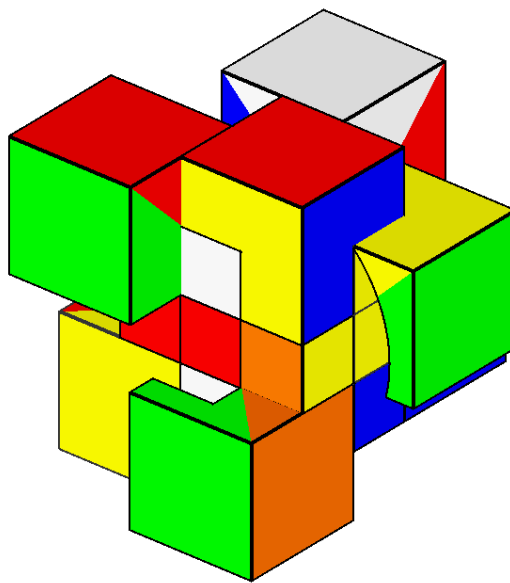


# Graph Algorithms and 3D Rendering of a Complex Shapeshifting Rubik's Cube

**Linus VandeVondele**

Maturitätsarbeit



Mathematisch-Naturwissenschaftliches

Gymnasium Rämibühl

6. January 2025

Supervised by

Pascal Forny

*Linus*  
*2025*

---

## Abstract

In this thesis, the Puppet Cube V2, a shapeshifting variant of the classic Rubik's Cube, is investigated in two parts, namely its 3D rendering and its solution finding with the help of a search. The interactive visualization of this cube incorporates features such as lighting and transparency. The primary focus of this study was the search. The Puppet Cube V2, represented as a graph, is used to investigate five different graph algorithms. The resulting program is able to find short solutions to randomly scrambled cubes quickly and improves the found solution with additional search time. A comprehensive description of the final implementation is provided, which is able to prove an optimal solution, although there exist  $5 \cdot 10^{18}$  positions of the Puppet Cube V2. The algorithm runs in parallel to enhance computational efficiency. Additionally, the thesis presents key properties of the Puppet Cube V2 and the employed algorithm. Notably, a lower bound for God's Number is established, which shows that there exist positions where 30 moves are required to solve the cube. Furthermore, the research highlights improvements in the average depth when searching for longer. Finally, a comparison to a state-of-the-art Rubik's Cube solver further proves the effectiveness of the proposed approach.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	The Puppet Cube V2 . . . . .	1
1.2	The Research Question . . . . .	1
<b>2</b>	<b>Visualization</b>	<b>2</b>
2.1	History and Software . . . . .	2
2.2	Pieces of the Puppet Cube V2 . . . . .	2
2.3	Program Flow Chart . . . . .	4
2.4	Lighting . . . . .	5
2.5	Transparency . . . . .	6
2.5.1	Order-Independent Transparency . . . . .	7
2.5.2	Sorting the Triangles . . . . .	7
<b>3</b>	<b>Search</b>	<b>9</b>
3.1	From Cubes to Graphs . . . . .	9
3.2	Legal Moves . . . . .	9
3.3	Random Scramble . . . . .	11
3.4	Search Algorithms . . . . .	12
3.4.1	Iterative Deepening Depth-First Search . . . . .	12
3.4.2	Visited Map . . . . .	12
3.4.3	Meet in the Middle . . . . .	12
3.4.4	Solve Corners First . . . . .	13
3.4.5	Greedy Search using a Heuristic Function . . . . .	14
3.4.6	Parallelization . . . . .	17
3.5	Flow Charts of the Program . . . . .	19
3.5.1	Main Function . . . . .	19
3.5.2	Search Manager . . . . .	19
3.5.3	Solve . . . . .	20
3.5.4	Search . . . . .	21
3.6	Analysis and Results . . . . .	23
3.6.1	Continuous Solution Improvement . . . . .	23
3.6.2	A Lower Bound for God's Number . . . . .	26
3.6.3	Comparison to standard Rubik's Cube solvers . . . . .	27

<b>4 Conclusion and Outlook</b>	<b>30</b>
<b>References</b>	<b>31</b>
<b>List of Figures</b>	<b>33</b>
<b>List of Tables</b>	<b>34</b>

# 1 Introduction

The Rubik's Cube was invented in 1974 by Ernő Rubik. It was released internationally in 1980 and in the same year received the German Game of the Year special award for Best Puzzle. Around 500 million cubes have been sold, and it is therefore the best-selling puzzle game in the world [1]. The title page of this thesis was signed by Ernő Rubik himself after the symposium of the 50th anniversary of the Rubik's Cube at ETHZ, November 15, 2024. In 1981, Thistlethwaite described a 4-phase algorithm showing every cube scramble can be solved in 52 moves. In 2013, it has been proven that every position of the Rubik's Cube can be solved in 20 moves (half-turn metric) or fewer [2]. This upper bound is also called God's Number. When only allowing quarter moves, 26 moves are needed [3].

There exist many variants of the original cube, some of them have more pieces while others have a different shape. As there are already many programs that solve the original Rubik's Cube, a different variant has been studied in this matura thesis.

## 1.1 The Puppet Cube V2

The Puppet Cube V2 is a twisty puzzle released in 2020 by MoYu [4]. It is an alteration of the standard  $3 \times 3$  Rubik's Cube, featuring a corner block design that limits the number of turns and increases the cube's difficulty. Due to the shapeshifting design and the 5,583,008,927,475,302,400 total number of possible positions the visualization as well as the search for a solution become a challenge.

This cube was chosen because solving this variant by hand is really challenging. The author's first try to solve took more than eight hours, and it is still one of the hardest cubes. The simplicity of the design that results in an intricate puzzle is astounding.

## 1.2 The Research Question

In this thesis, the following research question is answered:

Which shortest path algorithm is capable of solving a complex, shapeshifting Rubik's cube with reasonable time and memory requirements, and how can the solution of the cube be represented graphically in real time?

The whole program, which has been developed as part of this thesis, is publicly available on GitHub [5]. It consists of two major parts namely visualization, discussed in Section 2, and the search, discussed in Section 3. The implementation consists of more than 2000 lines of code, all written in C++.

## 2 Visualization

The goal of the visualization is to be able to interact with the program and additionally learn about 3D graphics. It enables the developer, the user, and the reader to get a better picture of the cube and have an idea of its shape.

### 2.1 History and Software

To represent a 3D object on a 2D computer screen is a long-standing problem. The beginning of 3D graphics dates back to 1960 [6–8]. In 1963 the first rendering software called Sketchpad was invented by Ivan Sutherland. [9] Later on, Ivan Sutherland became lecturer at the University of Utah where he taught *analog computer graphics*. At that time one of the most famous 3D models got created, the *Utah teapot* [7, 10]. The concept of a depth buffer was developed in 1974 [11]. In the 1980s, software developers wrote custom interfaces and drivers for each piece of software. The development of OpenGL, a cross-language, cross-platform API for rendering 2D and 3D vector graphics, began in 1991 [12]. The main competitor of OpenGL became Direct3D, released in 1997 by Microsoft [12].

One of the key facts about computer graphics is that it uses triangles to represent objects. With three points, one will always draw the same triangle. This is not the case with four dots, which can be put together in different ways as can be seen in Figure 1.

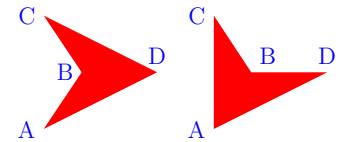


Figure 1: Different spanned areas with the same four corners.

The visualization of the Puppet Cube V2 was written in C++ with OpenGL, to be also compatible with Linux and macOS. As an additional abstraction layer, the combination of GLFW and glad was used. GLFW is a multi-platform library for OpenGL and provides a simple API for creating windows, contexts and surfaces, receiving input and events [13]. Glad is a well-known library that works great with GLFW to find the different versions of OpenGL drivers at run-time [14, 15].

### 2.2 Pieces of the Puppet Cube V2

The Puppet Cube V2 consists of 26 pieces. Just like the Rubik’s Cube, there are 6 centre pieces, 12 edge pieces and 8 corner pieces. As can be seen in Figure 2a, the corner pieces look special. The two opposite lying corner pieces, with the colours *yellow-orange-blue*

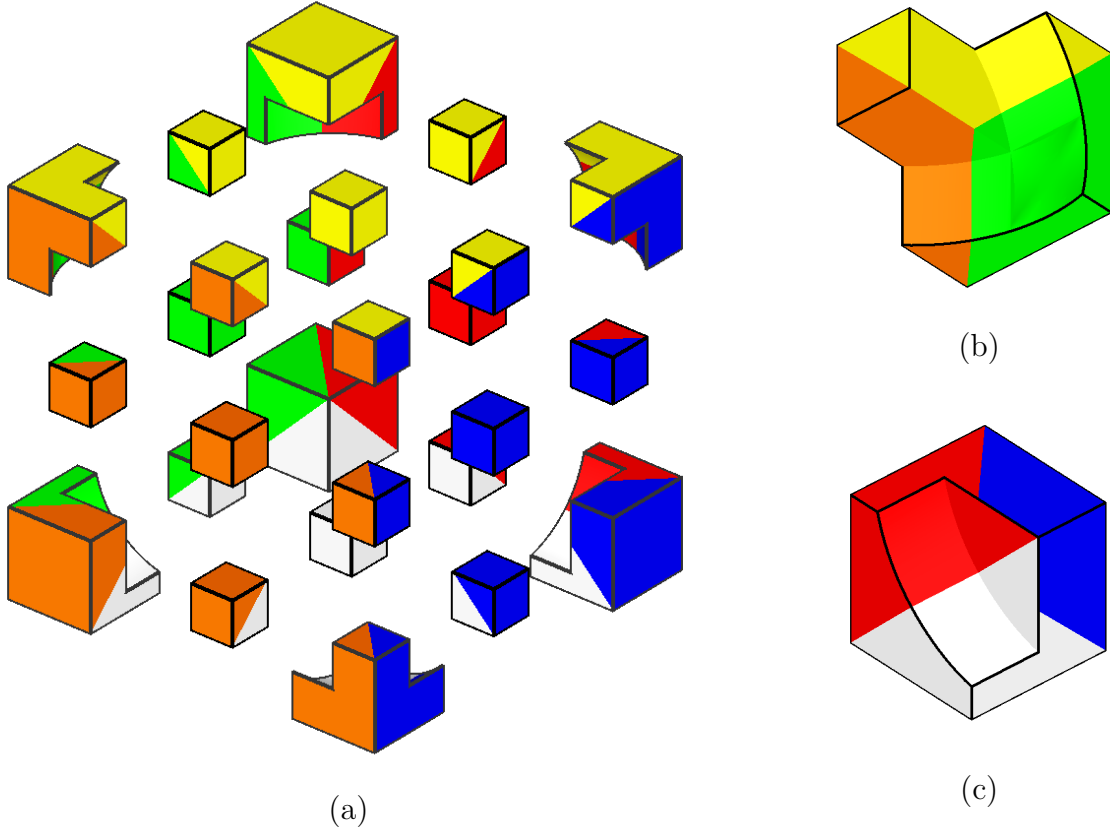


Figure 2: Visualization of the Puppet Cube V2. Panel (a) displays the cube with all pieces spaced apart. In panels (b) and (c) the special corners are visualized. (b) shows a corner piece where one direction is protruding, while (c) shows a corner piece where two directions are protruding. The black piece outline marks the edges of the surface touching the other pieces.

and *white-red-green*, are from the shape identical to the normal cube. The latter corner piece is scaled by a factor of two. The other six corners can be split in two groups (see Figure 2b and 2c). The first group has one direction that is protruding. This means it is in one direction bigger than the standard corner piece. The other group has two directions that are protruding.

