Karlsruhe Institute of Technology

Engineering Optimal Solvers for Rubik's Cubes

Bachelor's Thesis of

Tan Grumser

at the Department of Informatics
Institute of Theoretical Informatics, Algorithm Engineering (ITI)

Reviewer:	Prof. Sanders
Advisor:	M.Sc. Dominik Schreiber

01. May 2022 – 31. August 2022

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 26.08.2022

. . . .

(Tan Grumser)

Abstract

Optimally solving a given Rubik's Cube is an extremely hard task even for a computer assuming the immense size of the problem space. In 1997 Korf presented a work that first solved random instances of cubes by using pattern database and IDA*. He hypothesised that the run time (measured in generated nodes) has an inverse linear relationship with the pattern database size. In this work we generated nearly one Terra byte of databases and used them to show that Korf's hypothesis still holds true for database sizes three orders of magnitude greater than the ones he used in his work. We also present two techniques to detect whether a state was visited in one iteration of IDA* before or not. The first technique precomputes all move sequences up to a certain length together with a flag that indicates whether another move sequence will traverse the corresponding state or not, to prune so called duplicate states during the solve. Since a move sequence with a length of 8 moves already has over ten billion different combinations the memory required is too large for some machines. We also introduced another technique that has a more optimal usage of memory with a small trade off for computations. It exploits the fact that two different paths that lead the same state when applied to the solved state also lead to the same state when applied to any other arbitrary state. We precompute and store all states (represented by an index) that are reachable with two different paths from the solved state and use a function that constructs the index of the state that has the same relative position the solved state, as some arbitrary position has to the scrambled state. With this we are able to access the precomputed duplicate states with the constructed state index. The latter techniques allowed us to improve the run time by about ten percent and achieve better performances than the best known implementation of Korf's algorithm. With all improvements we were able to solve 5 000 random cube instances in an average of 63 seconds.

Zusammenfassung

Das optimale Lösen des Rubik's Cubes ist durch die immense Größe des Problemraums, selbst für einen Computer eine extrem schwere Aufgabe. Mit seiner 1997 veröffentlichten Arbeit war Korf der erste, der zufällige Instanzen des Rubik's Cubes mithilfe von Zustandsdatenbanken und IDA* löste. In seiner Arbeit stellte er die Hypothese auf, dass die Größe dieser Datenbanken einen invers proportionalen Zusammenhang mit der Laufzeit (gemessen an generierten Knoten) hat. In dieser Arbeit generierten wir knapp ein Terrabyte an Zustandsdatenbanken und benutzten diese um zu zeigen, dass Korfs Hypothese auch für Datenbanken, die drei Zehnerpotenzen größer sind als jene, die er in seiner Arbeit benutzt hat, gilt. Wir presentieren außerdem zwei Techniken um Zustände zu erkennen, die in einer Iteration von IDA* bereits abgearbeitet wurden. Die erste Technik erzeugt eine Datenbank von allen Zugsequenzen zusammen mit einer Flag, die angibt ob der dazugehörige Zustand von einer anderen Zugsequenz abgearbeitet wird, um Äste die sogenannte Duplicate States (zu Deutsch: doppelte Zustände) darstellen bei der Suche abzuschneiden. Da es nach bis zu acht Zügen bereits mehr als Zehnmilliarden verschiedene Kombinationen gibt, ist diese Technik nicht für alle Maschinen geeignet. Wir stellen deshalb noch eine weitere speichereffizientere Technik vor, die einen etwas größeren Rechenaufwand besizt. Diese Technik nutzt die Tatsache, dass wenn zwei unterschiedliche Abfolgen von Zügen, die wenn auf den gelösten Zustand angewendet, zum selben Endzustand führen, auch für jeden anderen beliebigen Zustand zum selben Endzustand führen. Indem eine Tabelle vor der eigentlichen Suche berechnet wird, in der alle Duplicate States (durch einen Index repräsentiert) gespeichert sind, kann eine Funktion genutzt werden, die den Index des Zustands konstruiert, der die gleiche relative Position zum gelösten Zustand hat, wie ein beliebiger Zustand zu dem vermischten Startzustand. Damit können auf die im Vorhinein generierten Duplicate States mit dem konstruierten Index zugegriffen werden. Die beiden Techniken haben es ermöglicht die Laufzeit um ungefähr zehn Prozent zu verbessern und damit eine bessere Performance als die beste bekannte Implementierung von Korfs Algorithmus zu erlangen. Mit diesen Verbesserungen konnten wir 5 000 zufällige Zustände durchscnittlich in 63 Sekunden lösen.

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

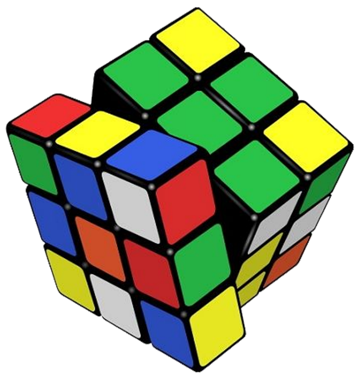

Figure 1.1.: A picture of a modern Rubik's Cube [Source: www.hiclipart.com]

The Rubik's Cube was invented in 1974 and named after it's creator the Hungarian architecture professor Erno Rubik [23]. He intended to find a task for his students while he was teaching geometry, but the cube he came up with quickly spread beyond his classroom. After filing a patent and releasing it to the public it became the fastest selling puzzle ever. [5] Departing from a randomly scrambled state, the goal of this puzzle is to transform the cube back to the state in which each face only shows one color by rotating the faces of the cube. When a trained human solves the cube he requires about 55 moves [10] to solve the cube. But it is known that every cube, with arbitrarily complex scrambles always can be solved in 20 moves or less [19].

To find the shortest sequence of turns, that transform the cube back to it's original state, specialized algorithms are required in order to deal with the enormous amount of of possible configurations of the cube. Only as recent as 1997 the first techniques for optimally solving the Rubik's Cube were published by Richard Korf [14]. In his work he used pattern databases that stored the shortest distance (from a scrambled to the solved state) of subgroups to estimate the real distance of an arbitrary state to the solved state. This estimate was used as a heuristic for Iterative-Deepening A* (IDA*). With that he was the first to find optimal solutions to random scrambles. Korf also hypothesized that the search space has a inverse linear relationship with the size of the pattern databases. This work concentrates on improving these algorithms and is structured as follows.

Chapter 2 introduces some basic notations that will be used throughout this work. Some basic properties of the cube will be presented together with the derivation of the size of the problem space. The chapter is concluded with the historically relevant achievement of finding God's Number (the greatest number of moves any scramble may take to be solved) for the Rubik's Cube.

Chapter 3 introduces three different representations of a configuration of the cube: Storing the 54 facelets of the cube, storing only the positions and orientations of corners and edges and storing only a four tuple coordinate for representing the cube. The chapter compares the advantages and disadvantages of these representations and explains how turns of the different layers can be applied to them. Next a naive approach is presented to show the complexity of the problem and to lay a foundation for more sophisticated techniques to optimally solve the cube. The pattern databases Korf used in his work as well as new databases that were used in the modern implementation of Korf's algorithm by Ben Botto are presented. How the pattern databases can be used with the IDA* algorithm is explained afterwords together with a short analysis and an optimality proof. At the end of this chapter Korf's hypothesis on the correlation between the size of pattern databases and the run time of IDA* will be shown to be true for the bigger databases used by Botto and us. Additionally this chapter can be seen as a survey on the different approaches that were developed to optimally solve the cube.

In Chapter 4 we present our contributions which consist of using a 1 TB RAM machine for the pattern database containing all 980 995 276 800 different arrangements of the edge pieces - which to the best of our knowledge has not been generated before, as well as two techniques to detect duplicate states to decrease the number of nodes that need to be generated when optimally solving a scramble. The run time of our program with the edge pattern database shows that Korf's hypothesis still holds true for pattern database sizes three order of magnitudes larger than the ones he used in his original work. The first technique for duplicate state detection is to store an all sequences of moves up to some sequence length bound and to attach a flag to all entries that indicates whether the state the sequence leads to is found by another sequence of moves. The keys for this table are indices that be computed from a sequence of moves. The other technique uses a state shift function to construct the state that would be reached if the transformation that brought a scrambled state to the current state in the search, would be applied to the solved state. This can be used to index a hash table that holds all states that can be reached through different move sequences (up to a certain sequence length bound) from the solved state and check whether a certain state has been traversed before or not. The whole edge configuration pattern base together with the duplicate state detection allowed us to solve each of 5 000 random scrambles in 63 seconds on average.

In Chapter 5 we take a closer look at some critical parts of the implementation, illuminate some implementation techniques used to improve the performance and explain how we parallelized IDA* and the pattern database generation.

More detailed results are presented in Chapter 6. We show what contribution of the whole edge lookup table and the duplicate state detection have on the search in isolation. Then we discuss these results.

Chapter 7 concludes everything presented thus far and gives an overview of future work that can be done in this topic. E.g. to generate other lookup tables that take corners and edges into account together instead of considering them in isolation. It briefly discusses other combinatorial puzzles that the duplicate state detection can be used for like the 15 tile puzzle and finishes with some thoughts on how to solve even bigger cubes.

2. Preliminaries

In this chapter we introduce some notations used throughout this work as well as some basic concepts that build the required knowledge base for the following chapters.

2.1. Notation

We define a move on the Rubik's cube by the rotations of a single side by an arbitrary angle. This type of counting moves is called face turn metric. There are other metrics like the quarter turn metric, which counts 90° turns as single move and therefore 180° turns as two moves. This means that rotating a face 180° only counts as one move throughout. Throughout this work we make use of two notations. One to refer to one of the six faces of the cube and their rotation and one to refer to the smaller pieces that make up the cube. The Singmaster notation, developed by David Singmaster, is a method for denoting moves that can be made on the Rubik's Cube [7]. Let the cube face its user with the top and bottom layer vertical and the front face pointing at him. In the following list the symbols denote a clockwise rotation of a layer, clockwise meaning rotating as if the layer was facing the user.

- **U**: rotates the top layer clockwise by 90 degrees.
- **D**: rotates the bottom layer clockwise by 90 degrees.
- **F**: rotates the front layer clockwise by 90 degrees.
- **B**: rotates the back layer clockwise by 90 degrees.
- **R**: rotates the right layer clockwise by 90 degrees.
- **L**: rotates the left layer clockwise by 90 degrees.

To denote a rotation in the counter clockwise direction we add an apostrophe after the symbol (e.g. U'). To denote a rotation by 180 degrees a 2 is added to the symbol (e.g. F2).

To refer to a single piece or a position of the Rubik's Cube the surrounding layer's symbols will be used. We provide some examples:

- **URF**: refers to the corner in the top right front.
- **DBF**: refers to the corner in the bottom right front.
- **BL**: refers to the edge that sits between the back and the left layer.

- **UF**: refers to the edge that sits between the up and the front layer.

Note that we will use this notation not only to refer to a pieces but also to refer to positions on the cube. To denote that we are referring to a piece we write "the UFR corner" or "the UF edge". To refer to a corner position we write "at position URF" or "at position UF" to refer to an edge position. On a solved cube at position XYZ always lies the XYZ corner (e.g. the UFR corner lies at position UFR on a solved cube) and at position XY always lies the XY edge (e.g. the UF edge lies at position UF on a solved cube). This notation allows to disregard the color scheme of a cube and makes it irrelevant which color is defined to be on top and on front.

