory usage is a lot lower than A*1 which indicates the changes made to A*2 improved the memory use. Some of the improvements that were applied was changing the heuristic see Section 6.1. some of the sources of errors would be A*2 on scramble 9, there is a drastic decrease in memory usage, which should not happen. This is the result of Java's garbage collection, and it will be elaborated in Section 9.4.

8.7 Nodes

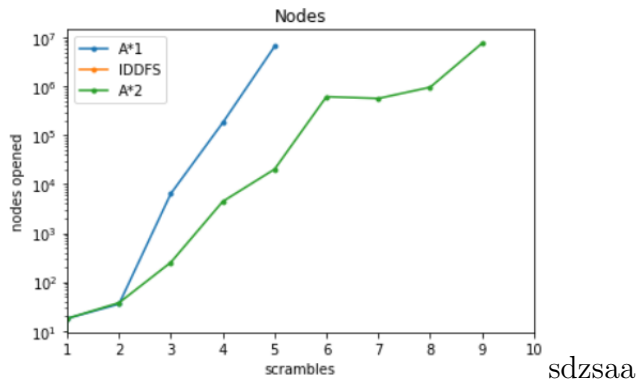


Figure 20: Graph showing average amount of nodes opened for a progressive cube scramble complexity

First of all IDDFS does not appear on the graph, Figure 20, because when developed the node system had not been implemented yet. Hence, the lack of data points. IDDFS is purely recursive and saves no data for each state it checks. The data from A*1 and A*2 it is evident that the number of nodes grows more or less exponentially to find more and more complex solutions. Though, A*1 opens significantly more nodes than A*2. Additionally if you look at Table 8 then you can see its solutions tend to be slightly longer than A*2, however this might be because the same scrambles were not used to gather data, more about this in Section 9.4. Even still, from Table 8 scramble 5 A*1 finds solutions that tend to be more moves than the scramble. There is always a solution that is as many moves as the scramble, the inverse of the scramble, or shorter. A*1 has a poor heuristic which opens a lot of nodes but still finds poor solutions. A*2 has a much better heuristic, which opens less nodes and consistently finds solutions of the same length or shorter than the scramble. Note that the reason for A*1's premature stop is because it opens more nodes at an earlier depth than A*2. This causes the program to crash at an earlier depth than A*2.

9 Discussion

Looking at the program, it is simple to see that it is not perfect nor optimal. Therefore, this section will look into the thoughts we have made for the future, were we to continue trying to solve the problem.

9.1 Heuristic

One of the most important parts of the algorithm is the heuristic within it and has been a great topic of discussion throughout this project. The current heuristic has come down to a crude estimate of the sum of how many moves an individual cubie is away from being positioned and orientated correctly. In an article written by Richard E. Korf[10] he applies a pattern database to his program and uses that as the heuristic. Since this pattern database would contain all patterns a Rubik's cube can have, it can estimate accurately how many moves away a cube is from being solved, becoming a well-suited heuristic. This could expand the functionality of the program, since with the current heuristic the program can not solve cubes scrambled with more than 10 moves applied to it. Although a pattern database would require a change in the programs data structure to support it, since this would be a major change to the program, this has not been applied. But is to be considered for future projects.

9.2 Algorithm

Originally the thought behind this report was to compare Cross – F2L – OLL – PLL (CFOP), the hand way of solving a cube, to IDA*, the computer way of solving a cube. Since IDA* was the main focus, CFOP turned into a lower priority. Additionally, since the algorithm of the program turned out to be more time consuming than expected, this report never went into implementing CFOP. Therefore, if expanded, the report should contain an implementation of CFOP into code and be compared to the current algorithm.

Furthermore, the current algorithm was also originally meant to be IDA* but through discussion and consideration we ended up implementing A* because of the same issue with CFOP, time. If IDA* were to be implemented in such a way that nodes were closed when it returned to the parent, many of the memory issues seen in the current program would fade away. This is due to the way IDA* searches through the tree. A*, as currently implemented, needs to keep track of all options opened, all open nodes, and evaluate the best one, while IDA* would only need to evaluate the newest options opened. This would make the list of all nodes considerably smaller and make it possible for the program to solve more advanced cubes.