To represent the geometry of these pieces is a bit more complicated, especially since 3D rendering only uses triangles. The perception of a curvature can be created by appropriately choosing triangles and lighting the scene. The most challenging part to represent is the intersection between the curvatures. This part has as many triangles as the rest of the piece combined to create a smooth surface.

## 2.3 Program Flow Chart

The window manager is the centrepiece of visualization. The flow chart of this function can be seen in Figure 3. It starts with initializing GLFW and glad, the creation of the window object and redirecting the callbacks. The callbacks can be used for debugging purposes and input handling. Since OpenGL is a low-level API, it lets the user configure a lot, but also gives him more responsibility. One important part are the shaders. There is a vertex shader and a fragment shader, which are part of the OpenGL rendering pipeline and have to be written by the programmer [16]. The vertex shader handles the processing of individual vertices, while the fragment shader outputs the colour and depth of each fragment (pixel of the triangle). The shaders are written in GLSL, the OpenGL Shading Language [17].

The initialization of the mesh creates the pieces using some basic maths and several predefined tables.

After the initialization is complete, the rendering of the images can start and as it is running on another thread, this part of the program does not need to care about the search. The visualization starts by clearing the background. After this it calculates the rotation of the different pieces according to the scrambled position of the cube and the current rotation. The rendering of the mesh is done by rotating it, scaling it and finally sending it to the OpenGL pipeline. This newly generated image, which is currently in the frame buffer, has to be displayed onto the screen.

After the rendering is done, the program gets the inputs from the user to be able to change the view direction and zoom factor. Pressing space can stop and resume the current rotation. Upon closure of the window GLFW will be terminated.

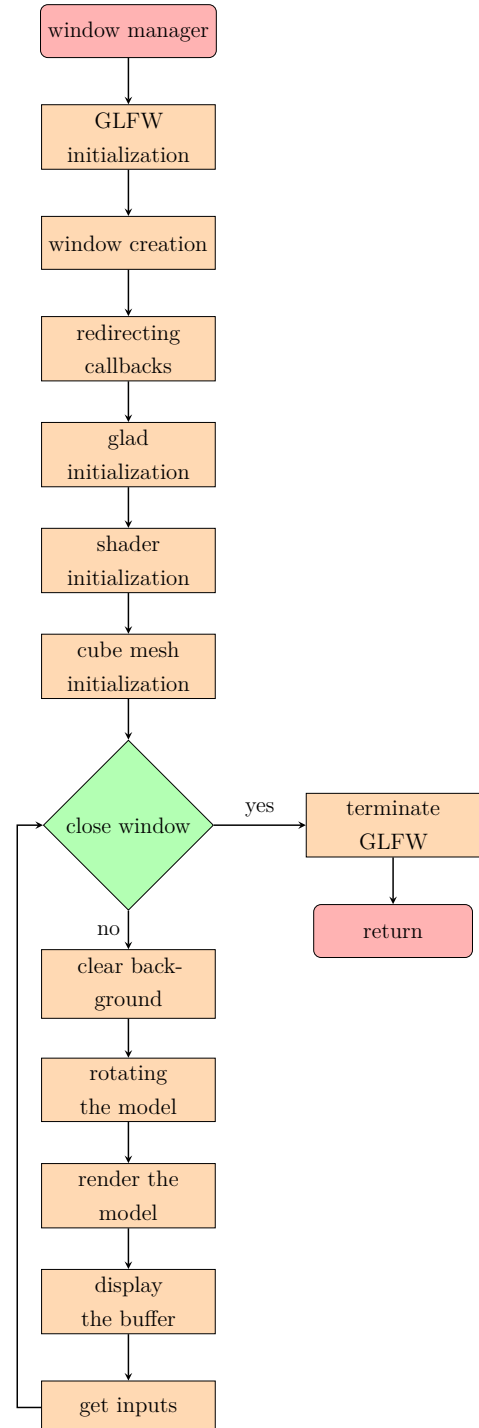


Figure 3: Flow chart of the window manager



## 2.4 Lighting

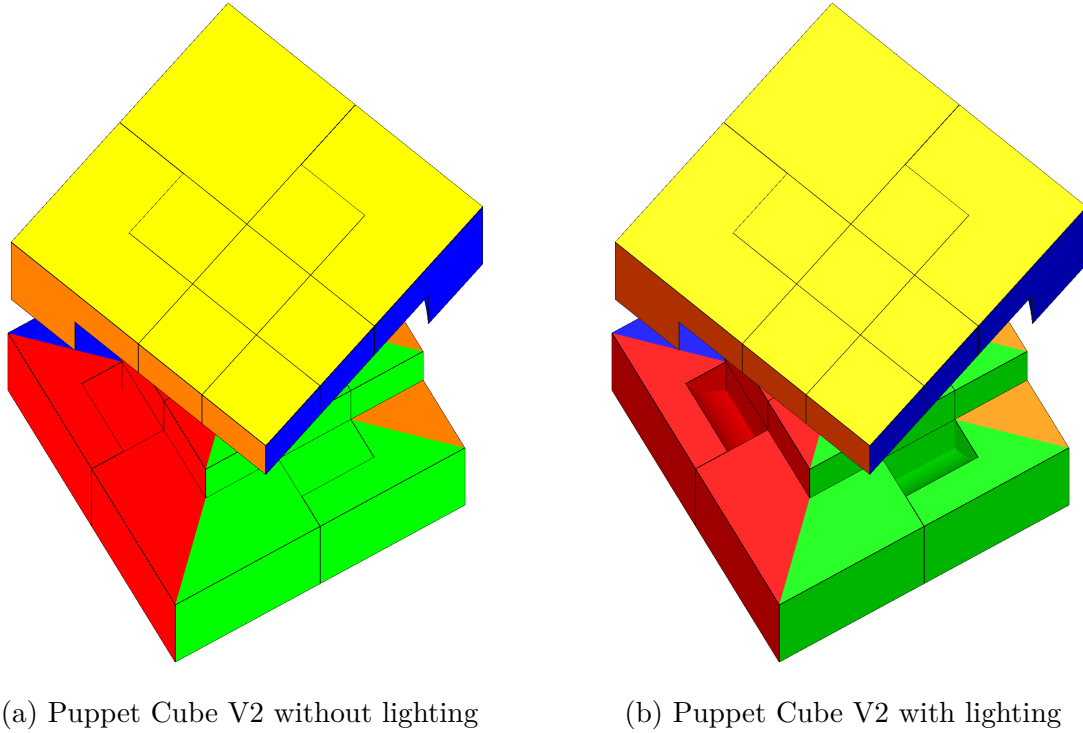


Figure 4: Effect that the lighting of the cube has upon the 3D perception of the object.

One significant part of the visual representation is the lighting. To let the object appear 3D, shading it makes a huge difference. There is a clear difference in the two renderings of Figure 4. The indentation of the lower layer can only be perceived when the lighting is enabled. This lighting effect uses the face normals, which is a vector pointing perpendicular to the spanned area. As the light is coming from the viewer perspective, areas with normals that are perpendicular to the viewing direction appear darker.

Although lighting the cube helps the user to get a 3D perception of the cube, it also comes with new challenges. The curved parts are as previously mentioned just some triangles. As the triangles have different orientations the colour of them changes abruptly as can be seen in Figure 6a. To circumvent this, one can either increase the triangle count and make it less noticeable or use some interpolation between colours within one triangle. The

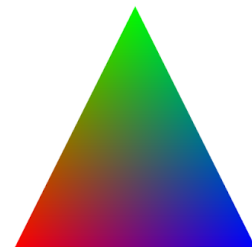


Figure 5: Default behaviour of a OpenGL triangle with different corner colours. (Source: The Coding Notebook)

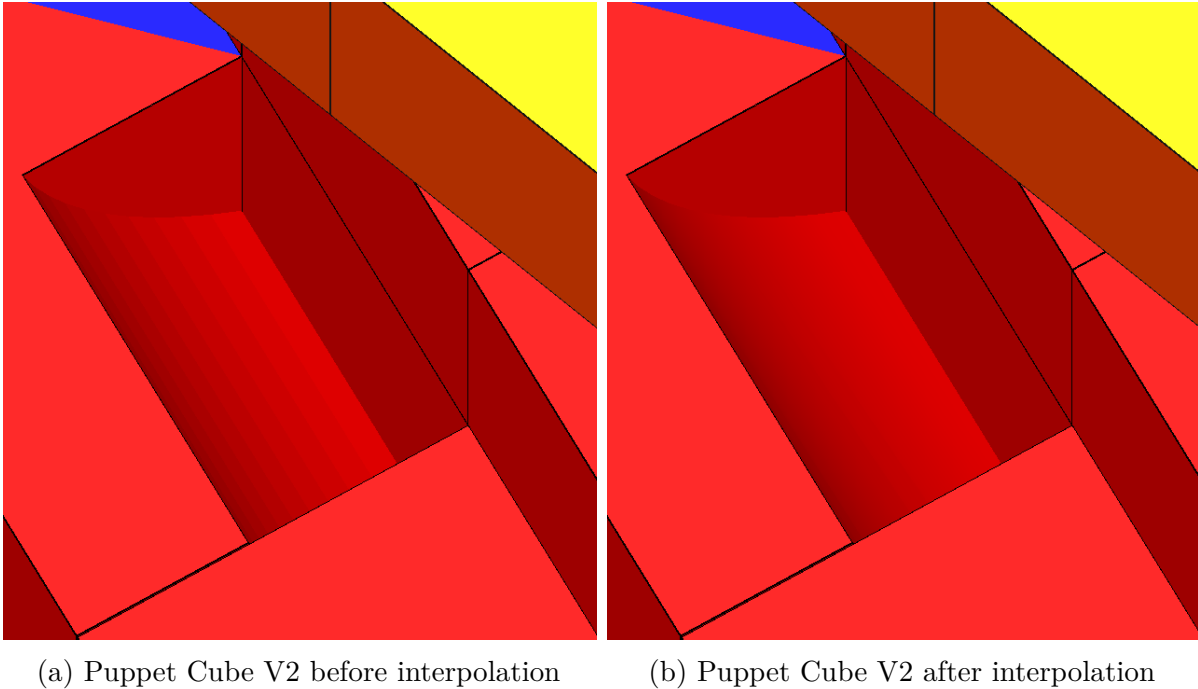


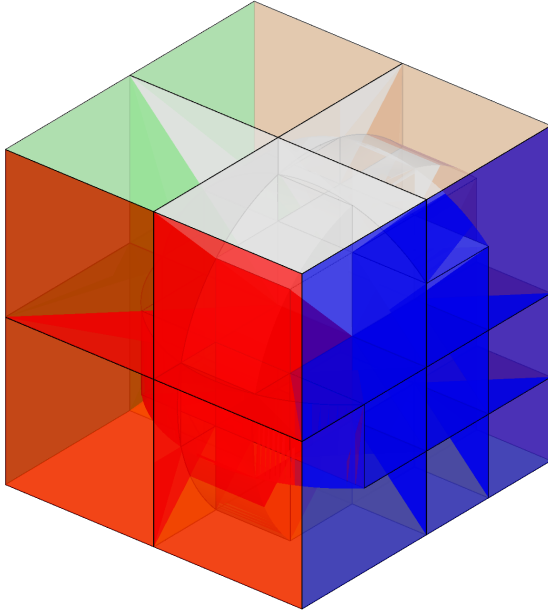
Figure 6: Interpolation of the face normals to get a smoother transition between triangles that form a curved plane.

default behaviour of OpenGL when giving the corner of the triangles different colours is to interpolate them (see Figure 5). The corner normals for triangles comprising a curved plane need to be averaged out according to the area of the triangles. The calculation of the normals can be done during the cube mesh initialization and uses a minimal angle between two triangles to count them as the same curvature.