2.2. Rubik's Cube Basics

The Rubik's cube consists of 26 cubies - 8 corner cubies, 12 edge cubies and 6 center cubies. Each face of the cube can be rotated by 90°, 180° or 270° (equivalent to 90° in the opposite direction). Such a turn affects the position of 4 corner cubies and 4 edge cubies and rotates one center cubie. Each corner cubie can have three different orientations and can occupy every corner position. The edge cubies have two possible orientations and every edge cubie can be in any of the twelve edge positions. The center cubies can be in one of four orientations, which are visually indistinguishable when dealing with regular Rubik's Cubes. Some cubes have pictures on the stickers (picture cubes) and require to also have the center pieces in a certain orientation, but for this work we will only consider cubes without pictures on their stickers. When assembling a cube, the first edge cubie can be inserted in one of twelve edge positions, the second edge cubie in eleven and so on. This yields 12! possible permutations of the edges. Similarly the eight corners have 8! possible permutations. Each edge has two orientations which leaves the cube with 2^{12} possible configurations of edge orientations and 3^8 configurations for the corners. For regular cubes this results in $12! \cdot 8! \cdot 3^8 \cdot 2^{12} \approx 5.1902404 \cdot 10^{20}$ possible configurations when assembling the cube. Not all of these configurations are reachable as it will be shown in the next section.

2.3. Laws of the Cube

There are some restrictions to configurations that are reachable through face turns. A single corner's orientation cannot be changed without another corner's orientation being changed as well. Also, a single edge cannot change its orientation. The sum of all corners and all edges have to be divisible by three and two respectively. Besides those two restrictions of corners and edges that apply to both types of cubies even in isolation, when considering the positions of corners and edges no two corners can be swapped without either swapping two more corners or swapping two edges. The same applies for the edges. This is due to the parity properties of the cube [21, p. 7]. The orientation of a center cubie also cannot be changed in isolation which divides the number of orientations of the center cubies by two. The corner restriction divides the number of reachable configurations by three. The edge restriction divides the number by two and the third restriction also divides the number by two. This means that we can only reach $\frac{1}{2\cdot3\cdot2} = \frac{1}{12}$ of all configurations when using face turns. In total there are

$$\frac{12!\cdot 8!\cdot 3^8\cdot 2^{12}}{2\cdot 3\cdot 2} = 43\,252\,003\,274\,489\,856\,000 \approx 4.3\cdot 10^{19} \tag{2.1}$$

possible configurations of the Rubik's Cube. On picture cubes the six center cubies could be in 4^6 orientations, but with regular moves only half of these configurations can be reached. Therefore the number of configurations is 2048 times larger for picture cubes.[6, p. 46f]

2.4. God's Number is 20

One important question about the Rubik's Cube regarding its complexity is what are the hardest configurations that one may solve and what is the least amount of moves it would take to solve those configurations. In combinatorial puzzles with a finite number of configurations, an algorithm that always yields the shortest sequence of steps resulting in the 'solved' configuration is sometimes referred to as 'God's algorithm' - a term coined by John Conway in the discussion on the Rubik's Cube [20, p. 26]. In connection to this the upper bound of number of moves optimal solutions have is sometimes referred to as 'God's number'. [11] A simple way for finding a lower bound for this number is to see how many configurations are reachable after a certain number of moves and comparing it to the number of all possible configurations. After one move, for example, there are 18 unique configurations that can be reached. This number arises because each of the six faces can be turned in three distinct ways. Since 18 is smaller than the total amount of possible configurations, there have to be configurations that need two moves to be solved. Extending this logic by assuming how long a sequence of moves needs to be to contain more combinations of moves than the number of configurations, one can search for the smallest n which satisfies

$$\sum_{i=0}^{n} 18^i \geq 4.3\cdot 10^{19} \tag{2.2}$$

With $\sum_{i=0}^{15} 18^i \approx 7.14\cdot 10^{18} < 4.3\cdot 10^{19}$ and $\sum_{i=0}^{16} 18^i \approx 1.29\cdot 10^{20} > 4.3\cdot 10^{19}$, it is found that n = 16 is the smallest integer, that satisfies equation 2.2. So 16 must be a lower bound for God's number [8]. With this method the lower bound of God's number can be increased to 18 by using more precise ways to count the number of reachable states after n moves. Some improvement for counting this number is introduced in Chapter 3.

An upper bound was found in the 1980 by Morwen Thistlewaite [22]. He developed a method to solve Rubik's Cube suboptimally and proved that his method would never require more than 52 moves. With that, an upper and a lower bound were known for God's number. But the quest of bringing them closer together took three decades to be completed. In 1995 Michael Reid showed that there is a configuration of the cube that cannot be solved in less than 20 moves. The configuration he used is called the *superflip* - every corner is at its solved position and has its correct orientation and all edges are

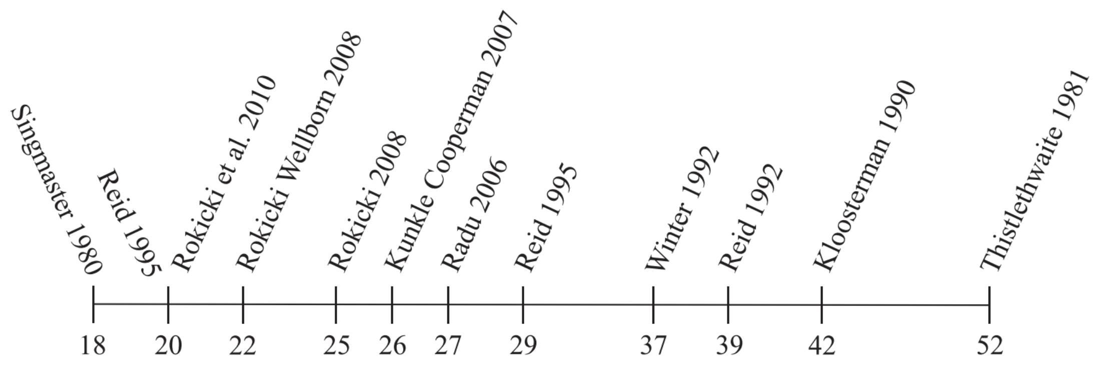

Figure 2.1.: Timeline of the progress on finding God's number [11]

in their solved position as well, but they are oriented the wrong way. He used special properties of this position to improve the run time of an optimal solver that he used to show that the shortest solution has length 20. Herbert Kociemba came up with a very fast algorithm that could solve cubes nearly optimally. Reid showed that this method will always find solutions with length 29. [18] Tomas Rokicki worked on further improving the upper bound since 2003 and showed in 2008 that no position ever needs more than 27 moves to be solved. [11, 18] This bound was lowered to 25 in the following years.

Rokicki worked together with the two Mathematicians Kociemba and Reid and the computer scientists Silviu Radu, Richard Korf, Gene Cooperman, Dan Kunkle and some others who contributed to the problem of finding God's Number. In 2008 progress on this problem was so significant, that Sony's John Welborn contacted Rokicki and offered him CPU time on the farm of computers that Sony Pictures Imageworks uses to render animated movies. With this Rokicki was able to extend the method used in his original 25-move upper bound to show that, every position can be solved in at most 22 moves. After those results Google donated 35 CPU years on their supercomputers. With this the number could be brought down to 20 and God's Number was found. [11, p. 265] [19] Four years later Rokicki and Morley Davidson also proved that God's number is 26 in the quarter turn metric. [9]

All contributions to improve the bounds are listed in figure 2.1.

3. Prior Algorithms

Solving a Rubik's Cube optimally can be mapped to a shortest path graph problem. A node represents a single state of the cube and an edge represents one of the 18 possible moves. The graph resulting from this construction can be arranged to a tree, with a scrambled state as the root, the 18 adjacent states at depth one and all states reachable with two moves at depth two. Naively this tree would have a branching factor of 18, but since a sequence of moves of the same face can always be substituted with one move (e.g. L L = L2, L2 L = L') the branching factor can be reduced to 15 (except for the root node). We can further improve this number by realising that opposite face turns are commutative (e.g. L R = R L, U2 D' = D' U2). This means that for every node with the upper edge representing a D move we can dismiss all subsequent moves of the U face since the state that would be reached with these moves are equivalent to the states that will be reached from the node with upper edge representing a U move at the same depth and moves of the D face. Take, for example, the path R U D. The path along R D U will result in the same state. In general, when generating the graph, the previous move P of any node can be stored to prune all edges that [14]:

- rotate the same layer as P
- rotate the D layer if P rotated the U layer
- rotate the B layer if P rotated the F layer
- rotate the L layer if P rotated the R layer

There are still some redundant states that can be reached with two different paths, such as U2 D2 F2 B2 and F2 B2 U2 D2. These paths yield the same state, but are not as easy to detect with such a rule. States that can be reached via such paths will be called duplicate states throughout this work. With the above described pruning rules the branching factor of the tree search reduces even further. Exact numbers of nodes at certain depths can be found in table 3.1. These rules result in a search tree with an asymptotic branching factor of about 13.34847. In can be seen that when applying these rules only at a depth of 18 the number of reached states exceeds the number of possible configurations. As described in section 2.4 this shows that 18 is a lower bound for God's Number.

3.1. Rubik's Cube Representation

When constructing the graph of configurations of the cube, it is required to have some representation of a state of the cube as well as a representation of moves that can be applied

depth	nodes
1	18
2	243
3	3,240
4	43,254
5	577,368
6	7,706,988
7	102,876,480
8	1,373,243,544
9	18,330,699,168
10	244,686,773,808
11	3,266,193,870,720
12	43,598,688,377,184
13	581,975,750,199,168
14	7,768,485,393,179,328
15	103,697,388,221,736,960
16	1,384,201,395,738,071,424
17	18,476,969,736,848,122,368
18	246,639,261,965,462,754,048

Table 3.1.: Nodes in a Rubik's Cube search tree as a function of depth [14]

to the cube. Since applying moves to the cube when traversing the graph is one of the most frequently invoked methods, it is crucial to use a performance optimized approach to represent the cube and applying moves to it. In the following, three approaches are presented.

3.1.1. Facelets

The most intuitive approach to represent a configuration of the cube is to specify which colors lie on the $6 \cdot 9 = 54$ squares of the cube. One colored square will from here on be referred to as a facelet. All facelets as well as all names for them can be seen in Figure 3.1. To store the colors on each facelet, a two-dimensional array with dimensions 6×9 can be used, where each color is described by a value between 0 and 5, representing a unique color. When turning a face of the cube, three facelets of four different sides will cycle as well as the eight facelets of the side that is turned. For a F move, for example, the facelets groups (U7, R1, D3, L9), (U8, R4, D2, L6), (U9, R7, D1, L3), (F2, F6, F8, F4) and (F1, F3, F9, F7) will cycle. The indices of the facelet groups that get cycled for each of the 18 moves can be stored in lookup table and used in a function for applying the move. The center facelets cannot be moved with regular face turns, so if the program only supports such

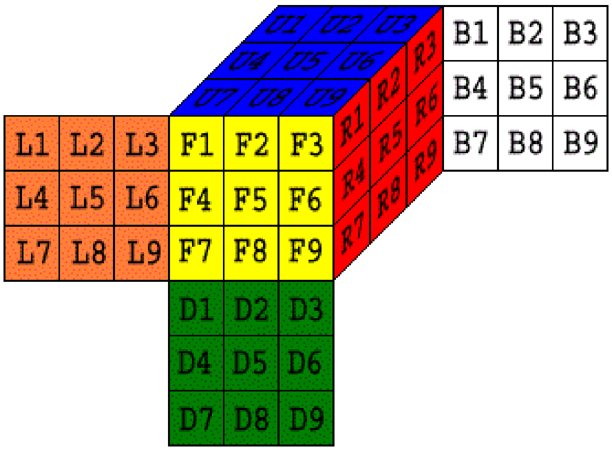

Figure 3.1.: Facelets [Source: kociemba.org]

regular moves[1] these facelets' colors can be omitted in the array that represents the state of the cube. With that the final dimensions of the array are 6×8.