To optimize the memory usage in the program, the path of every node has been changed from String array to byte array, read more about this in Section 5.2. The issue with the way that it has been changed is that it is converted throughout the program. This makes the program, first of all, entangled in itself, while also making it more possible for bugs to occur and making it harder to debug and eliminate said bugs. In hindsight, this change should have made us wipe the program and rewrite it with the change in mind, making sure that it is implemented correctly.

9.3 language

As mentioned in Section 5.2 our program is written in java and is object oriented. This decision could have been changed where a number of languages would fit just as well or better. Through reading technical reports and scientific articles, it can be seen that due to the insane amount of possibilities there are for a Rubik's cube, people often write in C++ [6]. Where the issue of memory becomes more manageable and solutions become available for the programmer. The issue with writing in C++ is: as of writing this report none of the co-authors know the language, and therefore we would need to learn the language from scratch while also trying to solve the issue of the cube. Further more we are all inexperienced programmers, making Java a more friendly and possible environment for us to program in, since java has many features, such as garbage collection, that has to be done manually in C++.

Looking past the functionality of our program to the user friendliness. In the future, if this program was to be expanded upon, it would be nice with something a bit more graphic than the text graphics our program uses. Since java is an old language, it does not have many options for Graphics User Interfaces (GUI's), and the ones it has are primitive. Therefore, for future reference, the use of C# (C-sharp) would be a smart possibility. The reason why C# is better than java at this point is, first of all, because it brings in a lot of the functionality and freedom of the C-family such as C++, but also because the game engine 'Unity' uses C#. This would allow us to make all the functionality as done in the current program, while outputting the relevant values to Unity for the visualization of the cube.

9.4 Data

To understand the data seen in Section 8 it is important to mention that the same scrambles were not tested on each algorithm. Currently the way the program runs and collects data, is simply to scramble the cube randomly and solve it. The relevant data is then written to a CSV file and saved. This was then repeated 50 times to gather substantial data to make an accurate average. If the scrambles had been the same for each algorithm at every scramble complexity, fewer data points would be needed to adequately compare the algorithms. A way to counter that the values are random is, as mentioned earlier, by taking an average. This should make the data deviate less from run to run, making the data more trustworthy. Making a small database containing the specific scrambles that are tested on. This is one of the functionalities that should be changed, were this project to be extended.

Further more, the way that memory is calculated and written to our CSV has errors. The way that it is collected at the moment, is to find the memory used before and after running the algorithm and then taking the difference between the two. The way that java collects garbage is to once a reference to a class or variable or method is removed, java will automatically delete it from memory. This is good in theory, but since it would be an issue if the program was interrupted by the garbage collector, it runs on a separate thread on the CPU. This causes the garbage collection to be asynchronous with the program, making the garbage collection inconsistent. Because of the asynchronous garbage collection, it will sometimes collect garbage while the algorithm is running, making the memory after the algorithm has run smaller than before it ran, giving us a negative memory usage. To counter this error the program contains a manual garbage collection call at the bottom of the data collection call. While this collects correctly, it does not stop the program from collecting if the memory starts overflowing, which it does at higher scrambles. Now if the program was written in C++ or another language which has manual garbage collection this could have been avoided, but java has an automatic garbage collector. The permanent solution to this could be, as previously mentioned, to change to a language that has manual garbage collection. Although such programs are far more advanced to manage, which our skill level does not allow.

Finally, the comparison of nodes between the three programs does not work since IDDFS does not run on the node system. To find a way of comparing this, IDDFS could contain a counter, counting the amount of recursions the algorithm has done. Since if the node system were to be implemented one recursive call would equals one node.