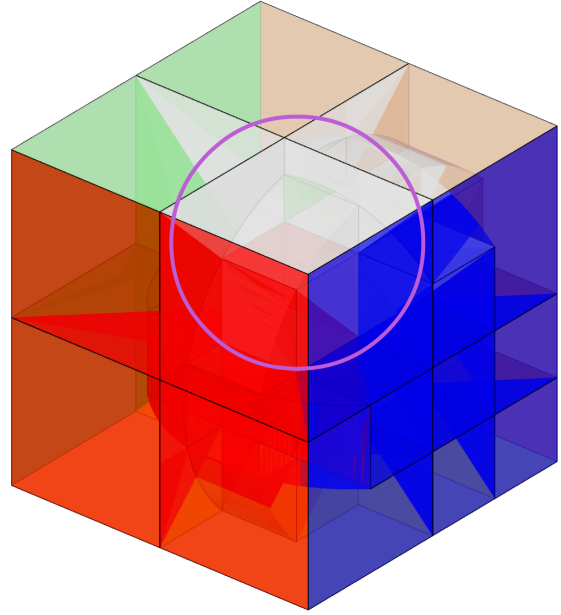
## 2.5 Transparency

Some positions of a Puppet Cube V2 may look the same from the outside from all angles but are different inside. An example for this sort of positions is the solved cube and the solved cube, where two of the hidden edge pieces are flipped (See Figure 7). To be able to see all pieces one can either displace the pieces as is shown in Figure 2a or enable transparency. Enabling transparency is simple, but displaying the scene correctly is difficult, as expressed on the OpenGL wiki [18]:

... you will find it very hard to avoid errors with acceptable real-time algorithms. It's largely a matter of what you are prepared to tolerate and what you know a priori about your scene content.



(a) Solved transparent cube



(b) The edges red-white and green-white are both flipped and can be seen due to the cube's transparency.

Figure 7: Cube with flipped or permutated inner edges looking solved from the outside.

There are two main approaches to solve this issue: Sorting the triangles or redrawing the scene multiple times. Both of them are discussed in the following section. A simplified version of sorting the triangles has been adopted. As can be seen in Figure 7, a transparent cube is implemented.

### 2.5.1 Order-Independent Transparency

With multiple redraws of the same scene, the depth-test approach can work. It does not only need one Z-Buffer but two. There is no dual depth buffer extension to OpenGL [19, 20]. This approach is also not trivial and the number of passes is dependent of the numbers of layers (but this should not be too many). This approach has not been adopted.

### 2.5.2 Sorting the Triangles

Just sorting the triangles is not always possible. If three triangles A, B and C have to be rendered and A lies on B, B lies on C and C also lies on A, sorting is not possible. In this case the triangles have to be split. For the visualization of the Puppet Cube the splitting up will not occur. Sorting them in  $O(n \log n)$  does not work because just looking at two random triangles is not enough to decide which one is on top of the other. Although the red line and the blue line lie at the exact same spot in Figure 8, the order of the lines

from the camera perspective is different. The simplified version of the transparency, which has been implemented, takes the depth of the centre point of the triangle and sorts them. This is not exact, and errors can be seen, but is a good compromise.

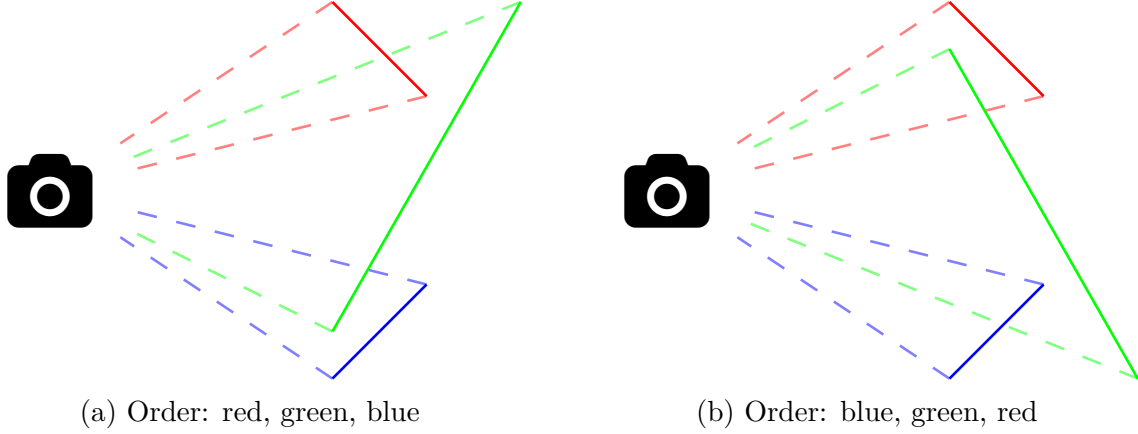


Figure 8: Shown is a 2D representation of the sorting problem. The order of red and blue is reversed in the two pictures, despite the fact that the lines are in identical positions.

## 3 Search

The search is the main part of the project and uses graph theory, algorithms and their implementation to solve the cube. The search aims to find either just a solution or find the shortest solution. Different algorithms are discussed and some facts about the cube are analysed.

### 3.1 From Cubes to Graphs

To study search, the cube must be represented as a graph. The following terminology is used when describing graphs: Nodes are different positions of the cube that are connected by edges, these edges are the legal rotations of the cube. Solving the cube means moving from one node in the graph, which is a scrambled position, to a node, which is the solved cube. The optimal solution is the shortest path between these two nodes. This graph theory problem is so hard because it has  $5 \cdot 10^{18}$  nodes and  $3 \cdot 10^{19}$  edges. This is too many nodes to visit, let alone to store.

### 3.2 Legal Moves

Table 1: Number of legal positions on the Puppet Cube V2 and the total number of edges.

11,382,336	$1 \cdot 10^7$	total number of corner positions
<b>5,583,008,927,475,302,400</b>	$5 \cdot 10^{18}$	total number of positions
<b>34,024,400,694,647,193,600</b>	$3 \cdot 10^{19}$	total number of edges in the graph

The first step for solving this graph theory problem is obtaining the legal moves of the cube. Having such a complex geometry makes it difficult to check if the pieces are intersecting during the rotation. Luckily it is equivalent to only check the intersection at the end of a turn. As the intersection only depends on the corner configurations, it is sufficient to only consider these. There are at most  $8! \cdot 3^7 = 88,179,840$  corner permutations (as this is the number of permutations of a normal  $2 \times 2$ ). So it is possible to precompute every corner configuration. Not all of them are legal in the Puppet Cube V2, because some positions have overlapping pieces. It has been determined that the Puppet Cube V2 has around 11 million legal corner positions and thus has approximately 8 times less possible corner permutations (see Table 1). During the precomputation the total number of edges of the graph was determined to be: 34,024,400,694,647,193,600. On average a random position has approximately 12.2 legal moves. As the edges are

undirected, there are only 6.1 times as many edges as there are positions.

To have extra evidence that the found values in Table 1 are correct the following verification check has been looked at. Using symmetry reasons, the number of corner positions needs to be divisible by various integer numbers. The following factors have been identified as:

- 3: orientation of the biggest and smallest corner
- 6: permutation of the corners with one protruding direction
- 6: permutation of the corners with two protruding directions
- 2: permutation of the smallest and biggest corner
- 4: 90° rotation around x-axis
- 2: 180° rotation around y-axis

$$3 * 6 * 6 * 2 * 4 * 2 = 1728$$

$$11382336 \equiv 0 \pmod{1728} \quad (1)$$

Indeed, inserting the found number of corners from Table 1 into Equation 1 shows that they are divisible. This is a strong indication that the number of legal positions is correctly determined by the program.

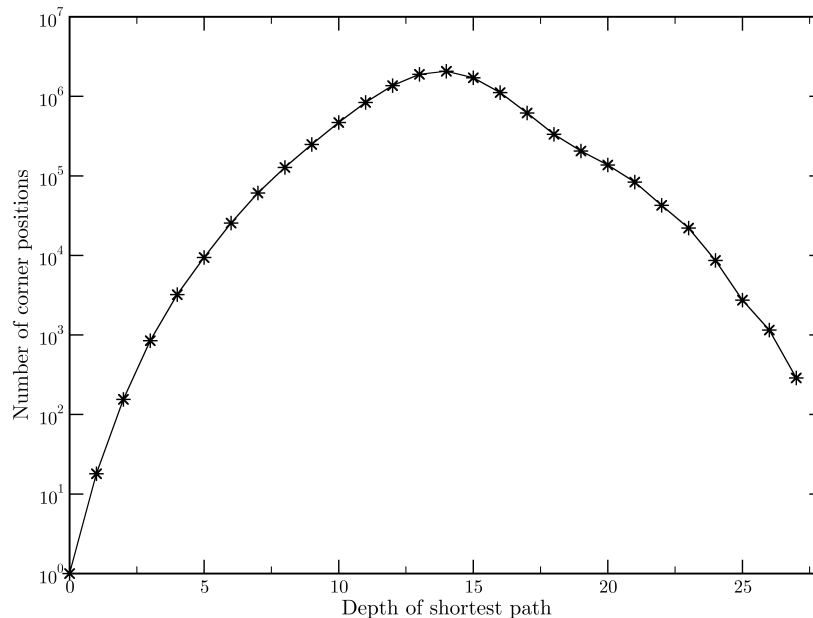


Figure 9: Shown is the distribution of the number of positions with a given shortest path length to solve, for a cube containing just the corner pieces. Most commonly the cube can be solved in 14 moves, but for 288 positions 27 moves are needed.

As can be seen in Figure 9, the maximal number of moves to solve just the corners of

the Puppet Cube V2 is 27. This is higher than the number of moves to solve the entire standard Rubik's Cube when using the quarter turn metric, which is 26 [3]. From the same precomputation the lower bound of God's Number, as the longest optimal solution is called, can be even increased to 28. The proof is as follows: Using dynamic programming, the exact number of ways to reach each of these end positions can be computed. From Figure 9 can be observed that only 288 different corner configurations require 27 moves to be solved. All of these 288 corner positions have less than 830 million ways to reach their respective corner configuration within 27 moves. This means that only at most 830 million different edge configuration for every corner configuration can be reached within 27 moves. However, for every of these corner configuration there are  $12! \cdot 2^{10} = 490,497,638,400$  legal edge configurations. From this, one can conclude that 27 moves is not enough to solve the full Puppet Cube V2 and at least 28 moves are necessary.

### 3.3 Random Scramble

To generate a random position for the Puppet Cube V2, 1000 random legal moves are computed, and the cube is scrambled. As the graph of the Puppet Cube V2 has cycles with odd length and God's Number is not much more than 30 moves long, as will be discussed later, all positions are reachable with 1000 moves. Still this does not lead to a truly random position. There is some bias to positions that have more legal moves. As can be seen in Figure 10, the likelihood of the scrambled position is proportional to the edge count. This can be shown using Markov chains and the Perron-Frobenius theorem [21]. The Rubik's Cube has always the same amount of legal moves and the scramble will generate a random position, as long as the graph created by the legal rotations contains an odd cycle. This is not the case if only quarter turns are allowed. Either slice moves or half turns need to be possible. The Puppet Cube V2 has positions that have 8 – 18 legal moves. This scrambling method results in some scrambled positions being twice as likely as others. In this thesis, this is not a big problem as there is not a huge difference in the likelihood of the different positions, but the fact that random positions are not totally uniformly distributed should still be kept in mind.