This form of representing the state of the cube is not optimal, since corners and edges have three and two facelets respectively that will never be separated. Therefore, a redundancy of information on the state of the cube arises. Even if memory constraints are irrelevant, moving more data than necessary results in computational overhead which can be saved with the representation introduced in the next section.

Early iterations of our program made of use this representation but the approach had to be discarded since the performance of applying moves wasn't good enough. As stated, too many operations were required for modifying the state by only one move. On the other hand, this representation has the benefit that passing in information about the state of a real Rubik's Cube is easier than with the representations introduced in the next sections. Also, for some forms of displaying a digital cube graphically, the easy lookup of the color of a facelet can be beneficial.

3.1.2. Cubies

As stated in the last section the facelets of corners and edges cannot be separated, which motivates a representation that only stores the orientation and position of corner and edge cubies. The orientation of corners can be described by number between 0 and 2. The orientation of edges can be described by a binary value. The position of the corners and edges can be represented by numbers in the range from 0 to 7 and 0 to 11 respectively. But since rotating any side of the cube needs to cycle pieces at specific positions, it is computationally more reasonable to store the index of a piece at a certain position. Hence, an array of size 8 with entries containing the index as well as the orientation of each corner uniquely describe the state of the corners of the cube. The first element in the array describes the position UBR on the cube and the next seven entries describe positions UFR, UFL, UBL, DBR, DFR DFL and DBL respectively. Analogously the edges can be stored in

[1]Rotating the whole cube is no regular move. Also, turning any middle slice is no regular move but can be achieved by rotating the two adjacent face in the opposite direction.

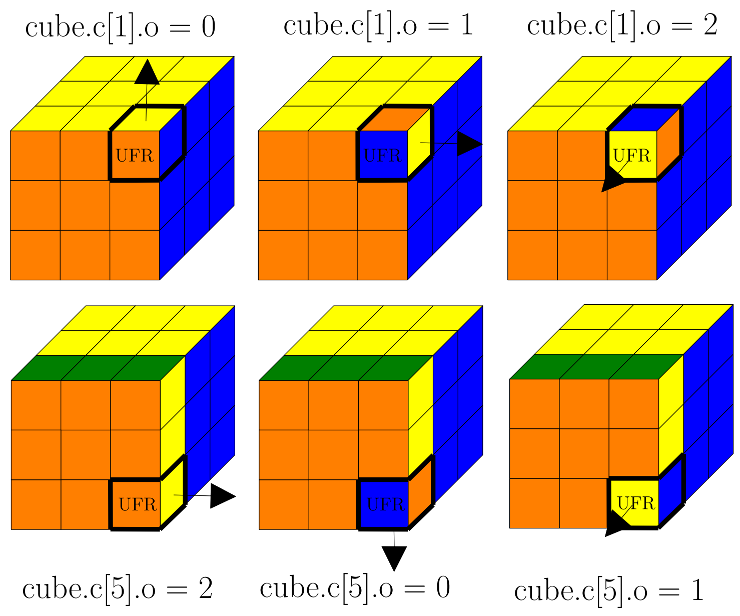

Figure 3.2.: Corner Orientations at different Positions

an array with twelve entries. The twelve entries of this array refer to the positions UR, UF, UL, UB, RB, RF, LF, LB, DR, DF, DL, and DB. The corner positions can be translated to indices by assigning the number 0 to 7 to the positions in the order they were enumerated (position UBR becomes 0, position UFR becomes 1, and so on). The numbers 0 to 11 can be assigned to the edges positions the same way (position UR becomes 0, position UF becomes 1, and so on). Similarly we can assign an index to a certain corner. So the UBR corner (disregarding its current position) has index 0. We denote the a corner of the cube at position index x with cube.c[x] and edges at position index y with cube.e[y]. The index of the piece at and its orientation will be denoted with cube.c[x].i and cube.c[x].o respectively. So cube.c[4].i = 2 means that at the position DBR sits the UFL corner.

To rotate a face of the cube four entries of the corner array have to be cycled and the orientation of these entries have to be adjusted. The same has to be done for the edges. For the change of the orientation of the pieces again a lookup table can be used. But in our implementation the 18 moves are split up in 18 functions, which each encapsulate the information of the orientation changes.

Orientation of a piece at a certain position alone is not sufficient to describe what the actual cube would look like. For the corners, every position on the cube needs to have some interpretation of the orientation of a piece. For instance, consider the UFR piece of the upper left picture in Figure 3.2. It is stored in position one of the corner array. For this piece at this position we define orientation 0 as the solved orientation. This means that for orientation 0 (for this specific color scheme) the yellow facelet is on the top layer. Conceptually we defined an upper facelet for the corner and an upper side for each corner positions and the orientation indicates how these two are orientated relative to each other. Increasing the orientation by one is equivalent to rotating the corner by 120° clockwise (when looking at the piece from outside of the cube), which the other two depictions of the cube in the first row of Figure 3.2 show.

Analogously we can define the upper side of all remaining corners positions and the upper facelets for the remaining corner pieces. For the corner positions adjacent to the top layer an orientation of 0 refers to the upper facelet of a corner being on the top face and for the remaining positions at the bottom an orientation of 0 refers to upper facelet being on the bottom face. Let the the corners adjacent to the top layer on a solved cube have their upper facelets on the top layer and the corners laying on the bottom face be defined to have their upper facelets in the bottom layer (For this color scheme, white is on the bottom layer, so the yellow and white facelets are the upper facelets of the corners). The same applies to the edges. We define the the upper facelet of edges adjacent to the top and bottom layer the same way. Let the remaining edges' upper facelet be defined to be on the right and left layer. As we will show in Chapter 5 the selection of upper facelets and upper sides affect the performance since more adjustments of orientation then necessary can result from a poor selection.

Figure 3.2 provides an example of how the position and orientations change when a move is applied. The states in the second row are the results of an F turn being applied to the states above them. When this move is applied to the solved state, the UFR corner which had position 1 in the array will go to position 5. The orientation of the piece will increase by 2, since the upper facelet of the corner which would face down with orientation 0 needs to be rotated two times by 120° to face right.

The representation introduces in this section is the one used in our implementation.

3.1.3. Coordinates

An even more sophisticated approach for representing the configuration of the cube is to store a single coordinate to represent the state of the cube. This coordinate consists of the four scalar values$(C_{orientation}, C_{perm}, E_{orientation}, E_{perm})$, each describing different parts of the cube. $C_{orientation}$ describes the orientation of all eight corners, $E_{orientation}$ describes the orientation of the twelve edges and C_{perm} and E_{perm} describe the permutation of corners and edges respectively.

As seen earlier, all eight corners can be in $3^8 = 6561$ different orientation arrangements. Each of these arrangements can be mapped to a number between 0 and 6560, but since a third of these orientation arrangements cannot be reached with regular move (no corner orientation can be changed independently) this value range shrinks to $3^7 = 2187$ possible values. To calculate this number, seven orientations can be written together and read like a ternary number. The edge orientation coordinate can be calculated analogously by constructing an eleven digit binary number from the edge orientations. This number ranges form 0 to $2^{11} = 2048$.

The number of permutation of the corners equals $8! = 40320$. Mapping an index to all these permutations is slightly more complex. As in Section 3.1.2 all eight corners receive an unique index between 0 and 7. Then, from the UBR corner in clockwise order and from the DBR corner also in clockwise order (or any other preferred order), listing out the corners at these positions (e.g. URF, UFL, ULB, UBR, DFR, DLF, DBL, DRB) and replacing the corner with its corresponding index results in a permutation of eight unique numbers.

For this permutation its corresponding Lehmer code can be calculated. The Lehmer code is a special way to encode permutations.

Let σ be a permutation with n elements where σ_i is the i-th element of the permutation. The Lehmer encoding $L(\sigma)$ for a permutation σ can be computed with

$$L(\sigma) = (L(\sigma)_0, \ldots, L(\sigma)_n) \quad \textbf{where} \quad L(\sigma)_i = \sigma_i - \{j < i : \sigma_j < \sigma_i\} \tag{3.1}$$

For each element at position i, its index minus the number of elements to the left with a smaller index becomes the coefficient $L(\sigma)_i$. We provide the example of how the permutation of corners look like after an F move is applied:

Position	URB	URF	ULF	ULB	DRB	DRF	DLF	DLB
Corner at position	URB	ULF	DLF	ULB	DRB	URF	DRF	DLB
Index at position	0	2	6	3	4	1	5	7
i	0	1	2	3	4	5	6	7
$L(\sigma)_i$	0	1	4	1	1	0	0	0

The number with digits constructed from the $L(\sigma)_i$'s ($L(\sigma)_0$ at highest value position and $L(\sigma)_7$ at the lowest value position and all other digits in descending order in between) can be interpreted using the factorial number system to compute the final permutation index. To convert the permutation of corners σ_c and permutation of edges σ_e to the decimal numbers C_{perm} and E_{perm} the following equations can be used.

$$C_{perm} = \sum_{i=0}^{7} L(\sigma)_i \cdot (7-i)! \qquad E_{perm} = \sum_{i=0}^{11} L(\sigma')_i \cdot (11-i)! \tag{3.2}$$

In the example the index of the lexicographic rank of the corner permutation comes out to be

$$0 \cdot 7! + 1 \cdot 6! + 4 \cdot 5! + 1 \cdot 4! + 1 \cdot 3! + 0 \cdot 2! + 0 \cdot 1! \cdot 0 \cdot 0! = 1230 \tag{3.3}$$

To compute $L(\sigma)_i$ all elements with a smaller index to the left of position i need to be counted. Since $O(n)$ elements need to be counted and n Lehmer digits need to be computed, this calculation has a time complexity of $O(n^2)$. In 2005 Korf et. al. published paper titled "Large-Scale Parallel Breadth-First Search" [16] in which they described a linear algorithm for computing lexicographic ranks. The algorithm uses a bitmap where the i-th bit is set when the coefficient for the element i is computed and a lookup table which is indexed by the bitmap used as a binary number and has the number of one of this binary number stored. The details as well as an implementation will be omitted in this work but we use the algorithm in our implementation. Ben Botto published "Sequentially Indexing Permutations: A Linear Algorithm for Computing Lexicographic Rank", an article with further details on the algorithm [3].

Now any configuration of the cube can be uniquely represented by the four-dimensional coordinate $(C_{orientation}, C_{perm}, E_{orientation}, E_{perm})$. It has to be noted, that due to the laws of the cube coordinates exist that correspond to states that cannot be reached with regular moves, since permutations of corners and edges are separated and for some configurations of the edges certain permutations of corners cannot be reached and vice versa. To transition to an adjacent cube state, the approaches in the last two chapters cannot be used anymore since there is no more representation of single cubies that could be moved around. To resolve this problem, lookup tables which contain all adjacent states for all different values of the coordinates are required. For example, the corner orientation can be in 2187 different states and for each one the 18 adjacent states must be saved in a lookup table, which will have $2087 \cdot 18 = 37566$ entries. The edge orientation lookup table would have $2048 \cdot 18 = 36\,864$ entries, whereas the corner and edge permutation lookup tables would have $40320 \cdot 18 = 725760$ and $479\,001\,600 \cdot 18 = 8\,622\,028\,800$ entries respectively. If one entry is saved in one byte the last lookup table would take about 8 GB.