10 Conclusion

In conclusion, the Rubik's cube is an advanced color combinatory puzzle which can be in many different states. To solve a scrambled cube there are many algorithms available, where some are more efficient than others. This report has looked into and implemented the algorithms: A* and Iterative Deepening Depth First Search (IDDFS), where other algorithms have been considered. These algorithms are Depth first search, which looks in the depth of a solution, Breadths first search, which looks into the width of a solution, and Iterative Deepening A*(IDA*) which applies heuristics and an iterative deepening depth to find the best solution. Further more, the strength of the heuristic can change drastically how efficient an algorithm can be, where a more crude approach will use more resources. The data structure of the program is also valued with great importance, since the complexity of the Rubik's cube means even small changes in a data types memory consumption gets inflated exponentially with the more complex you make the scramble. We have developed two data structures. Firstly, one which had issues with debugging making us develop a second one, to get around those issues. We had plans for a third which would allow for the usage of a database as a heuristic, but did not have time. Finally, with this in mind, the language written in also matters, since the freedom of what a programmer can do, is heavily influenced by the language they write in. We have implemented and tested two algorithms, one with two different heuristics, and found that the best was A*2 because its heuristic guided the search best. A*2 was faster and used less memory than the other two algorithms, and would often find solutions that required less moves to solve. The plan was to originally make two programs, one running on the CFOP algorithm and one on IDA*, although this turned out to be too time consuming, and was there fore scrapped. Finally, the algorithm in-which this report ended up implementing, A*, seems to be very memory heavy, since it saves all the nodes that the algorithm can chose to move to. If IDA* was used, the algorithm, in theory, should be far lighter on memory usage although it would be more CPU consuming.

References

- [1] Erik D Demaine, Martin L Demaine, Sarah Eisenstat, Anna Lubiw, and Andrew Winslow. Algorithms for solving rubik’s cubes. In *European Symposium on Algorithms*, pages 689–700. Springer, 2011.
- [2] badmephisto. <http://badmephisto.com/>, 2021. [Online; accessed 05-11-2021].
- [3] Christian Alexander Steinparz, Andreas P Hinterreiter, Holger Stitz, and Marc Streit. Visualization of rubik’s cube solution algorithms. In *EuroVA@ EuroVis*, pages 19–23, 2019.
- [4] Richard E. Korf. Finding optimal solutions to rubik’s cube using pattern databases. pages 700–705, 1997.
- [5] Brian Jackson. <https://theconversation.com/how-many-stars-are-there-in-space-165370>, 2021. [Online; accessed 05-11-2021].
- [6] Ben Botto. Implementing an optimal rubik’s cube solver using korf’s algorithm. <https://medium.com/@benjamin.botto/implementing-an-optimal-rubiks-cube-solver-using-korf-s-algorithm-bf750b332cf9>, 2020. "[Online; accessed 05-11-2021]".
- [7] Baruch Awerbuch. A new distributed depth-first-search algorithm. *Information Processing Letters*, 20(3):147–150, 1985.
- [8] Robbi Rahim, Dahlan Abdullah, Saiful Nurarif, Mukhlis Ramadhan, Badrul Anwar, Muhammad Dahria, Surya Darma Nasution, Tengku Mohd Diansyah, and Mufida Khairani. Breadth first search approach for shortest path solution in cartesian area. In *Journal of Physics: Conference Series*, volume 1019, page 012036, 2018.
- [9] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of a. *Journal of the ACM (JACM)*, 32(3):505–536, 1985.
- [10] Richard E. Korf. Analyzing the performance of pattern database heuristics. *Association for the Advancement of Artificial Intelligence*, pages 1164–1169, 2007.
- [11] J perm. https://www.youtube.com/watch?v=MS5jByTX_pk&t=310s, 2020. Online; accessed 16-12-2021.
- [12] Badmephisto. Bad mephisto. <https://badmephisto.com>. [Online; accessed 03-11-2021].
- [13] Nathan S. <https://stackoverflow.com/questions/60130124/heuristic-function-for-rubiks-cube-in-a-algorithm-artificial-intelligence>, 2020. [Online; accessed 25-11-2021].
- [14] Raja. <https://www.tutorialspoint.com/how-to-check-the-memory-used-by-a-program-in-java>, 2019. [Online; accessed 22-11-2021].