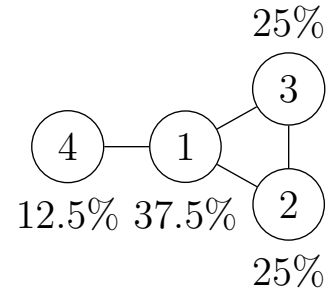


Figure 10: After scrambling an infinite amount of time some nodes are more likely to be the scrambled position than others.

## 3.4 Search Algorithms

### 3.4.1 Iterative Deepening Depth-First Search

As random access memory (RAM) is one of the most important resources for this problem, given there are so many possible positions, a graph traversal algorithm that runs in  $\mathcal{O}(\text{depth})$  memory is definitely a consideration. An iterative deepening depth first search (short IDDFS) is an algorithm that first searches all paths that are  $N$  moves long before searching the paths that are  $N+1$  moves long. This ensures that the first solution found is always the shortest solution. It works similar to a breadth-first search, but needs nearly no RAM. The big downside of this algorithm is that it is really slow:

$$\mathcal{O}(b^d) \tag{2}$$

The time grows exponentially with depth  $d$ . The branching factor  $b$  is approximately 12, the average number of legal moves. As there are positions that need at least 28 moves to solve the cube (i.e.  $d \geq 28$ ), the execution time of this algorithmic approach would exceed the age of the earth by at least a factor thousand.

### 3.4.2 Visited Map

As the IDDFS visits the same position many times, the next step is to reuse information of some positions. Every position that has been visited for a given depth gets stored in a visited map. Using IDDFS with a visited map is nearly equivalent to a breadth-first search. It will be discussed later that the memory of this algorithm grows for quite a long time exponentially with base 10 (See Figure 12). The initial idea of combining IDDFS and using visited maps instead of a standard breadth-first search was to be able to only store some positions to that map instead of being forced to store all positions. Storing all of them is not possible because of memory reasons.

### 3.4.3 Meet in the Middle

Using a meet in the middle algorithm helps with both time and memory usage. Instead of searching from one side until the search finds the solved state, the program solves the cube from both sides at once. The shortest solution is found the first time that the searches meet (see Figure 11). As the two sides grow both exponentially the time complexity can be analysed to be:

$$\mathcal{O}(b^{\frac{d}{2}}) \tag{3}$$



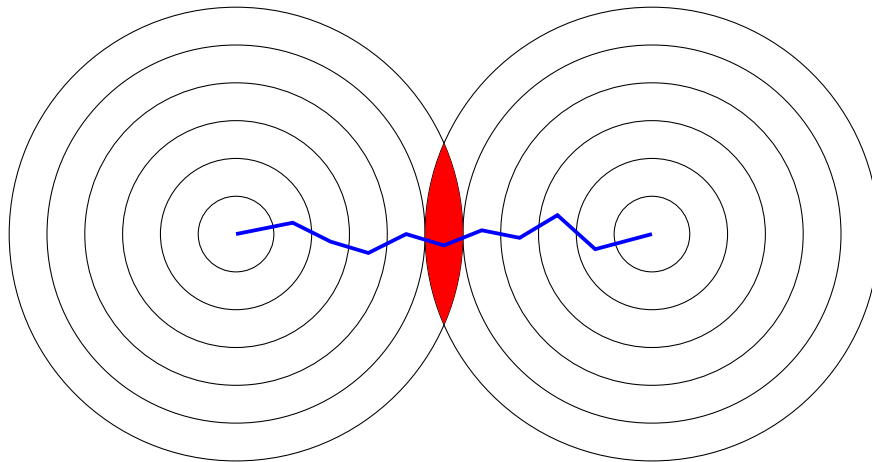


Figure 11: Visualization of the idea of meet in the middle. The red area represents the first intersection of the two searches. The blue line shows a possible path that is optimal.

where  $b$  is the branching factor and  $d$  is the depth of the optimal solution. This is a very large improvement over the  $\mathcal{O}(b^d)$  of IDDFS. The check if both searches have the same position has approximately a constant cost, as unordered maps were used instead of ordered maps. The latter would have an additional logarithmic cost factor.

To get the branching factor  $b$  for the Puppet Cube V2 the search from the starting position has been studied. This side is also called the tablebase as it is a big lookup table that has been generated on the start of the program and is independent of the position, which has to be solved. The tablebase shows exactly how many unique positions can be reached in a specific number of moves and not with fewer moves than that. From Figure 12 the branching factor  $b$  can be read to be approximately 10. This approximation holds definitely until depth nine, but it is clear that this will at some point decline, reaching ultimately zero as the number of nodes is finite.

Still, the meet in the middle program is only able to solve a fraction of the positions as many positions need more than depth 18 to be solved. Depth 18 is the limit to this approach as the tablebase can, with the available RAM, only be calculated until depth 9 and the other side appropriately the same depth. As depth 18 is significantly below the lower bound for God's Number of 28, better approaches have been developed in the following.

#### 3.4.4 Solve Corners First

As searching for optimal solutions is time and resource intensive, the following approach tries to get a solution that is short, but not necessarily optimal. The algorithm starts to solve the corners of the cube first, totally independent of its edges, with the hope that the

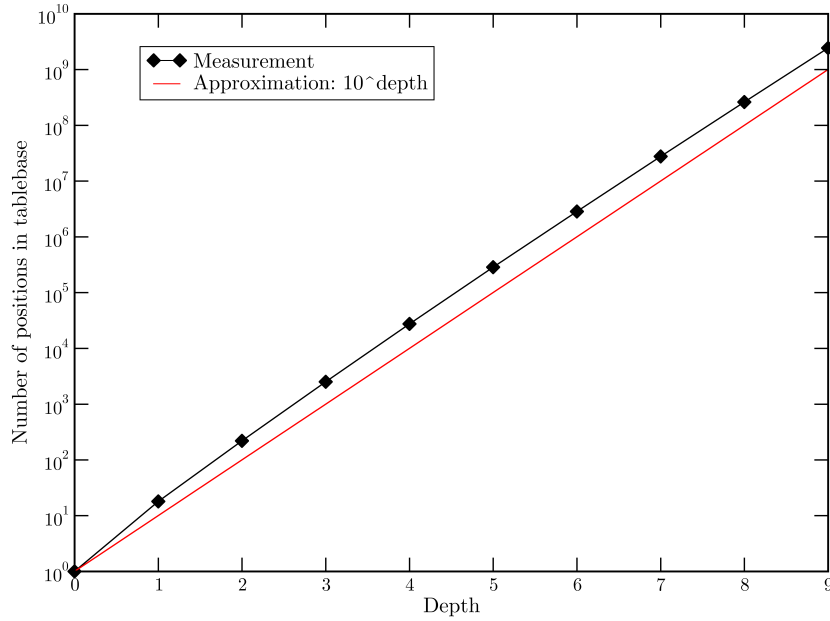


Figure 12: Shown are the number of positions in the tablebase as a function of the depth ( $N$ ). This number equals the number of unique positions reachable from the solved cube in exactly  $N$  moves. This number, at least initially, grows exponentially with  $N$ , approximately increasing with a factor 10 for each depth increment.

cube with all corners correct is easier to solve. Solving the corners was chosen because during the precomputation phase all corner configurations have been analysed. With this table precomputed, first solving the corners is simple. Still the search afterwards is not easy. When starting from random positions, but after solving the corners, very few solutions are found with an additional search depth 14. In fact only 6.6% have been solved with an average combined depth of 27 moves. In comparison, without the corner solving none of 10,000 tested positions were solved with depth 14. However, searching with this algorithm up to depth 16 already solves 71.9% of positions with an estimated average combined depth of 29.

### 3.4.5 Greedy Search using a Heuristic Function

This next algorithm attempts to solve the cube based on a heuristic function. It always searches the node that is heuristically closest to the solved state first. The distance is determined using the heuristic function and in this case the smallest value is the nearest. If this heuristic function ( $h$ ) were defined as

$$h(\text{depth}) = \text{depth} \quad (4)$$

it would be equivalent to a breadth-first search and would therefore find an optimal solution. Such a heuristic function is not helpful, of course, and more useful variants are described in the following.

During the precomputation of the legal moves, all corner configurations of the Puppet Cube have been computed and for each of them the distance to the solved position has been stored in addition to the legal moves. Similar to the corners, the edges also get precomputed but as there are  $12! \cdot 2^{11}$ , which is approximately  $9 \cdot 10^{11}$ , possible configurations the edge pieces must be split up into two groups, each consisting of six pieces. These new smaller groups have each only 42 million positions, and can thus be precomputed as well. The heuristic function is

$$h(\text{corner}, \text{edge}_1, \text{edge}_2) = \text{corner} + \text{edge}_1 + \text{edge}_2 \quad (5)$$

where *corner* is the corner heuristic, the number of moves needed to only solve the corner pieces of the position, and *edge*<sub>1</sub> and *edge*<sub>2</sub> are the corresponding edge heuristics. It is the number of moves that would be needed if the three parts could be solved independently. With this heuristic, the average time to find a solution is only 0.15 seconds. This is thus a great improvement over the exact algorithm. Unfortunately the average length of the solution is 255 moves long.

When this heuristic function encounters an equality case, a random position with this lowest value is chosen. To improve this case, another deciding factor, the depth, can be added to the heuristic function. As it should only influence in equality cases the rest of the heuristic is scaled by a large number.

$$h(\text{corner}, \text{edge}_1, \text{edge}_2, \text{depth}) = (\text{corner} + \text{edge}_1 + \text{edge}_2) \cdot 100 + \text{depth} \quad (6)$$

This function results in an average depth of 152, which is quite a big improvement. The equation can be generalized to:

$$h(\text{corner}, \text{edge}_1, \text{edge}_2, \text{depth}) = (\text{corner} + \text{edge}_1 + \text{edge}_2) \cdot \text{factor} + \text{depth} \quad (7)$$

When changing the constant factor, Figure 13 was produced. Taking a smaller factor increases the time usage dramatically but also increases the quality of the result. This is a nice way to regulate the time usage. Using a factor 0 would result in Equation 4 and be a breadth-first search. Because the corner and edge precomputation give a lower bound on the number of moves it takes to solve the cube, taking the maximum of the

three precomputations will also result in a lower bound. As

$$\max(a, b, c) \geq \frac{a + b + c}{3} \quad (8)$$

picking the factor  $\frac{1}{3}$  or lower for Equation 7 will result in an optimal solution. Running the program with this takes too long, but taking the maximum is a technique that can be used independently of the heuristic function to cut of some branches in the search. It can be shown that these branches will always exceed the best solution found so far.

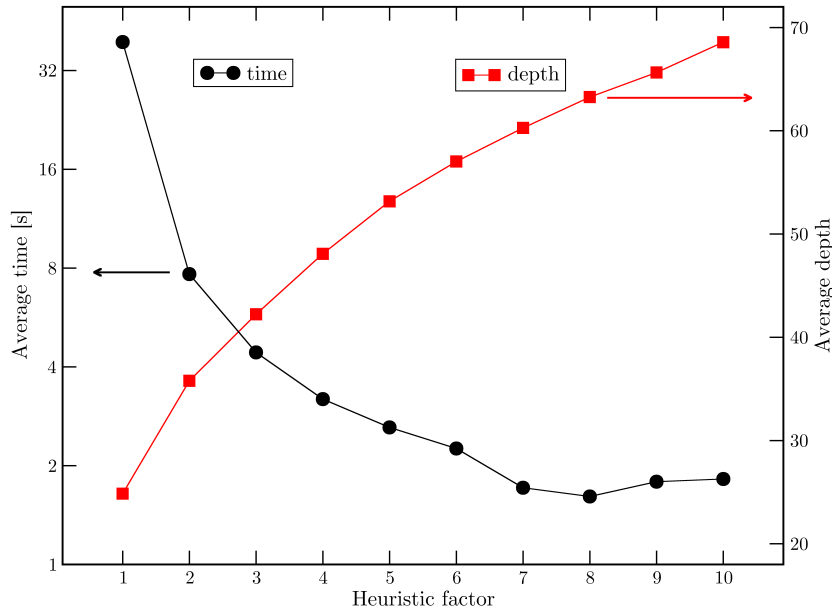


Figure 13: Effects of the heuristic factor in Equation 7 on the average depth of the first solution found and the required time.