3.2. A Naive Approach

Algorithm 1 IDDFS

```
function IDDFS(cube)
    $moveStack \leftarrow []$
    $bound \leftarrow 1$
    while not BoundDFS(cube, 0, bound) do
        $bound \leftarrow bound + 1$
    end while
    return moveStack
end function

function BoundedDFS(cube, depth, bound, moveStack)
    if depth > bound then
        return False;
    end if
    if IsSolved(cube) then
        return True;
    end if
    for move={U , U’, U2, ..., D, D’, D2} do              ▷ Loop trough all possible moves
        $ApplyMove(cube, move)$
        moveStack.push(move)
        if BoundedDFS(cube, bound, depth + 1) then
            return True;
        end if
        moveStack.pop()
        $ApplyMove(cube, Inverse(move))$
    end for
    return False
end function
```

A simple candidate for finding shortest paths is breadth-first search (BFS). It has a space complexity of $O(|V|)$ and a time complexity of also $O(|V|)$, where V is number of nodes generated for finding the solution. With a branching factor of about 13, configurations with a solution sequence with more than 10 elements already become nearly impossible to solve on normal hardware. Even if states could be stored in one byte, over 240 GB of memory would be required in the worst case. For solutions of lengths ≥ 15 run time gains relevance as well. If one billion nodes could be traversed per second, such a scramble would require over 90 days to be solved.

A* is an algorithm for finding shortest paths in a graph. It uses a heuristic and a cost function f(n) = g(n) + h(n), where g(n) is the distance from the start node to the current node and h(n) is a heuristic that estimates the remaining distance to the destination. By traversing nodes sorted by the cost computed with f(n) A* can find the shortest paths with fewer nodes than a BFS. Though informed searches like A* can decrease the time complexity, they can similarly require prohibitive amounts of memory. Therefore an approach with a space complexity that is sublinear in the traversed states is required. A depth-first serach (DFS) has close-to-constant space complexity for our means but does not guarantee optimal solutions. But with some modifications a DFS-based approach can still be used. Iterative deepening DFS (IDDFS) is an algorithm which borrows the two properties of optimality and close to constant space complexity of BFS and DFS respectively. It works by performing a DFS up to certain depth bound. If no solution is found, the depth bound is increased. IDDFS guarantees to find optimal solutions , since if a path with length n exist it will be found when searching with a limit of depth n and any longer paths cannot be found beforehand because they would only be traversed later in the search with a greater depth limit. Pseudo code of the IDDFS algorithm can be found in Algorithm 1. The pseudo code here as well as all other instances will opt to use recursive version of algorithms for didactic reasons and more readability.

In the IDDFS function, the cube parameter is a configuration of the Rubik's Cube. The *IsSolved* function returns true if the passed cube equals the solved cube. *ApplyMove* modifies the state of the passed cube state by applying the passed move. *Inverse* inverts the move that is passed in (e.g. L becomes L' U2 stays U2).

This algorithm has a space complexity of $O(b \cdot k)$, where k the depth at which the solution can be found and b is the branching factor. Since no pruning rules are used, b = 18 and k is at most 20. The time complexity is $O(b^k)$. For shuffles with a solution length of less than ten and with the pruning rules described in chapter 2.2, early tests have shown that the program will halt in less than a couple hours. But without any further information that can be used for pruning, the run time of the program is too big to generate solutions for harder scrambles in reasonable time.

3.3. Pattern Databases

As described in the last chapter, solving scrambled Rubik's Cube configurations requires a near to constant space complexity approach that makes use of some information of the graph to reduce the number of states that need to be traversed. A solution for the first requirement was already presented, but what information about the state of a cube can be used to reduce the size of the search space? To estimate the distance to the solution state a lookup table for the number of moves it takes to solve partial problems can be used.

A partial problem of the Rubik's cube can be solving subset of pieces of the cube. One could for example ask how many moves it would take to only solve only the corners of a cube and take this information to better estimate the number of moves it takes to solve the whole cube. This information can be used as a heuristic for an informed search. In the following sections pattern databases that are used in Korf's, Botto's and our solver are presented.

To use this information the lookup tables for several partial problems need to be generate beforehand and can then be used for all solves of cubes afterwards, amortizing the generation costs. To generate any lookup table the distance from the solved node to all other nodes of the new graph that arises from inspecting only a subset of pieces and their configuration needs to be calculated and then stored. To do so, some sort of algorithm for finding shortest paths is needed. Although a BFS could be used a IDDFS was chosen to generate all lookup tables presented in this work.

Before presenting the algorithm to generate the lookup tables the question of how to order and how to then access the information in the final search needs to be answered first. To be able to access the stored distances it is required to map some key representing the configuration to a value. A quick and naive approach could be to write out the order of pieces of interest and attach the string of orientations to them. This would generate unique keys but would not allow quick access since some sort of hash table would be needed to access the entries. So optimally a function should be used that takes in a state of the cube and yields a unique number in the range of zero to the number of entries the lookup table will have. In section 3.1.3 the concept of lexicographic ranks for permutations of n distinct elements and n non distinct elements was presented. These two methods together can be used to develop a function that suffice the stated requirements. We saw that a coordinate unambiguously can describe the state of the orientations and permutation of all corners. These two can be combined into one number with the same properties for all configurations of the corners. Let the number a bidirectional mapping from a configuration of any subset of pieces of the cube to this number be called the index of this state. For the corner example this index can be calculated from the corner permutation and corner orientation coordinate with the following equation

$$I_c = C_{permutation} * N_{co} + C_{orientation} \tag{3.4}$$

where $N_{co} = 3^7 = 2187$ is the number of possible corner orientation arrangements reachable with regular moves. I_c lies in the range from 0 to $3^7 \cdot 8! = 88.179.840$. In the following subsections it will be explained in detail how to calculate this index for the configuration of the corners, the configuration of a subset of the edges and the permutation of edges disregarding their orientation. Lets assume that a generic function *CalculateIndex* exist, that will return the index for a subset of pieces. This together with *numStates*, the number of states reachable in this subset, an algorithm for generating lookup tables for this partial problems can be constructed. An implementation for this can be found in Algorithm 2.

Algorithm 2 Generate Lookup Table

```
$foundStates \leftarrow 0$
$lookupTable \leftarrow array[numStates]\{-1, -1, \ldots, -1\}$

function GenerateLookupTable
    $bound \leftarrow 0$

    while not $foundStates == numStates$ do
        $boundedDFS(solvedCubeState, 0, bound)$
        $bound \leftarrow bound + 1$
    end while

    return moveStack
end function

function boundedDFS(cube, depth, bound)
    if depth > bound then
        return
    end if

    $index \leftarrow calculateIndex(cube)$

    if lookupTable[index] == -1 then                    ▷ Set the entry when not set yet
        $lookupTable[index] \leftarrow depth$
    end if

    for move={U , U’, U2, ..., D, D’, D2} do            ▷ Loop trough all possible moves
        $ApplyMove(cube, move)$
        BoundedDFS(cube, bound, depth + 1)
        $ApplyMove(cube, Inverse(move))$
    end for
end function
```

3.3.1. Optimization

The algorithm in this form is too slow to generate the tables in reasonable time, since as it will seen entries up to depth 14 will be need to be populated. This means that all states up to depth 14 need to be traversed. As seen in table 3.1 this would be 7 768 485 393 179 328 nodes. But there are some observations that can be used to drastically improve the performance of the population of the lookup table.

3.3.1.1. Two paths, different length, same state

The first observation is that if a state s is reached with depth d and an entry at its index with a depth d’ that is greater than d the traversal can be stopped. This is because, if s can be reached with d moves then all subsequent states at subsequent depths can be reached with $d + \epsilon$ moves, where ϵ is the difference of d and one of the subsequent depths. And since $d + \epsilon < d' + \epsilon$ no state will be reached with a depth smaller or equal to what the lookup table has already have stored at the state's index.

We provide an example of when this is happening. Consider the path p = U2 D2 R2 L2 U2. This path results in the same state as the path p' = D2 R2 L2 does. Since the length of p is greater than p', all subsequent states reached from p will never yield smaller depths than those reached via p' and therefore any further traversal from p cannot result in any state found with its shortest solution.

3.3.1.2. Two different paths, same length, same state

Inspired by the first observation, the number of nodes that need to be traversed can be decreased even further by realizing that, no new entries will be populated from a search of a state, that was already reached within the same depth bound. Take for example the path p = U2 D2 R2 L2 and p' = R2 L2 U2 D2. When p gets traversed first in the iteration with the depth bound 5, the entry for the state that is reached with U2 D2 R2 L2 F will be set in the lookup table. But the same state will later be reached with R2 L2 U2 D2 F and no new information is generated. This and all other subsequent states that follow from a state that was reached before in an iteration will be traversed for nothing. So pruning the search at these states would decrease nodes that need to be traversed even further. Unfortunately there is no way to detect whether a node has been visited before or not without storing which states have been traversed already. When searching for a solution of a scrambled cube, it is not reasonable to store all traversed states, since the number of possible configurations and is just too big, but in the algorithm shown there is already an array which holds information about every state that is reachable for the subset of pieces of interest. Furthermore is it known, that no configuration (even if the configuration of the whole cube is used) needs more than 20 moves to be solved. This means that the entries in the lookup table only need 5 bits to store the distance of a state. If the entries are stored in a byte, there are three bits left to store information about each state. So it is possible to mark all reached states by setting a bit flag for states that have been reached in an iteration and stop any further traversing of the graph when a state is reached that is marked. Note that in between every iteration all flags have to be reset to avoid that the search is stopped immediately, but the overhead is completely compensated with the improved generation performance.

3.3.1.3. Inverse Lexicographic Rank

The last presented optimization of the lookup table generation is most relevant for the bigger lookup tables that will be generated. For the last couple iterations a lot of entries of the lookup table are already populated and many entries for states at the depth bound already set. This motivates the approach to dismiss the IDDFS from a certain depth bound and rather set the unpopulated states directly. This can be done by checking if the state that corresponds with an unset entry has any adjacent states, whose corresponding entry has already been set and if so setting the entry of the original state to the value of an adjacent state's entry plus one. If the search finished the iterations with a depth bound of 8 for example, all entries that haven't been set yet must have a distance of 9 or more. If a state that corresponds with any unset entry that has distance 9, for one of the 18 moves, applying it must result in a state which entry was set to 8. So the array can be scanned linearly and for all unset entries the state can be reconstructed and checked if an adjacent state is set already. Therefore this method requires it to be possible to reconstruct a state from its index.

To reconstruct the state, the first step is to extract to two coordinates for the permutation and the orientation. It is not hard to see that the index calculated with equation 3.4 can be decomposed back into the original coordinates with

$$C_{perm} = \lfloor I_c / N_{co} \rfloor \qquad \textit{and} \qquad C_{orientation} = I_c \mod N_{co} \tag{3.5}$$

where again N_{co} is the number of possible corner orientation arrangements. Now from the two coordinates the 7 orientations of the corners (the sum of the last orientation together with all other orientations must be divisible by 3 and can therefore be computed with this law) as well as the permutation of the 8 corners must be reconstructed. The orientations can be constructed by converting $C_{orientation}$ to a base-3 number and interpreting each digit as the orientation of the corresponding corner. Therefore an equation for the orientation $C_{orientation,i}$ of the i-th corner looks like

$$C_{orientation,i} = (C_{orientation}/3^i) \mod 3 \tag{3.6}$$

The edge orientations can be reconstructed in the same manner by converting $E_{orientation}$ to a base-2 number. The i-th orientation of the egdes $E_{orientation,i}$ can be computed with

$$E_{orientation,i} = (E_{orientation}/2^i) \mod 2 \tag{3.7}$$

Now the permuation of corners and edges need to be extracted as well. First the coefficients $L(\sigma)_i$ of the Lehmer code needs to be computed from the lexicographic rank of the corners with

$$L(\sigma)_i = \left\lfloor \frac{(C_{perm} \mod (7-i+1)!)}{(7-i)!} \right\rfloor \tag{3.8}$$

To reconstruct a permutation from its Lehmer code each σ_i can be obtained successively with $\sigma_i = \Sigma(\omega, L(\sigma)_i)$, where ω is a bitmap with the k-th bit set when σ_i was set to k in an earlier iteration and $\Sigma(b, n)$ is a function that returns the index of the n-th zero in a bitmap b. We elaborate the process a little further. The first element of the permutation is always equal to the first Lehmer code digit, since there are no elements to its left. The bit at $L(\sigma)_0$ is set after σ_0 is set. The following σ_i become the position of the $L(\sigma)_i$th zero of the bitmap.

In the example shown in Section 3.1.3 the coefficients were 0, 1, 4, 1, 1, 0, 0 ,0. So the first edge index will 0. The first bit in the bitmap is set ($L(\sigma)_1 = 1$). The first zero of the bitmap is at position 1. So σ_1 becomes 1 and the bit at index 1 is set. σ_2 now becomes the forth zero in the bitmap which now is at index 6. Continuing this process σ can completely be reconstructed.

As with computing the lexicographic rank for a permutation the inversion of this process, has a time complexity of $O(n^2)$ when a naive implementation is used since for the n elements of the permutation the position of the k-th zero needs to be computed. Both computations scale linear in the operations needed with respect to the number of elements of the permutation.

We derived the presented algorithm together with the following optimization, which we later identified to be a rediscovery of an algorithm that was first proposed in "Efficient Algorithms to Rank and Unrank Permutations in Lexicographic Order" [1], which also introduced a method improving the asymptotic run time to be linear. In the work they precompute a lookup table that contains entries for all possible bitmaps and every index of zero. This table has $2^n \cdot n$ entries, where n is the number of elements of the permutations that will be reconstructed. By that they can find the k-th zero in $O(1)$.

Our tests have shown that using the first two introduced optimization methods up to a depth of 8 and then populating the rest of the table with an inverse lexicographic rank search[2] yields the best results. The inverse lexicographic rank search performs best when few entries of the table are uninitialized, whereas the IDDFS is efficient for generating entries up to a small depth bound.

If the graph can be traversed only with coordinates as described in Section 3.1.3 no reconstruction of the permutations is required. But since we opted to use a cube represented which does not purely rely on coordinates for the lookup table generation the inversion is still required.

3.3.1.4. Space Optimization

The maximum move count stored in all pattern databases discussed in this work is 14. So every entry in the databases could be stored in half a byte, but this can be improved even further by realising that adjacent states can only differ by one move in their distance from the solved state. Therefore the database can be confined to only store the real distance mod 3. The actual distance for a state then can be computed by taking the difference between the current state's stored value and the adjacent state's value and adding it to the heuristic value of the adjacent state. [12]

For pruning values in any search the absolute heuristic value is needed rather than a relative one. But the absolute heuristic value of an arbitrary state s can be computed beforehand: Start a search from s and only traverse nodes that decrease the heuristic value, until the solved state is found. The distance of s to the solved state is the absolute heuristic value. This needs to be done for every lookup table used. Since every node in the pattern databases has a neighbour that is closer to the solved state, this search only needs to generate $O(b * d)$ nodes. Here b is the branching factor and d is the maximum distance any entry of the database has.

With this method only 0, 1 and 2 are valid values for an entry. But instead of storing 4 entries in one byte (where each two bits represent one entry) 5 entries can be stored in one byte when they are compressed because the number of configurations of five entries does not exceed the number representable by one byte ($3^5 = 243 < 256 = 2^8$). But if the

[2]The edge group and the edge permutation pattern database generation used the IDDFS search only to a depth of 7.

first byte stores the entries (0,1,2,3,4) and the second byte (5,6,7,8,9), and so on, then the index of the byte has to be computed with div-5 and mod-5 operations. Storing the entries like (0,1,2,3,x), (4,5,6,7x+1), ... where x is about $\frac{4}{5}$ of the size of the whole database, allow using div-4 and mod-4 operations, which are faster. [4, 12]

Since a fourth value (uninitialized) is required when generating pattern databases, one byte can only store four values during the generation and only if all entries are set the database can be compressed. To use a flag to indicate whether a state has been reached in one iteration of the IDDFS-based generation, even more than two bytes need to used for one entry. [4, 12]

This compression is a trade off of time for less space, since the lookup requires more operations. For one lookup an entry has to be decompressed (this can also be done with small lookup table). That is why we opted to use a full byte for one entry during the solve of a scramble. But the pattern databases can be stored compressed and inflated only when required. This adds some extra startup time of the program.

In our program only pattern databases that store absolute heuristic values are used. In Chapter 5 a discussion on how the compression affects performance is presented.

3.3.2. Corners

One of the pattern databases that was used by Korf in his original work contained the distances for all corner arrangements. With only $8! * 7^3 = 88\,179\,840$ entries in the table, the database takes up only about 88 MB. The furthest configuration of corners from the solved state has distance 11 so 4 bits are enough to store the distances and one byte can hold two entries. With that all entries can be stored in about 44MB.

The index I_c mapping the configuration of the corners to a number can be calculated with

$$I_c = C_{permutation} * N_{co} + C_{orientation} \tag{3.9}$$

where $N_{eo} = 2^{11}$ is the number of possible edge orientation arrangements reachable with with regular moves. This value can be used for indexing the pattern database. The state distribution with respect to the depth can be found in Figure 3.2. The expected heuristic value of the corner pattern database is 8.764.

depth	configurations
0	1
1	18
2	243
3	2 874
4	28 000
5	205 416
6	1 168 516
7	5 402 628
8	20 776 176
9	45 391 616
10	15 139 616
11	64 736

Table 3.2.: Corner configurations distribution per depth

With all optimization methods the generation of the corner pattern database takes less than 60 seconds using 6 threads for the IDDFS and 8 threads for the inverse lexicographic rank search.

3.3.3. Edge Groups

Using only information of the corners is not enough to obtain a sufficiently precise estimates of the distance to the solved state, therefore Korf also used two pattern databases that stored the distance for the configurations of the six edges [14].

When considering the positions of 6 of the 12 edges, there are $\frac{12!}{6!} = 665\,280$ possible permutations. Each edge can be in one of two orientation, so $2^6 = 64$ different orientation arrangements are possible. In total there are $\frac{12!}{6!} \cdot 2^6 = 42\,577\,920$ different configurations for six edges. To compute an index of the configuration of this subset of pieces an extension of the method introduced in Section 3.1.3 needs to be used. To calculate the Lehmer code of a partial permutation the same method can be used, but for converting the Lehmer code to a decimal number the coefficients have to be multiplied by a different base. But the permutation of a subset of pieces has to be constructed in a slightly different way. Instead of the index of an edge at position p becoming the p-th element of the permutation, the position of the piece with index i becomes the i-th element of the permutation. We provide an example.

p	0	1	2	3	4	5	6	7	8	9	10	11
cube.e[p].i	3	2	1	6	4	9	10	11	8	0	5	7

cube.e[p].i is the index of the edge at position p. When considering the edge group consisting of indices 0 to 5 the permutation would be (9, 2, 1, 0, 4, 10). For indices 6 to 11 the permutation would be (3, 11, 8, 5, 6, 7). With that permutations of any subset of pieces can be constructed that can be used to compute their Lehmer code.

Let σ_e be the permutation of one edge group and $E_{g,perm}$ be the lexicographic rank of one of the edge groups then with

$$E_{g,perm} = \sum_{i=0}^{6} L(\sigma)_i \cdot pick(N-1-i, K-1-i)$$
$$\textbf{where } pick(n,k) = \frac{n!}{(n-k)!} \tag{3.10}$$

the permutation coordinate of one edge group can be computed [3]. N = 12 is the number of edges and K = 6 is the number of edges in the edge group. To compute the lexicographic rank of the configuration (position and orientation), $I_e = E_{g,perm} * N_{co} * E_{g,orientation}$ can used. $N_{co} = 2^6$ is number of configurations for the orientation of the six edges and $E_{g,orientation}$ is the orientation coordinate of the six edges. Latter can be computed similarly as described in Section 3.1.3 by just omitting the orientations of unused edges when constructing the binary number.

Korf used edge groups with six elements since each database requires only about 21 MB when compressed [15]. Adding one more edge increases the number of configurations to $\frac{12!}{5!} * 2^7 = 510\,935\,040$ which would require about 255 MB when compressed. This seemed to have been too much of a memory requirement for that time. Ben Botto used two pattern databases storing the distances for configurations of 7 edges [2]. So did we in our implementation.

While the pattern database Korf used only contained states up to a depth of 10, the database used by Botto and in our implementation contains states up to a depth of 11. Korf's database yields an expected heuristic value of about 7.668 moves, while the larger database improves this to about 8.507 moves. The state distribution with respect to the depth can be found in Table 3.3. The generation of the pattern database took about 230 seconds.

depth	configurations
0	1
1	15
2	191
3	2 455
4	30 519
5	356 462
6	3 766 700
7	32 719 467
8	186 297 009
9	274 719 633
10	13 042 507
11	81

Table 3.3.: Edge group configurations distribution per depth

3.3.4. Edge Permutation

One pattern database used in out program that was used by Botto but not by Korf contained the permutation of all edges, ignoring their orientations. This database has $12! = 479\,001\,600$ entries and needs about 240MB of memory when compressed. For indexing the database E_{perm} can be used which can be computed as described in Section 3.1.3. The state distribution with respect to the depth can be found in Table 3.4.

depth	configurations
0	1
1	18
2	243
3	3 240
4	42 535
5	542 234
6	6 529 891
7	66 478 628
8	310 957 078
9	94 443 600
10	4 132

Table 3.4.: Edge permutation distribution per depth

The expected heuritisc value of this database is about 8.027 moves. The generation of the database took about 170 seconds.

3.4. State of the Art: Iterative Deepening A*

With a near to linear space complexity graph search algorithm and an heuristic for the remaining distance to the solved state, a first approach to solve any scrambles in reasonable time can be made. This chapter introduces the algorithm that we used to solve random instances of a cube.

3.4.1. The Algorithm

Richard Korf contributed to the solving of many combinatorial puzzles like the 15 sliding tiles puzzle and the Rubik's cube. In 1985 he published an article titled "Depth-first iterative-deepening: An optimal admissible tree search", where he first describes an algorithm, today known as *iterative deepening A**(IDA*) [13].