To pick a reasonable value for the factor in Equation 7, the value of it was varied as shown in Figure 13. The figure shows that the cost increases steeply with decreasing value, and the final version of the program uses 1 as the factor, which allows for some additional optimizations. With other independent improvements, the average time to find the first solution becomes just 4 seconds, 10 times faster than the implementation shown in the figure. The great advantage of this approach is that when running the program longer it will improve the solution. This fact will be elaborated in Section 3.6. As the solution is improved further and further, the program will at some point be able to prove optimality of the solution for the position. This algorithm can also be combined with the tablebase approach.

### 3.4.6 Parallelization

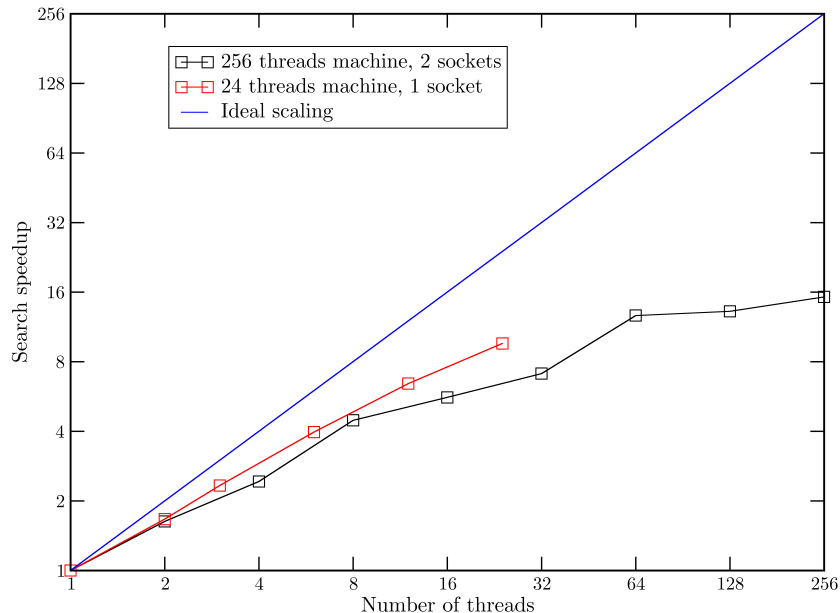


Figure 14: Shown is the search speedup when using multiple threads. By using more of the computer’s potential, the program becomes more than ten times faster. When comparing the results to the ideal scaling, it gets apparent that there are definitely some performance losses due to multithreading.

A way to speed up programs is by using multiple threads. This is not always simple, and a naive implementation may not even be beneficial. An easy way to parallelize the search of the Puppet Cube is by running different scrambles simultaneously. The big downside of this approach is that each thread will use a lot of memory. Having twenty threads would need ten to twenty times as much memory as when using one thread. Because memory is a precious resource, another method is more helpful. This leads to multithreading the search, which is not trivial at all. As can be seen in Figure 14, a significant speedup has been achieved. Using multiple threads, the program runs 15 times faster!

To obtain parallelism, C++ threading has been employed, using atomic variables and locks. For two key data structures, open source implementations have been used. Saving the visited map was implemented using *The Parallel Hashmap* by *greg7mdp* [22]. It implements a fast parallel version of the unordered map. The difficult part is replacing the needed priority queue as these are difficult to parallelize. To circumvent this problem, the priority queue has been replaced by a vector of queues, one queue for each value, and as

these are small integers, they are not many. This allows for the use of the *concurrentqueue* by *cameron314*, which is a fast lock-free queue implemented for C++ [23].

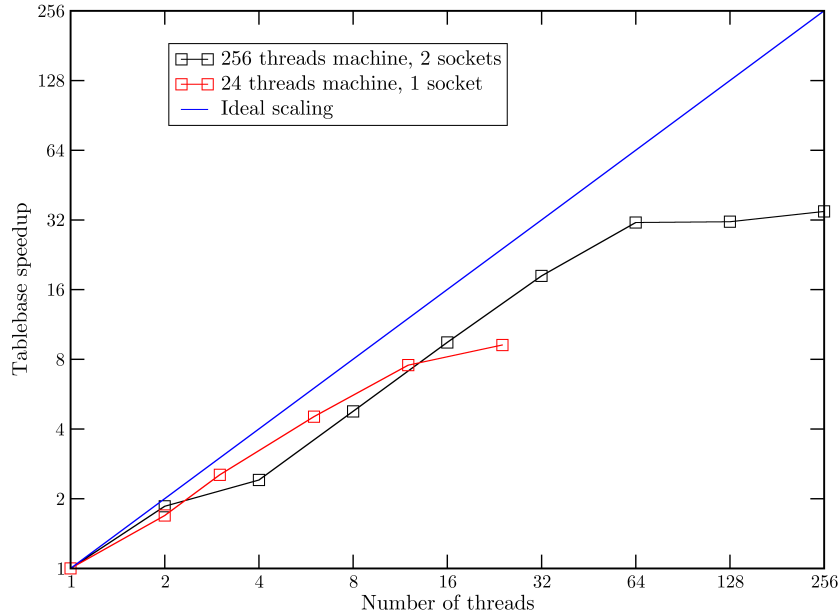


Figure 15: Shown is the tablebase precomputation speedup when using multiple threads. The improvements work really well until hitting the per socket core count.

The parallelization of the tablebase achieved an even higher speedup. Tablebase initialization of depth 9, which took 45 minutes without multithreading, now only needs 5 minutes on a PC with 12 cores / 24 threads. On stronger hardware, the speedup levels off at the core count of one socket. As the speedup is quite linear until hitting this limit, it is well possible that a single processor with more cores would be even faster.

Finally, on powerful hardware, the program is now capable of searching more than a million positions per second. This allows for searches reaching up to  $3 \cdot 10^{10}$  positions in five and a half hours, which was necessary for the studies presented in Section 3.6.

## 3.5 Flow Charts of the Program

### 3.5.1 Main Function

The flow chart of Figure 16 describes the preparation of the search. The main function of the program starts by initializing the error handler. This is a convenient way to debug and have additional information when an error occurs. The settings are command line arguments like the number of threads, tablebase depth, number of positions to search, number of runs, the scrambling depth, etc. The whole list can be found in the GitHub repository [5]. This enables simple testing of different aspects of the cube and simplifies running the code with some changed variables. To communicate between search and the visualization a shared class called actions was used. The window manager, which is responsible for the 3D rendering, is started on a different thread if the user wants to have a visualization. This can be disabled using an argument or during the compilation, so OpenGL is not a requirement to run the search. Afterwards the corner and edge data, which are once precomputed, are read from a file. Now the program is ready for the search manager. When the search manager finishes and the window manager is also closed the threads are joined together and the program finishes.

### 3.5.2 Search Manager

The search manager starts with the precomputation of the tablebase. Then it initializes the cube class, the representation of the cube in the search. The program loops over all the runs and scrambles the positions. The moves to scramble the cube are sent to the window manager via the actions. Then it is time to solve the cube. This is the central part of the program and discussed in the next two sections in depth. Finally, to have more information some statistics are printed to the screen. The time usage, the number of positions, and the depth are all shown with statistical information: total, maximum, 3rd

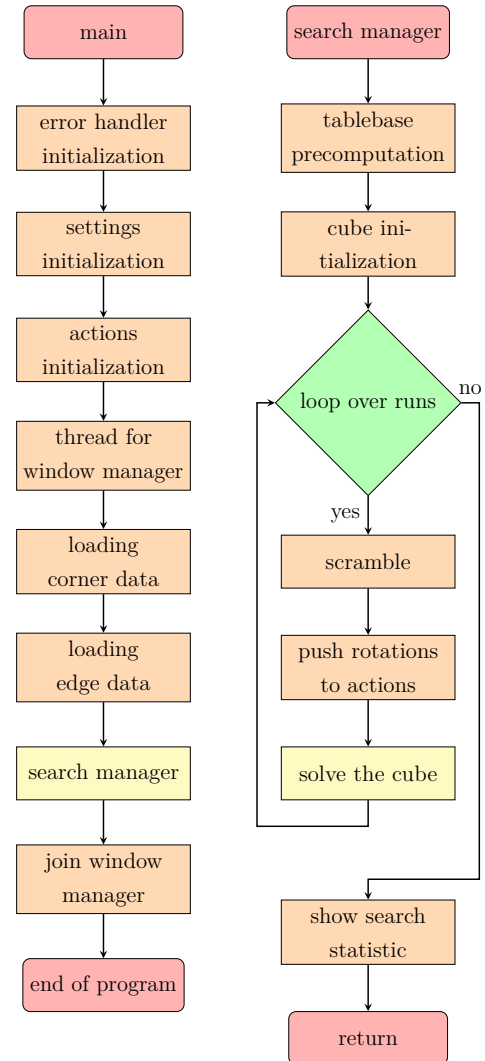


Figure 16: Flow chart of the main program and the search manager

quartile, median, 1st quartile and the minimum.

### 3.5.3 Solve

The program first checks if the position, which needs to be solved, is already in the tablebase. If this is the case, solving the tablebase position is not so difficult. The program tries to always decrease the tablebase value. As it is known that the distance function is optimal and there always exists an edge to a position that is one closer to the solution than the current position, it is possible to always take the path that has a decreasing distance. If the position is not in the tablebase the search queue, a vector of queues, is initialized. Each queue corresponds to a possible value of the heuristic function. The visited map is also created.

As the search uses multiple threads, some shared atomic variables have to be initialized. The different threads all start with the search approximately at the same time. The search is discussed in the following section. When they have finished they are all joined together.

If the found solution is optimal, which is the case when the size of the search queue is zero, it is optionally printed to the console. This is valuable information, which will be used later. If no solution is found it will also print that and returns false.

Once the search has found a solution, the tablebase has to solve the other side of the solution. The cube, which is in the tablebase as well as in the visited map and is part of the shortest solution, is used when solving the tablebase position. The shortest path on the search side also gets pushed to the actions. This completes the solve function.