Let $f(n) = g(n) + h(h)$ be a cost function, where g(n) is the sum of the edge costs from the initial node s to a node n and h(n) a heuristic function that estimates the distance from n to the goal node. Let τ be the cost threshold of a IDDFS iteration and d(s) the shortest distance from s to the goal node. IDA* works like IDDFS, but it prunes paths from any node n that satisfies $f(n) > \tau$. The cost threshold in the first iteration is set to the cost value of the initial state ($f(s) = g(s) + h(s) = 0 + h(s) = h(s)$). The cost threshold τ of the next iteration will be set to the lowest cost of all pruned nodes. It continues until a goal is found, that not exceeds the cost threshold. Algorithm 3 shows an implementation of the algorithm.

Algorithm 3 Iterative Deepening A*

```
function ITERATIVE DEEPENING A*(cube)
    $moveStack \leftarrow []$
    $bound \leftarrow GetHeuristic(cube)$
    while IDA_Iteration(cube, 0, bound) == False do ▷ As long as no solution is found
        $bound = nextBound$
        $nextBound = MAX\_VALUE$              ▷ Let this be accessible for IDA_Iteration
    end while
    return moveStack
end function

function IDA_ITERATION(cube, depth, bound)
    $estimatedMoves \leftarrow GetHeuristic(cube) + depth$
    if estimatedMoves > bound then
        return False;
    else
        if estimatedMoves < nextBound then
            $nextBound \leftarrow estimatedMoves$
        end if
    end if
    if IsSolved(cube) then
        return True;
    end if
    for move={U , U’, U2, ..., D, D’, D2} do              ▷ Loop trough all possible moves
        $ApplyMove(cube, move)$
        moveStack.push(move)
        if IDA*(cube, bound, depth + 1) == True then
            return True;
        end if
        moveStack.pop()
        $ApplyMove(cube, Inverse(move))$
    end for
    return False
end function
```

Lemma 1. *The algorithm will find shortest paths if h(n) is admissible. A heuristic function is admissible* $:\Leftrightarrow \forall n \in N : h(n) \leq d(n)$ *where N is the set of all nodes and d(n) is the smallest distance of n to the goal node. In other words: IDA* is optimal if and only if the heuristic function never overestimates the remaining distance to the goal node.*

Proof. In the first iteration $\tau = f(s) = h(s) \leq d(s)$ and since only paths $\{s, n_1, \ldots n_k\}$ with $f(n_i) \leq \tau$ will be traversed, a solution is only contained if $\tau \geq d(s)$. In the next iteration τ is set to the smallest cost value found so $\tau \leq d(s)$ stays true. This is repeated until $\tau \geq d(s)$ and a solution is found. With $\tau \leq d(s)$ in every iteration and $\tau \geq d(s)$ only if a solution with length τ is found, τ must be d(s) and the solution therefore optimal. [17] □

Since the heuristic function used is a pattern database that stores only distances for subsets of pieces the heuristic values never exceed the real distance to the solved state and therefore only optimal solutions will be found. To use multiple pattern databases to estimate the distance to the solved state, the only way to combine the information without the loss of admissibility is to take the maximum of all databases [14]. To further improve performance Botto's and our implementation sort the neighbours of a node that gets traversed in best-first search manner to improve the number of traversed nodes in the last iteration. Our tests have shown that sorting the adjacent nodes decreases the number of nodes that can be generated per second by approximately 10 percent but improved the run time overall.

3.4.2. Experimental Results

Korf reported that he solved ten random configurations of the Rubik's Cube and generated between 3 720 885 493 and 1 021 814 815 051 nodes during the solve. Of the ten scrambles one had a solution length of 16, three had a solution length of 17 and six had a solution length of 18. He could traverse 700 000 nodes per second [14]. With the new lookup tables Botto introduced, we solved 16 random scrambles and generated between 274 289 501 and 39 526 346 646 - about 9 177 813 092 on average. We also had solution lengths between 16 and 18. Exact numbers are presented in Chapter 6.

Botto claims to have to have the fastest implementation of Korf's algorithm. To the best of our knowledge this is true. We used his program on the same hardware all other tests also ran on and found that our implementation is approximately as fast as his when running single threaded. All results shown in this work used multi threading. More details on how the parallelization works and how it scales with the thread count is shown in Chapter 5.

In his work "Finding Optimal Solutions to Rubik's Cube Using Pattern Databases", Korf stated that the number of nodes that need to be generated when solving a cube has an inverse linear relationship with the size of the pattern databases used for pruning.

3.4.3. Performance Analysis

Let n be the number of states in the entire problem space, Let b be the brute-force branching factor, let d be the average optimal solution length for a random problem instance, let e be the expected value of the heuristic, let m be the amount of memory used, in terms of heuristic values stored, and let t be the running time of IDA*, in terms of nodes generated. Korf assumes that the average depth at which IDA* finds a solution can be estimated with $d \approx log_b n$, e can be estimated with $e \approx log_b m$ and t can be estimated with $t \approx b^{d-e}$. With all that he derives

$$t \approx b^{d-e} \approx b^{log_b n - log_b m} = \frac{n}{m} \tag{3.11}$$

So the running time may be approximated with $O(\frac{n}{m})$ [14].

The number of states in problem space of the Rubik's Cube is about $n \approx 4.3 \cdot 10^{19}$ as shown in Section 2.3. The size of the pattern databases used by Korf is $m = 2 \cdot (\frac{12!}{6!} \cdot 2^6 + 8! \cdot 3^7) = 173\,335\,680$. So $t = \frac{n}{m} = 352\,656\,042\,894$, which is only off by a factor of 1.4 from the average of nodes generated by our 16 solves.

We used 4 pattern databases that have a combined size of $m' = 2 \cdot (\frac{12!}{5!} \cdot 2^7 + 8! \cdot 3^7 + 12!) = 1589051520$. With $t' = \frac{n}{m'} \approx 27\,060\,167\,312$, we are only off by factor of about 2.95 from the real average of nodes generated in our 16 solves.

4. Contributions

In this chapter we present our Contributions. First we explain how we generated the full edge pattern database and discuss the findings. After that we introduce two techniques for detecting states, that are have been visited before in an iteration of IDA* without the need to store all states.

4.1. Full Edge State Database

One limitation of the pattern databases is that they need to be small enough to fit into the RAM , since the latency of hard drives is so slow, that it becomes unreasonable to do a lookup for every node. This is why a pattern database containing all possible configurations of the edges has not been generated before. To the best of our knowledge, we are the first to generate the whole database for the all edge permutations. As shown in Section 3.1.3 the edges can have 12! permutations and 2^{11} orientation arrangements. So the indices of this database can be in the range from 0 to $12! \cdot 2^{11} = 980\,995\,276\,800$. With the coordinate values $E_{permutation}$ and $E_{orientation}$, which can be computer as described in Section 3.1.3, the edge index I_e can be computed with

$$I_e = E_{permutation} * N_{eo} + E_{orientation} \tag{4.1}$$

where $N_{N_e o} = 2^{11} = 2048$ is number the number of reachable edge orientation configurations.

This pattern database requires about 1 Terra byte of memory when uncompressed. To use it a machine with such an amount of memory is needed to prevent the long latency of hard drives.

At the Karlsruhe Institute of Technology we used a machine with 1 Terra byte of RAM to generate and use the database. In an early iteration of this work, we tried to generate the database without the use of the inverse lexicographic rank optimization (see Section 3.3.1.3) and found, that after two weeks generation wasn't finished. After implementing the optimization we were able to generate the lookup table in less than three days. We used 18 threads for the IDDFS step of the generation and 128 threads for the inverse lexicographic rank step of the search. The lookup table contains the distances for states up to a depth of 14. The distribution of states with respect to their depth can be found in Table 4.1. The expected heuristic value of this database is about 11.17.

Loading the uncompressed lookup table into the RAM takes about 90 minutes, but afterwards arbitrarily many shuffles can be solved. The three pattern databases described in Section 3.3 that used a subset of the edges or the permutation of all edges were not used, since they never contain greater heuristic values than the full edge database. We solved 5000 randomly generated shuffles with solution lengths that reached from 15 to 19 moves.

depth	configurations
0	1
1	18
2	243
3	3240
4	42807
5	555866
6	7070103
7	87801812
8	1050559626
9	11588911021
10	110409721989
11	552734197682
12	304786076626
13	330335518
14	248

Table 4.1.: Edge group configurations distribution per depth

None of the solves took more than 900s and the maximum number of nodes generated is 1 489 814 309. On average we needed about 78 seconds and 149 881 988 nodes to solve the scrambles.

With a pattern database size of $m = 12! * 2^{11} + 8! * 3^7 = 981\,083\,456\,640$, we can use Korf's Equation 3.11 to get an estimated number of nodes of about 43 829 094. This number is off by a factor of about 3.42. This shows, that Korf's asymptotic approximation for the run time of IDA* still holds true for pattern databases that are nearly 3 orders of magnitude greater in size than the ones he used.

4.2. Duplicate State Detection

One deficiency of IDA* is that some nodes will be traversed multiple times. For the graph constructed by applying moves to the Rubik's Cube, this problem gains even more relevance because a lot of states can be reached via different non trivial (paths that only differ by swapped commutative moves) paths along the graph. The simplest way to detect nodes that are reached multiple times is to store every traversed node together with the depth at which it was reached in hash table and check for every node if it was reached with a more shallow depth and if so prune it. But just like with the searching algorithm the space of the problem is too great to store all traversed nodes. Table 3.1 shows an upper bound of entries the hash table would have for searches up to a certain depth.

One easy way to solve the problem is to store and check only nodes up to a certain depth. On household hardware (with about 8 GB of RAM) our tests have shown that up to a depth of 6 storing all states lies in the capabilities of the memory, but the overhead of hash table operations resulted in too few nodes generated per second to improve performance.

Since populating the hash table during run time is too slow, some sort of table, generated ahead of time, is required that can be accessed quick for looking up whether a state is a duplicate or not. We present two types of lookup tables that improve the performance of the search.

4.2.1. Relative States

Before explaining how indexing a state works, we have to address another problem: We can compute which sequence of moves or which states, relative to some starting configuration lead to a duplicate state, but the main search starts from some arbitrary state and the entries in the lookup table will have absolute indices - meaning that if some state that was shown to be a duplicate state when starting from a solved state, it is not necessarily a duplicate state when starting from some arbitrary configuration. To resolve this problem we either have to rely on a relative indexing, like using the all moves applied or take some state indexing method, that is not absolute. We first show why using an index computed from all moves applied resolves this problem.

Let p be the sequence of moves $(p_1, \dots, p_n)$ and let s be the solved state. Further lets define $p \cdot s = g$ to be the operation, that applies p_1 followed by the successive p_i's to s to reach the state g. Now assume a p' with $p' \cdot s = p \cdot s = g$ with $p \neq p'$. Applying p and p' both transform s to g, or in others words applying the moves of p leads to the same state as applying the moves of p' does. In this case we would call g a duplicate state, since there are at least two sequence of moves that lead to g. Note that without restricting the length of the length of p and p' every state is duplicate state[1] So we're just interested in duplicates states with a length bound for p and p'.

We know that p and p' transform s into the same state g, but we need that p and p' transform any state s' into the same state g'. We can proof this property with the following thought experiment. Take a solved Rubik's Cube and put a sticker on every facelet to construct a any arbitrary state. If you apply p and then take off the stickers you will find the same state, that you would have reached if you would have applied p' in the same process. Since the state of the cube with the added stickers have to the same for finding the same state after removing the sticker with p and p', we have shown that every for every starting configuration p and p' yield the same state after applying them.

4.2.2. Hashing a Rubik's Cube State

We can compute the coordinate $(C_{orientation}, C_{perm}, E_{orientation}, E_{perm})$ for a given state s as described in Section 3.1.3 and compose the corner and edge coordinates I_c and I_e as described in Sections 3.3.2 and 4.1. In our program these two number as a tuple define the index of a state of the Rubik's Cube. For using this index in a hash table it has to be

[1] A sketch of an informal proof could look like: For every reachable state g there is a shortest path p that transforms the solved state s to g. Another shortest path p' with $p' \neq p$ can be constructed with some random state g' with $(s \neq g' \neq g)$. By taking the shortest paths $p_{sg'}$ and $p_{g'g}$ with $p_{sg'} \cdot s = g'$ and $p_{g'g} \cdot g' = g$ where $p_{sg'}$ and $p_{g'g}$ are no subset of p. Concatenating $p_{g'g}$ and $p_{sg'}$ yields a path p', With $p' \cdot s = p_{g'g} \cdot (p_{sg'} \cdot s) = p_{g'g} \cdot g' = g$. Since $p_{sg'}$ and $p_{g'g}$ are no subsets p, we found a p' with $p' \neq p$ that also transforms s to g.

comparable and mappable to an 64-bit integer. The number of states these two number can be in is $12! \cdot 2^{11} \cdot 8! \cdot 3^7 \approx 8.65 \cdot 10^{19}$ which is twice as big as the number of reachable states, since configuration of corner and edge permutations that are not reachable with regular moves are contained in this number (see Section 2.3). With $log_2(8.65 \cdot 10^{19}) \approx 66$, we see that the number cannot not be stored in 64-bit variable. But since I_c can be stored in 27 bit ($log_2(8! \cdot 3^7) < 27$) and I_e can be stored in 40 bits ($log_2(12! \cdot 2^{11}) < 40$), a simple hash function can be used. In our program we use H(I) = $I_e \oplus (I_c$ « (64 - 27)) as the hash function, where ⊕ is the XOR-operator and « is the left bit shift operator. With this only 27 + 40 - 64 = 3 bits are overlapping.

4.2.3. Sequence Index Calculation

After applying maximally n moves there are $\sum_{i=1}^{n} 18^{i-1}$ possible sequences of moves. Let $M = \{m_0, \ldots, m_n\}$ be a sequence of moves with length n where m_i is a number between 0 and 17, representing one of the 18 moves. To calculate a sequence index I_s based on the moves applied we can use the following formula

$$I_s = \sum_{i=0}^{n} (m_i + 1) * 18^{n-i-1} \tag{4.2}$$

With this index a lookup table can be generated that contains all paths up to length of n, where each entry is a flag indicating whether a search from the corresponding node needs to be continued or pruned. Since this flag can be stored in one bit a byte can hold 8 entries. For our program we used lookup tables with move sequences of length 7 and 8. The smaller one has $\sum_{i=0}^{7} 18^i = 648\,232\,975$ entries and requires about 81MB when compressed. The larger one has $\sum_{i=0}^{8} 18^i = 11\,668\,193\,551$ entries and requires about 1.46 GB when compressed.

During the solve of a scramble the current path can be used to compute the sequence index. This index can then be used to retrieve the information about whether the path was reached or will be reached reached or not[2]. The search is then continued or pruned accordingly.

4.2.4. Shifted Configuration Index

The sequence-index-based approach saves for every sequence of moves whether a state will be reached by at least one other path or not. This approach has the disadvantage, that a lot of paths do not reach duplicate states and are therefore stored for nothing. Furthermore all paths are stored that reach a duplicate state. Optimally only the duplicate states represented by their index are stored. A table with only these state indices can be computed ahead of time and while finding an optimal solution, for every node it can be checked if it is contained in that lookup table and pruned if it was set to visited, or set to have been visited otherwise. With this approach just a fraction of memory would be required to detect duplicate states.

[2]Which sequence to a duplicate state is found first depends on the order the state space is searched in while solving, during the lookup table generation or how the parallelization traverses the tree.

Assume a lookup table was generated that contains all state indices of states that can be reached with two different paths (with less than 8 moves) from the solved state. This does not help finding duplicate states when starting from an arbitrary configuration, unless some sort of index shift maps the indices in the table to indices of states that are reachable with the same two same paths from some arbitrary state. Formally we require a state shift function s(g, c) that for some state g and some starting configuration c returns the state that would have been reached if the transformation that transformed c to g is applied to the solved state[3]. We provide an example that clarifies this property. When we apply the moves p = R2 L2 U2 L2 to the solved state s, we reach a state g with the state index I = (C_c = 88177653, C_e = 845111005184). Applying the moves p' = U2 D2 R2 L2 yields the same state with the same state index. For an arbitrary configuration c that we want to solve both move sequences p and p' lead to the same state g', but g ≠ g', unless $c = s$. So the state index I' of g' will be different to I. To detect that p and p' lead to the same state for every starting configuration, the shift function should return s(g', c) = s($p \cdot c$, c) = s($p' \cdot c$, c) = $p \cdot s = g$. Since we know that the state index I that corresponds with g is stored in the duplicate state lookup table, we know that only for one of p and p' the search needs to be continued.

For two arbitrary states c and g we want to compute the transformation that applied to c leads to g and apply it to the solved state. Assume a single corner. For any arbitrary configuration c we can find the position p of the corner and can take the orientation o of the corner. After applying a sequence of moves to this configuration we reach g. The corner will be at position p' and have orientation o'. For the solved cube we can look know which piece is at position p and what orientation it has. After applying the moves that transformed c to g to the solved state, this corner must be at p'. The transformation changed the orientation o to o' so the difference $\Delta o = o' - o \mod 3$ ($\mod 3$ prevents negative orientations). Since all pieces on a solved cube have orientation 0, the corner that lied at position p on the solved cube will have the orientation Δo after the transformation is applied.

We provide an example with Figure 4.1. The following steps describe what happens in the figure:

1. Apply the scramble to reach c and save the positions and orientations of the pieces.
2. 2. In the search for a solution for this scramble we apply some move sequence, reach g and see compare the new positions and orientations of the pieces with the old ones.
3. 3. We apply the changes to the solved state to compute the state index that we can use to check if we reached a duplicate state.

The scramble move sequence S = {F}, which contains only one move for simplicity. The UFL corner (corner index 2) will be at the UFR position (index 1) and have orientation 1 after S is applied to the solved state, so we store the old position of the UFL corner as 1 (oldPos[2] = 1) and its orientation (oldOri[2] = 1). The resulting cube can be seen on the

[3]An easy way to achieve this would be to just apply the moves that reached the current state in the search to the solved state, but this approach's number of operation scales linear to the depth. This is not as fast as the method presented here.

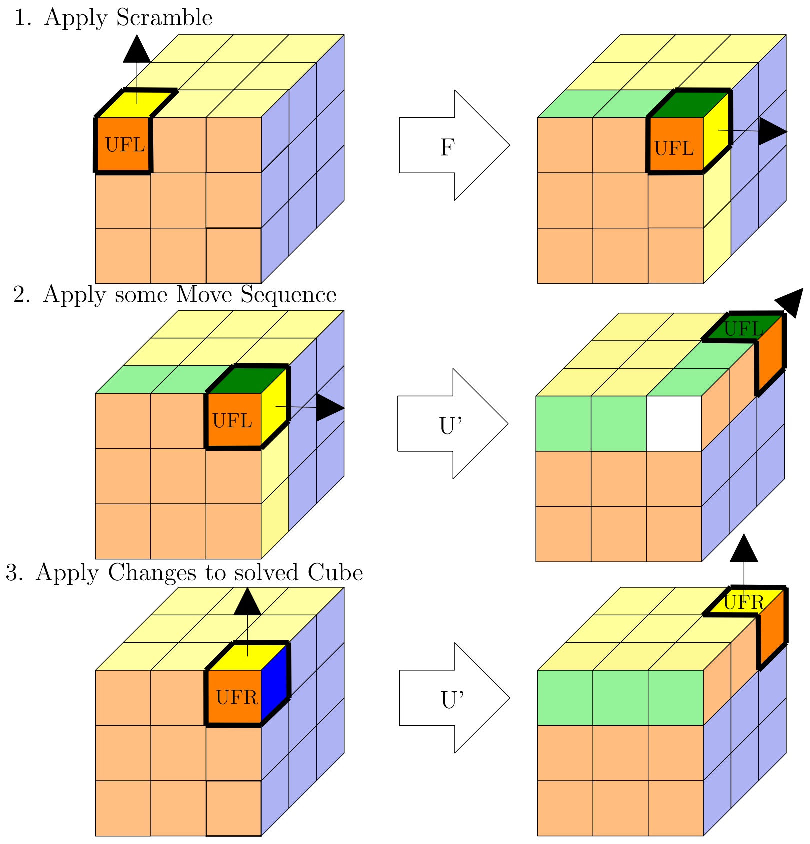

Figure 4.1.: Process of State Shifting

right of step 1 in the illustration. When we apply another move sequence P = {U'}, also containing only a single move, to the already scrambled state, the corner moves to position UBR and has orientation 1. So we know that the UFL corner now is at the UFB position (newPos[0] = 2) and its orientation has not changed (newOri[0] = 1). We can compute the the index and orientation for a corner at position i to reconstruct the state that would have been reached if P would have been applied to the solved state. To compute which corner is at position UFB (index 0), we can look at newPos[0] = 2 and find the position the corner with index 2 came from with OldPos[2] = 1. So we know that the corner at this position must have been at position 1 before p was applied. On position x of the solved cube always lies the piece with index x, so finally we can conclude that at position UBR (index 0) will be the UFR corner (corner index 1). To compute the orientation we do the same with the saved information about the orientation. We check which orientation the piece at position 0 had before p was applied (step 2, left) and find that it had orientation 1 (oldOri[newPos[0]] = 1). We compare it with the orientation it had after p was applied with newOri[0] = 1. The difference $\Delta o = 1 - 1 = 0$ is the orientation of the corner (we would need to take the orientation of the corner of the solved state into account, but per definition all corners of the solved state have orientation 0). With that we found that the configuration of one corner. We can do the same for all other corners and the edges to reconstruct the whole cube. How all of this put together looks as an algorithm is shown in the next Section.

In total, with a given scrambled state c and another state g that is reached in the search after applying a move sequence P. We can now compute which state arises when the move sequence is applied to the solved state and check if it is a duplicate state with a lookup hash table. This can be used to check if a node represents a duplicate state:

1. Do a search to depth n from the solved state and store all state indices in a hash table. For every state found that already has an entry in the hash table, save it to a separate lookup hash table.

2. When arriving at some node g that was reached with the path p from the scrambled state while solving a random scramble, take g' = s(g, c). g' is the state that would have been reached if p was applied to the solved state (note that this all works without knowing the moves of p).

3. Compute the state index I(g') that corresponds with g' and check if the lookup hash table contains g'.

4. When the state index is contained set the value of the entry in the hash table to true.

5. When another node with the same state index is found the hash table tells us that the entry has already been set, a duplicate state is found that already has been traversed and the node can be pruned.