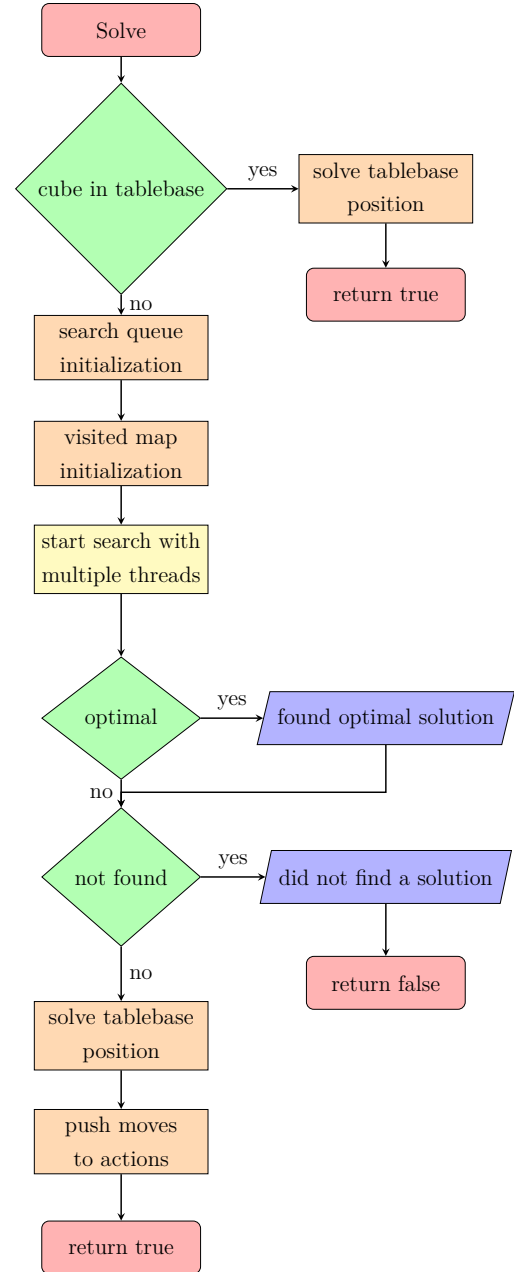


Figure 17: Flow chart of the solve function

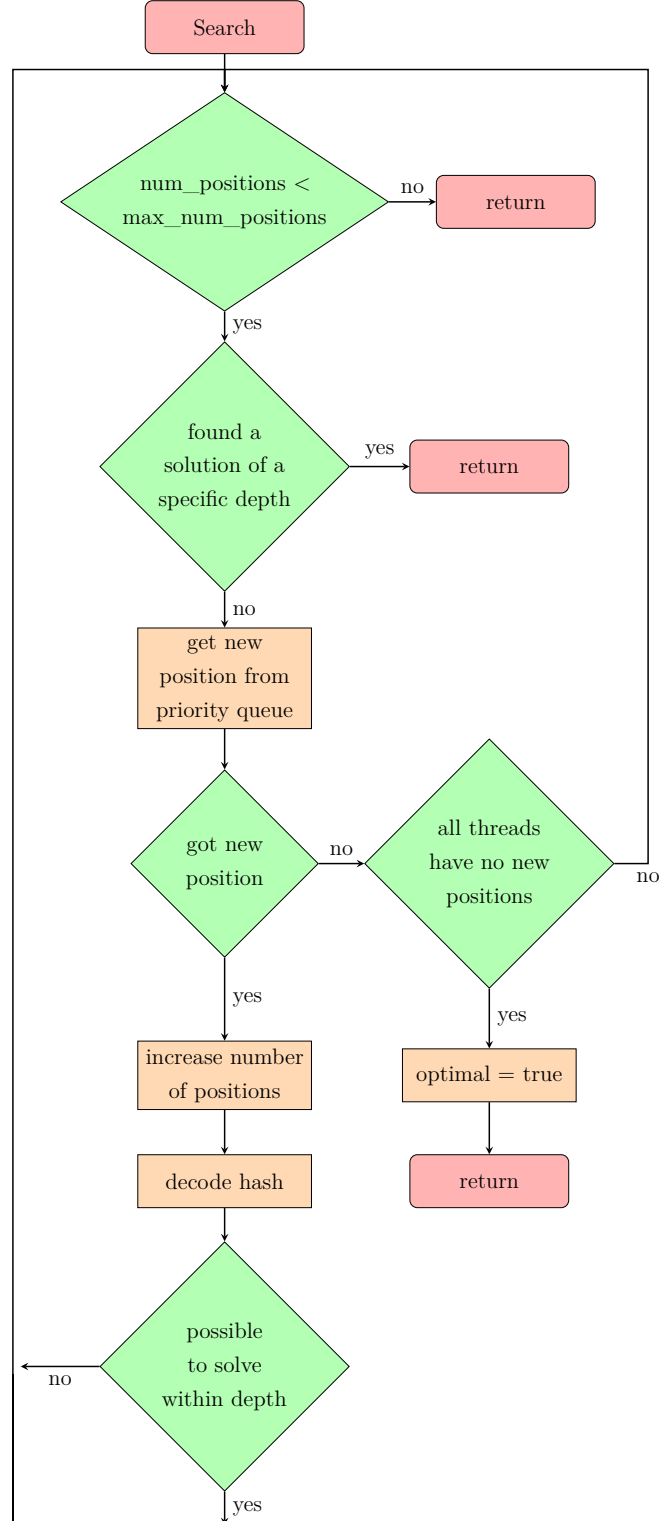


### 3.5.4 Search

This is the central part of the program where the critical algorithms are implemented. As the search is similar to a breadth-first search, it is also implemented iteratively. This does not mean recursively calling a function, but looping over the priority queue. To have control over how long the program takes and how much the memory it is approximately going to use, the implementation searches until a specified number of positions has been visited. The number of visited positions is an atomic variable shared between the threads. Additionally, there is a command line option to stop when the program found a solution that has a specific depth. This is useful to find lower-bounds for God's Number higher than a previously found one.

The thread tries to obtain a new position from the priority queue, implemented using a vector of concurrent queues. If the thread did not get any new position it checks whether the other threads also have no new positions. If this is the case, they have searched through the whole subtree and have therefore found an optimal solution. Otherwise, the program just tries again, as it is possible that one thread is still searching on a position and adds new positions to the priority queue.

If it found a new position it increases the number of positions counter and decodes the position hash, the smallest representation to identify every unique position. As



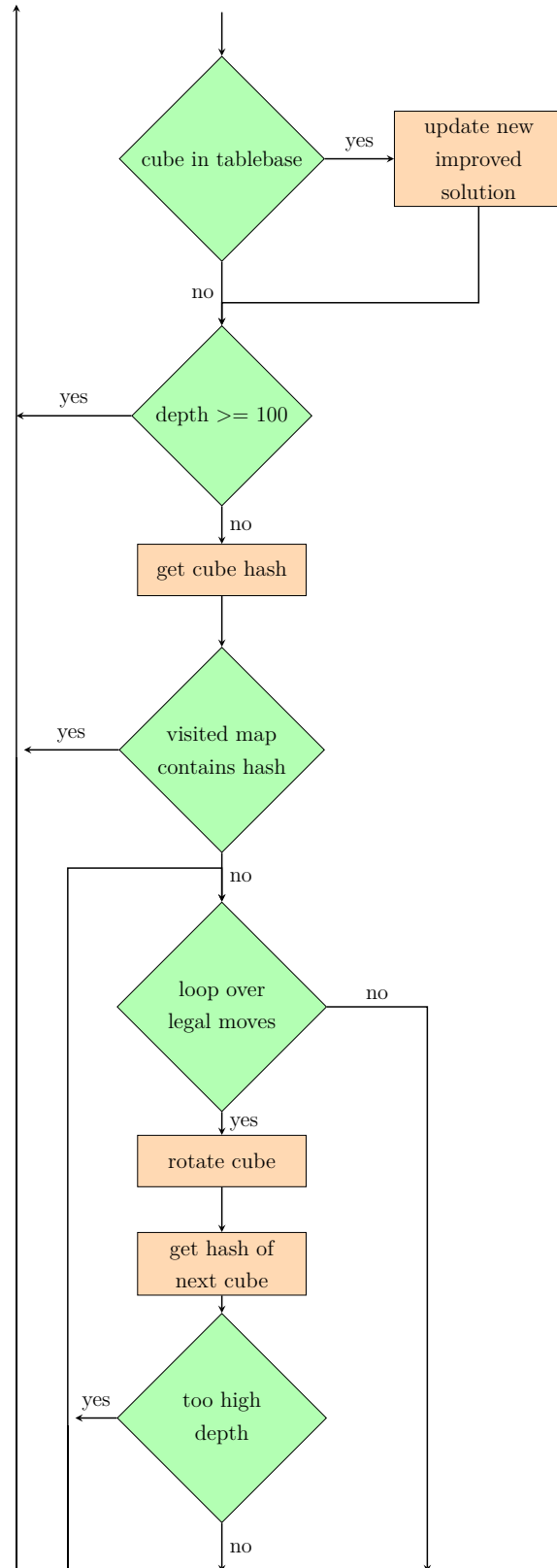
mentioned in Section 3.4.5 using the maximum of the heuristic function an additional bound can be created. This is used to stop searching branches where it is not possible to improve upon the current solution.

If the current cube is in the tablebase, the meet in the middle algorithm found a new improved solution. This has to be updated with all other threads.

As the solution depth never reaches a hundred moves and the rest of the implementation is dependent on a maximal depth, every position that reaches such a high depth will be safely ignored. Additionally, the program has to check if the visited map contains the current position or if this position has been visited before with a lower depth.

Now that all cases where the program skips the position have been looked at, the program loops over the legal moves. The current cube first gets rotated according to the move and thereby the new cube is created. Afterwards the new hash of the cube gets calculated.

The two following checks, if the next position of the cube has a too high depth and if the position is already visited, are both done to reduce the memory usage. Although the program now checks them twice, it is beneficial to do so. If the new position has not yet been visited or visited with a higher depth, the new position gets inserted, or the old position gets replaced by the new one. Finally, this position gets inserted in



the search queue.

After looping through all legal moves, there is another optimization. The final program not only uses Equation 7 with the factor equal to one, but has an additional summand. This is a variable that describes how often this position was looked at. Instead of always adding all next nodes to the search queue, it is possible to do so in batches. The heuristic function only changes slightly. It improves by a maximum of 2 and a minimum of  $-4$ . This allows the program to first solve all positions that are improving or equal to the current position, afterwards position that are 1 heuristic worse, 2 heuristic, 3 heuristic and 4 heuristic worse. With this addition the search queue and the visited map have fewer positions at once and the memory usage decreases by roughly a factor 4.

### 3.6 Analysis and Results

In the following sections the most important properties of the cube as well as the program are being discussed. To conduct research about this cube, 45 runs of differently scrambled cubes were computed until the solution of each cube was proven to be optimal. The Figures 19-22 show different aspects that can be deduced from this data set. Additionally, some specific searches for a lower bound of God's number were conducted.

#### 3.6.1 Continuous Solution Improvement

As previously explained, the search improves the found solution when continuing searching for more positions until an optimal solution is found. In Figure 19 the best found solution is shown after a specific number of searched positions for 6 selected runs. It is observed that the solution improves commonly more than four times. Improving a found solution can take some time. An example is run 4. Between the second last depth and last

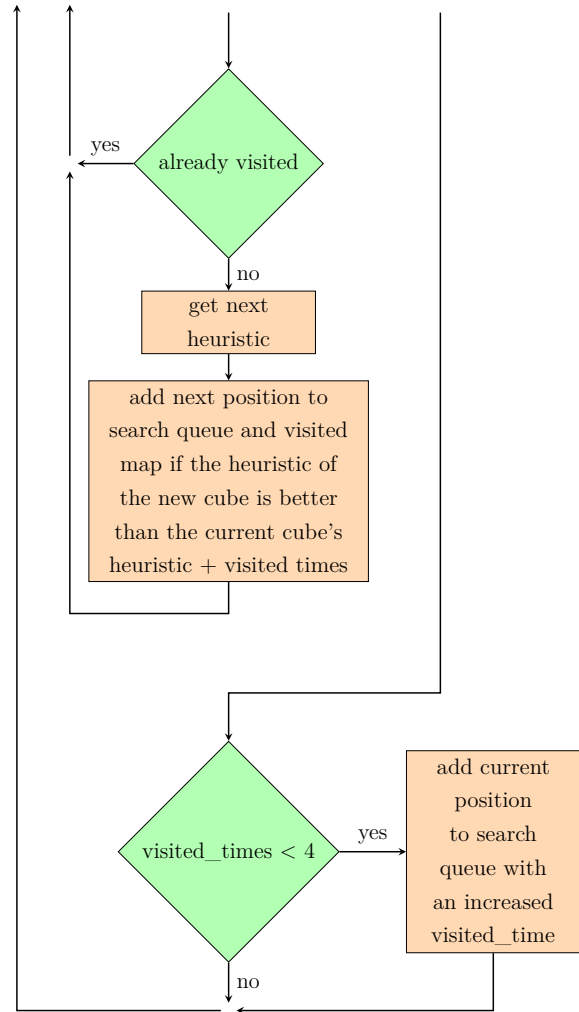


Figure 18: Flow chart of the search function.

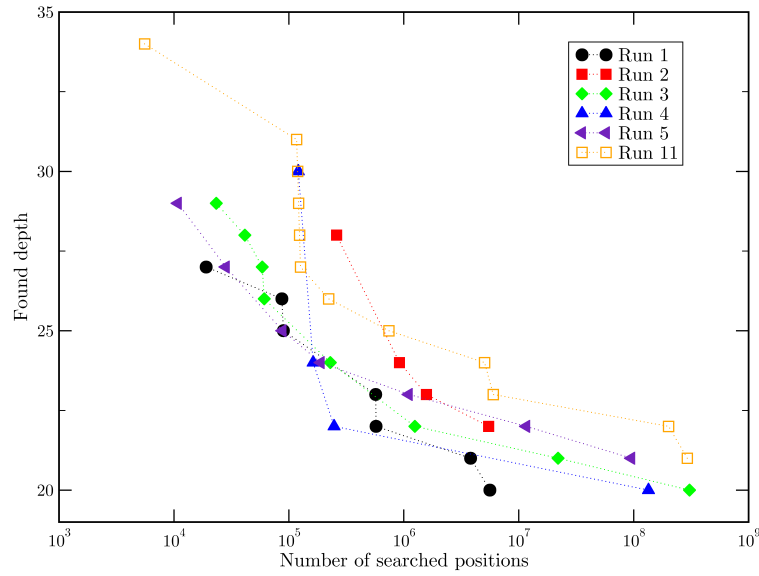


Figure 19: Shown is the length of the best found solution after searching a specific number of positions for different starting scrambles (runs). When searching for longer the algorithm finds better and better solutions, until optimality. Run 11 was hand-picked because it has so many improvements.

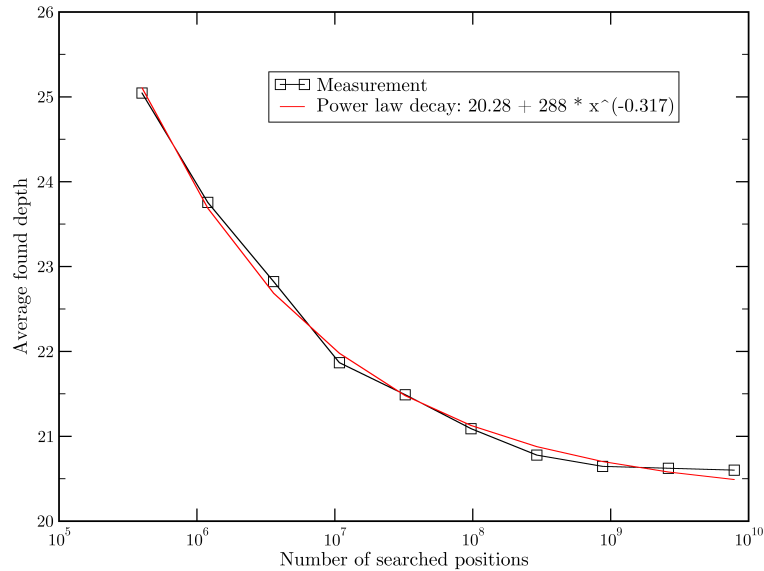


Figure 20: Shown is the average depth over all 45 differently scrambled cube as a function of the number of searched positions. This has been fitted with a power law decay function. The average optimal solution will thus probably lie between 20 and 21.

solution depth the program searches a thousand times longer. This also implies that it uses approximately a thousand times more memory. This fact shows that the approach of the greedy search algorithm is valuable. Run 11 was hand-picked because it has so many improvements.

Since the solution improves when searching longer, also the average depth of the found solutions (see Figure 20) decreases when searching more positions. The measured average from the 45 runs matches a power law decay function. The found average is approximately 20.6, but from the fitted and extrapolated power law decay function the average could be around 20.3. It is quite safe to assume that the average lies between 20 and 21. As there are cubes that are still improving after  $5 \cdot 10^9$  positions searched it takes long to compute many runs. Figure 20 is also good to show that it is not always practical to find the optimal solution. As this graph uses a logarithmic x-axis the average depth decreases really slowly. Between  $10^9$  and  $10^{10}$  positions searched, the average decreases only by 0.04. Most users are probably fine with having a solution that is one move away from the optimal solution.

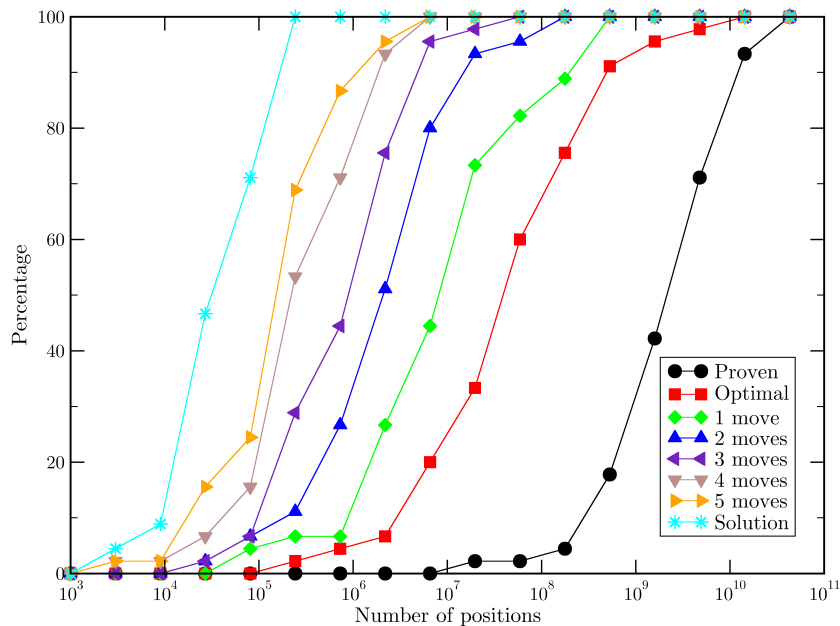


Figure 21: Percentage of positions where: the program found a solution, the program found a solution that is only 5, 4, 3, 2 or 1 moves away from the optimal solution, the program found the optimal solution, and the program has proven that the solution is optimal.

Figure 21 is a very interesting graph as many characteristics of the program can be

seen. In the 45 runs, the program found a non-optimal solution to all positions within  $10^6$  positions. Already at  $10^6$  positions there are solutions that are optimal. It takes quite some time until 50% of the cube have an optimal solution and even longer until they are all optimal. Solving all of them optimally lasts  $10^4$  times longer than finding a first solution. Proving that a solution is optimal is much harder. On the diagram can be seen that it takes 100 times longer to prove positions than to find an optimal solution. The memory usage of proving the most demanding solution peaked at over 260 GB. There are even positions (when searching for God's Number and not a part of this random set) where 512 GB RAM is not enough. Proving a solution is on the limit of even strong hardware.

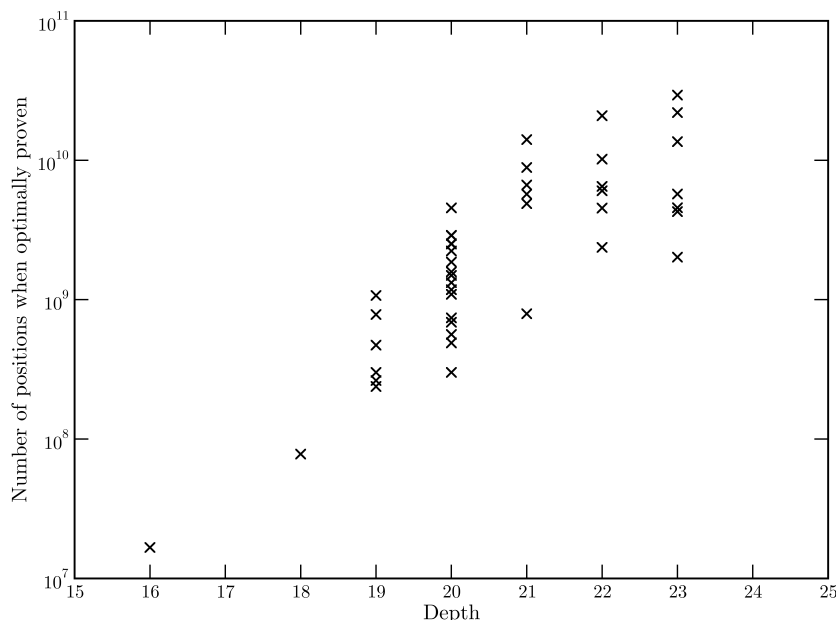


Figure 22: Shown are the number of positions needed to prove the optimal solution for 45 random positions. As it takes much time and memory to prove these positions only a small amount have been calculated.

Proving a solution is much easier when the depth of the optimal solution is lower (see Figure 22). This is only a general trend as there are positions with an optimal solution of depth 20 that needed much more positions to prove than a position with depth 23.

### 3.6.2 A Lower Bound for God's Number

Finally, a study was conducted to establish a lower bound for God's Number for the Puppet Cube V2. Searching through thousands of cubes, the longest proven solution was 30 moves long. The scramble presented in Table 2 depicts to the hardest found cube

with its reverse being an optimal solution. Interestingly, to prove this cube, only 260 GB memory and  $3 \cdot 10^{10}$  positions were needed. This is only as much as the highest cube with an optimal solution of depth 23. The key factor lies in the corner heuristic. The hardest cube had 27 moves as the corner heuristic. Not unexpectedly, hard cubes are difficult because the corners are in a difficult position. In the found example, the edge scramble contributes only 3 additional moves. Search for even higher depths were conducted from scrambled cubes that all had a starting corner heuristic of 27. Other positions were found with 31 moves as the solution, but optimality could not be proven as the program ran out of memory.

### 3.6.3 Comparison to standard Rubik's Cube solvers

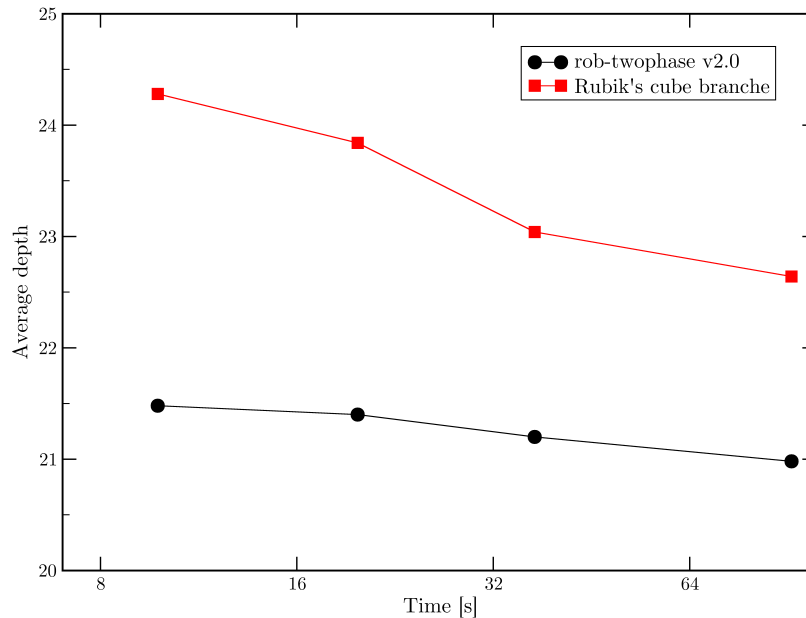
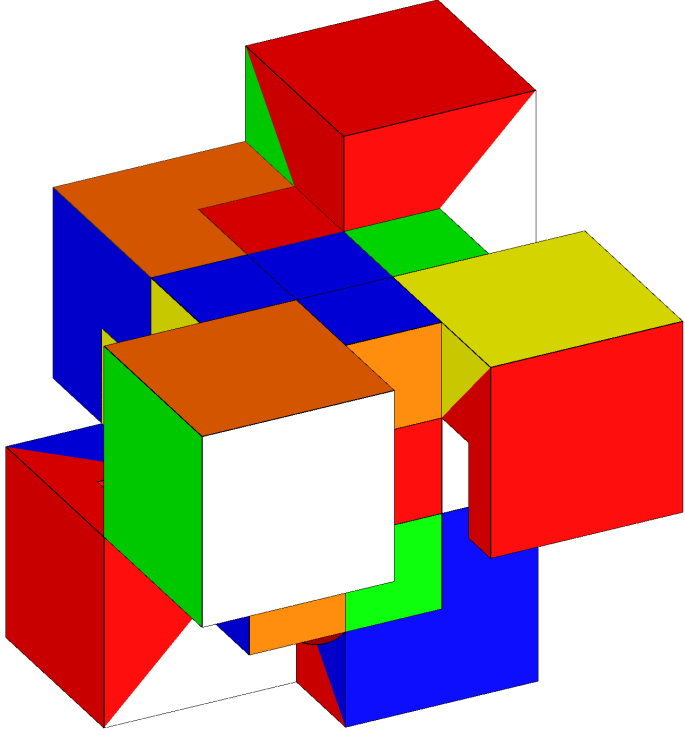


Figure 23: Comparison of the Rubik's Cube branch of the program with one of the best Rubik's Cube programs (rob-twophase v2.0). This average depth is measured in the quarter turn metric and without any slice moves. Both programs use 12 threads. The gap between these programs gets smaller with more time.

As there were no other programs for the Puppet Cube V2 found on GitHub, the Rubik's Cube was used as a comparison. Because the Puppet Cube V2 and the Rubik's Cube are both quite similar, the Rubik's Cube branch introduces no big changes. A branch on GitHub is a different version of the program where in this case the algorithm was rewritten to be able to solve the Rubik's Cube. In Figure 23, it can be seen that the implementation of the algorithm designed for the Puppet Cube is also really strong for solving the normal cube.

Table 2: Position with the longest proven solution. The shortest solution is 30 moves long and is the reverse of the scramble shown.

		
D' L' U' L F L F D' M' D L E' F M B' L B' U' R L B R F' F' L D D F' U' U'		
where	L L' R R' F F' B B' U U' D D' M M' E'	rotating the left face clockwise rotating the left face counterclockwise rotating the right face clockwise rotating the right face counterclockwise rotating the front face clockwise rotating the front face counterclockwise rotating the back face clockwise rotating the back face counterclockwise rotating the up face clockwise rotating the up face counterclockwise rotating the down face clockwise rotating the down face counterclockwise is equivalent to rotate R and L' at the same time is equivalent to rotate R' and L at the same time is equivalent to rotate U' and D at the same time
starting with	yellow orange blue	on the top on the front on the right



Although rob-twophase's algorithm relies on Herbert Kociemba's two-phase algorithm and combines many of the best tricks from other excellent implementations [24], the program finds on average only a solution that is 2 moves faster for a given short search time. As time increases this difference becomes smaller, ultimately both finding the same optimal solution. The two-phase algorithm first uses only a subset of the moves. This subset is reliant on the form of the Rubik's Cube graph, which has properties that differ from the Puppet Cube graph. A detailed explanation of Kociemba's algorithm can be found in the paper [2].

## 4 Conclusion and Outlook

The implemented visualization is a great tool to give the user a realistic picture of the cube. All pieces look like their real counterpart. The most complicated pieces have curved surfaces, and although they are only represented with triangles, the triangles are not noticeable. Lighting was used to let the object appear 3D and some interpolation of the face normals was used to get a smooth transition between triangles. Although there is no official support for transparency, different approaches have been discussed and partially implemented. The Puppet Cube V2 can be represented graphically in real time!

The developed search can efficiently calculate a short solution and is able to improve it over time. The implementation is also able to prove an optimal solution, even though the graph contains  $5 \cdot 10^{18}$  nodes. Multiple algorithms for the search have been discussed. The final implementation is explained in detail. Furthermore, properties of the cube as well as the algorithm were presented. The program is able to solve the Puppet Cube V2 with reasonable time and memory requirements!

The original research question has thus been answered positively. Further developments of the program could include implementing the transparency more accurately, improving the memory usage when proving a solution by not saving all positions to the visited map and generating really random positions, which account for the number of edges. More difficult would be finding God's Number for the Puppet Cube V2 and even prove this. For this to work, a totally different approach probably needs to be adopted.

As a new feature, the user interface could be extended with the ability to solve a scrambled cube only from a given position, without requiring a scramble sequence. This would allow the user to solve scrambled cubes where the rotations of the current position are not known. Finally, this could be combined with a scanning program, which could use camera vision to obtain the scramble of a cube.

## References

- (1) Contributors to Wikimedia projects Rubik's Cube - Wikipedia [https://en.wikipedia.org/w/index.php?title=Rubik%27s\\_\\_Cube&oldid=1264047600](https://en.wikipedia.org/w/index.php?title=Rubik%27s__Cube&oldid=1264047600) (accessed 01/01/2025).
- (2) Rokicki, T.; Kociemba, H.; Davidson, M.; Dethridge, J. C. *The Diameter of the Rubik's Cube Group Is Twenty*; tech. rep.; 2013.
- (3) Tomas Rokicki, M. D. God's Number is 26 in the Quarter Turn Metric <https://www.cube20.org/qtm> (accessed 15/12/2024).
- (4) MoYu Puppet one | two [http://moyucube.com/en/Product4\\_e.asp?option=view&ProductId=4437](http://moyucube.com/en/Product4_e.asp?option=view&ProductId=4437) (accessed 09/11/2024).
- (5) VandeVondele, L. puppet-cube-v2 <https://github.com/linusvdev/puppet-cube-v2> (accessed 10/05/2024).
- (6) Ivo 3D Rendering - a brief history of its evolution <https://www.tornado-studios.com/blog/3d-rendering-brief-history-its-evolution> (accessed 10/12/2024).
- (7) 3D Rendering: Where It All Began, history of 3d rendering – VisEngine <https://visengine.com/3d-rendering-where-it-all-began> (accessed 10/12/2024).
- (8) Bluentcad The History of 3D Rendering: From Teapots to Toy Stories <https://www.bluentcad.com/blog/history-of-3d-rendering> (accessed 10/12/2024).
- (9) Sutherland, I. E. *Sketchpad: A man-machine graphical communication system*; tech. rep. UCAM-CL-TR-574; University of Cambridge, Computer Laboratory, 2003.
- (10) Contributors to Wikimedia projects Utah teapot - Wikipedia [https://en.wikipedia.org/w/index.php?title=Utah\\_teapot&oldid=1255131529](https://en.wikipedia.org/w/index.php?title=Utah_teapot&oldid=1255131529) (accessed 10/12/2024).
- (11) Contributors to Wikimedia projects Z-buffering - Wikipedia <https://en.wikipedia.org/w/index.php?title=Z-buffering&oldid=1250489078> (accessed 11/12/2024).
- (12) Contributors to Wikimedia projects OpenGL - Wikipedia <https://en.wikipedia.org/w/index.php?title=OpenGL&oldid=1261631641> (accessed 11/12/2024).
- (13) An OpenGL library <https://www.glfw.org> (accessed 10/05/2024).
- (14) glad <https://github.com/premake-libs/glad> (accessed 10/05/2024).
- (15) De Vries, J. LearnOpenGL - Creating a window <https://learnopengl.com/Getting-started/Creating-a-window> (accessed 11/12/2024).

- (16) Rendering Pipeline Overview - OpenGL Wiki [https://www.khronos.org/opengl/wiki/Rendering\\_Pipeline\\_Overview](https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview) (accessed 12/12/2024).
- (17) OpenGL Shading Language - OpenGL Wiki [https://www.khronos.org/opengl/wiki/OpenGL\\_Shading\\_Language](https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language) (accessed 12/12/2024).
- (18) Transparency Sorting – OpenGL Wiki [https://www.khronos.org/opengl/wiki/Transparency\\_Sorting](https://www.khronos.org/opengl/wiki/Transparency_Sorting) (accessed 18/08/2024).
- (19) Everitt, C. Order-Independent Transparency <https://developer.download.nvidia.com/assets/gamedev/docs/OrderIndependentTransparency.pdf> (accessed 18/08/2024).
- (20) Louis Bavoil, K. M. Order Independent Transparency with Dual Depth Peeling [https://developer.download.nvidia.com/SDK/10/opengl/src/dual\\_depth\\_peeling/doc/DualDepthPeeling.pdf](https://developer.download.nvidia.com/SDK/10/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf) (accessed 18/08/2024).
- (21) Grigorescu, I. The Perron Frobenius Theorem and a Few of Its Many Applications [https://www.math.miami.edu/~igrigore/perron-frobenius\\_kruczek.pdf](https://www.math.miami.edu/~igrigore/perron-frobenius_kruczek.pdf) (accessed 02/01/2025).
- (22) greg7mdp parallel-hashmap <https://github.com/greg7mdp/parallel-hashmap> (accessed 24/12/2024).
- (23) cameron314 concurrentqueue <https://github.com/cameron314/concurrentqueue> (accessed 24/12/2024).
- (24) efrantar rob-twophase <https://github.com/efrantar/rob-twophase> (accessed 01/01/2025).

## List of Figures

1	Different spanned areas with the same four corners. . . . .	2
2	Visualization of the pieces of the Puppet Cube V2. . . . .	3
3	Flow chart of the window manager . . . . .	4
4	Lighting of the cube . . . . .	5
5	YUVAL Understanding the OpenGL triangle. <a href="https://www.thecodingnotebook.com/2021/01/understanding-opengl-triangle.html">https://www.thecodingnotebook.com/2021/01/understanding-opengl-triangle.html</a> (accessed 14/12/2024) .	5
6	Interpolation of the face normals . . . . .	6
7	Transparent cube with two outwardly indistinguishable positions . . . . .	7
8	Sorting transparent objects . . . . .	8
9	Distribution of the number of positions with the shortest path for solving the corners. . . . .	10
10	Distribution of a scrambled graph . . . . .	11
11	Meet in the Middle visualization . . . . .	13
12	Number of positions in the tablebase depending on the depth. . . . .	14
13	Effects of the heuristic factor upon the average depth and time usage . . .	16
14	Search speedup depending on the thread count. . . . .	17
15	Tablebase speedup depending on the thread count. . . . .	18
16	Flow chart of the main program and the search manager . . . . .	19
17	Flow chart of the solve function . . . . .	20
18	Flow chart of the search function . . . . .	23
19	Improvement of the best solution found depending on the number of positions.	24
20	Improvement of the average depth depending on the number of positions with a fitted power law decay function. . . . .	24
21	Percentage of positions where: the program found a solution, the program found a solution that is only 5, 4, 3, 2 or 1 moves away from the optimal solution, the program found the optimal solution, and the program has proven that the solution is optimal. . . . .	25
22	Number of positions needed to prove the optimal solution. . . . .	26
23	Comparison of the Rubik's Cube branch with other programs. . . . .	27

## List of Tables

1	Number of legal positions and edges of the Puppet Cube V2 . . . . .	9
2	Position with the longest proven solution . . . . .	28