Computing all states indices that can be reached with two different move sequences (with a length limit) from the solved state only has to be done once. The resulting lookup hash table can be stored in a file. Only the keys need to be stored. They can be used to populate a hash table on the startup of the program.

4.2.4.1. The State Shift Algorithm

Algorithm 4 shows how an implementation of the state shift. The algorithm first extract the positions and orientations for all pieces of the scrambled state. This step has to be done only once for solving one scramble. It then uses these information to reconstruct the state that would have been reached if the transformation that transformed *scramble* (the state that was passed to *ComputeShiftedState*) to *cube* (the state that is passed to *GetStateShift*) is applied to the solved state. This state can then be used to compute its state index for indexing the lookup hash table.

Algorithm 4 ComputeShiftedState

```
ciMap ← array[8]                                              ▷ corner indices mapping
coMap ← array[8]                                              ▷ corner rotation offset
eiMap ← array[12                                              ▷ edge indices mapping
eoMap ← array[12]                                             ▷ edge rotation offset
function GetStateShift(cube)    ▷ For one scramble this needs to be computed once
    for p = 0 to 7 do
        $ciMap[cube.c[p].i] \leftarrow i$
        $coMap[cube.c[p].i] \leftarrow (3 - cube.c[p].o)\%3$
    end for
    for p = 0 to 11 do
        $eiMap[cube.e[p].i] \leftarrow i$
        $eoMap[cube.e[p].i] \leftarrow cube.e[p].o$
    end for
end function

function ConstructState(cube)
    $rCube \leftarrow Cube()$                                 ▷ reconstructed cube
    $orientationSum \leftarrow 0$
    for p = 0 to 7 do
        $rCube.c[p].i = ciMap[cube.c[p].i]$
        if i < 7 then
            $rCube.c[p].o \leftarrow (cube.c[p].o + coMap[cube.c[p].i])\%3)$
            $orientationSum \leftarrow (orientationSum + rCube.c[p].o)\%3$
        end if
    end for
    $rCube.c[7].o \leftarrow (3 - orientationSum)\%3$
    $orientationSum \leftarrow 0$
    for p = 0 to 11 do
        $rCube.e[p].i = eiMap[cube.e[p].i]$
        if i < 11 then
            $rCube.e[p].o \leftarrow (cube.e[p].o + eoMap[cube.e[p].i])\%2)$
            $orientationSum \leftarrow orientationSum + rCube.e[p].o$
        end if
    end for
    $rCube.e[11].o \leftarrow orientationSum\%2$
    return rCube
end function
```

4.2.5. Lookup Table Generation

As with the pattern databases the lookup tables have to be computed beforehand to be used for solving random scrambles. Algorithm 5 shows an implementation of for the generation. We use a IDDFS to find the states ordered by depth. We store all states found and mark a state as duplicate when it is found again. When we found a duplicate we can stop traversing further cause all subsequent states will have been found before (See 3.3.1.2). After the generation is finished the list of duplicate states and the array of move sequences can be stored in a file and loaded on the startup of the program. The duplicate sequence array can be used for indexing right away, but the every element of the duplicate states list has to be put into a hash table before it can be used in the search.

The optimization presented in Section can also be applied here to decrease the time needed for the generation.

Algorithm 5 GenerateDuplicateStateTable

```
function GenerateDuplicateStateTables(maxDepth)
    $moveStack \leftarrow$ new Stack
    $reachedStates \leftarrow$ new HashTable
    $duplciateStates \leftarrow$ new List
    $duplicateSequences \leftarrow Array[GetSequenceSize(maxDepth)]$
    for depthBound = 0 to maxDepth do
        IDDFS(solvedCube, 0, depthBound);
    end for
end function

function IDDFS(cube, depth, bound)
    $stateIndex \leftarrow ComputeStateIndex(cube)$
    if reachedStates.Contains(stateIndex) then
        duplicateStates.Add(stateIndex);
        $duplicateSequence[GetSequenceIndex(moveStack)] \leftarrow true$
        return
    else
        reachedStates.Add(stateIndex)
    end if
    for move={U , U', U2, ..., D, D', D2} do            ▷ Loop trough all possible moves
        $ApplyMove(cube, move)$
        moveStack.push(move)
        IDDFS(cube, bound, depth + 1)
        moveStack.pop()
        $ApplyMove(cube, Inverse(move))$
    end for
end function
```

The function *GetSequenceSize* takes a length and returns the number of different move sequences that can be construct up to that length. *GetSequenceIndex* takes a move sequence and returns its sequence index. *ComputeStateIndex* takes a configuration of the cube and returns the corresponding state index.

5. Implementation

Since we measured absolute run times of our program to be able to compare it with other implementations is crucial to optimize every aspect of the search. In this chapter we show how we improved the performance of IDA*, the usage and generation of the pattern databases, as well as applying moves to the cube.

5.1. IDA*

The implementation shown in Algorithm 3 shows a recursive implementation of IDA* but tests have shown that an implementation without recursion yields better performance. To avoid the recursion we used a state stack on which the nodes adjacent to the currently processed nodes are pushed on. The entries of the state stack consist of the state of the cube, the depth this configuration was found at and the move that lead to this state. In the Algorithms 1 and 3 a stack that stored all moves that were applied was used to extract the sequence that solves the cube. But when no recursion is used the move stack cannot be used normally, since jumps from nodes at high depths to nodes at low depths can occur. This problem is solved by creating an null move, that is always set at the next place in the move stack. When the search is finished only the moves up to the null move are used as the solution. A more performant way to realize the move stack is to use an array with length 21 (since we know that no solution will be longer than 20) and indexing it with the depth of the current node.

Before a node is pushed on the state stack we check if it is pruned by the duplicate state detection or by the pattern database heuristic, instead of doing this for nodes taken from the state stack to avoid multiple stack operations.

Another optimization for IDA* is to omit the storing of the smallest cost threshold and just increase it by one in between every iteration. This can be done since a threshold delta of two adjacent states that is greater than 1 can only occur if a the heuristic yields two values of neighbour edges with a difference greater than 1 but the pattern databases used do not have this property even when combined.

5.1.1. Parallization

Our implementation used 18 threads for all experiments that we are shown in Chapter 6. The parallelization is realised by distributing the branches that branch of the first node (the scrambled state) onto 18 different threads. Each thread is running IDA* on a branch of the state tree. As soon as one thread finds a solution the search is aborted. This approach to parallelize IDA* theoretically can be used with as many threads as desired but when the expected sizes of the branches distributed to the threads is not equal, some threads will stop traversing earlier. This leads to a less utilization of the parallelization, since new threads will only be spawned when all others have finished. Therefore 18 threads is a good candidate, since there are exactly 18 adjacent states to the solved state for which an equal amount of subsequent states can be expected. Figure 5.1 shows how the 18 adjacent states of the solved state would be distributed when six threads would be used.

The state-index-based duplicate state detection has the potential of race conditions when more than one thread accesses the hash table. One thread could reach a state at the same time as another state does before the state was marked as visited. So both threads would continue traversing the subsequent nodes. To fix this some sort of semaphore would be required. But our tests have shown, that this event is so unlikely and that the consequences are so negligible that using some sort race condition prevention results in a worse worse performance.

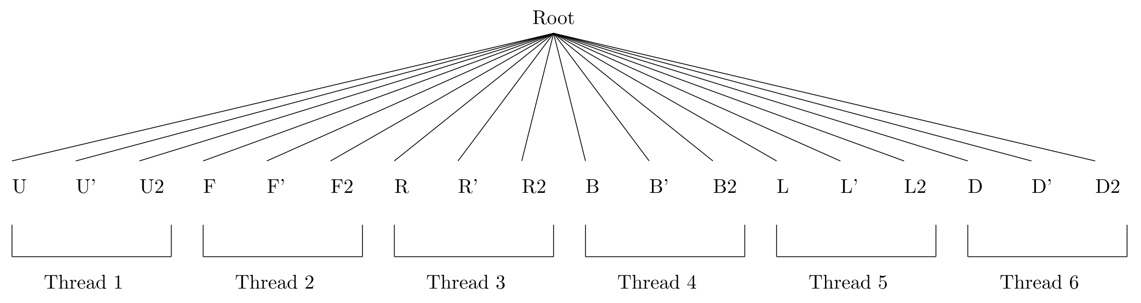

Figure 5.1.: IDA* Parallelization

5.2. Pattern Databases

The pattern databases (especially important for the full edge pattern database) only need to be generated once but still optimization were made for generation, that allowed us to generate all database in under three days. But during the solve the performance of the usage of the databases also can be improved. The improvements made in our implementation are presented in the next two sections.

5.2.1. Generation

The generation of the pattern databases, especially the full edge database, required substantially less time with parallelization. The inverse lexicographic rank technique presented in Section 3.3.1.3 can easily be parallelized by dividing the database in equally sized sections and distributing the population to multiple threads. The IDDFS step together with the optimizations described in Section 5.2.1 require a lit more deliberation when parallelized, since without any prevention racing conditions can lead to false corrupted results. The first obvious problem is that when counting how many states were reached the variable that stores that information could be accessed by two threads at the same time and incremented only once. This can be fixed by using an atomic integer. Another problem is that if a state is reached at the same time by two threads and they both check if the entry is unset before the other thread can set it, the number of states is counted incorrectly. So before checking an entry and until it is set a semaphore needs to be used. This leads to worse performance per thread, but ensures that the pattern database is generated correctly.

When reconstructing the configuration of the cube from its lexicographic rank some small optimizations were used for the partial permutations of the edge groups. When finding the positions of the 12 edges which indices are used to generate the rank usually all pieces would need to be processed, but a counter can be used to stop when all positions were found. When the indices of the relevant edges are set from the rank the remaining positions theoretically would be need set to the remaining indices to generate a valid configuration of the cube. But since the configuration will only be used to compute the edge group rank of an adjacent state the indices of the remaining positions can be set to a constant value. Note that this constant value may not interfere with the indices of the edge indices of interest.

5.2.2. Lookup

The size of the pattern databases exceeds the cache sizes of all three caches of most household machines. The full edge pattern database for example is nearly four thousand timer bigger than the L3 cache of the machine all tests ran on. This leads to a lot of cache misses when accessing the pattern database, so every unnecessary lookup should be avoided. An unnecessary lookup could be when we are at depth 6 and have a depth bound of 12 in the current iteration and one lookup already yielded a distance greater than 6. Regardless of the distances the other pattern databases contain the state will be pruned. So one optimization we used is to pass the smallest value for which a the state will be pruned to the lookup function and only access the other pattern databases if no previous lookup exceeded that value. The lookups were ordered by the expected values of the pattern databases.

The pattern databases require some bit manipulation when compressed, so we opted to use uncompressed pattern databases during the solve.

5.3. Rubik's Cube Representation

Applying moves to the cube is on of the most frequent invoked methods of our program so every optimization applied to this part of the code yields more extensive performance improvements. In Section 3.1.2 we already pointed out that the selection of the upper sides and upper facelets of corners and edges influences the number of operations all moves together require. The upper sides of the corners point in the directions the top and bottom layer face. This allows any U and B rotation as well as every 180° rotation to leave the orientations unchanged. Also the upper sides for the edges in the top and bottom layer should face in the same direction to decrease operations. When the upper side of the edges of the front and back layer also point in the direction of the two faces, only 90° rotations of the front and back layer need to adjust edge orientations.

The adjustment of the corner orientation increases the orientation by 1 or 2 and then takes the sum modulo 3 as the result. A mod 3-operation is expensive and can be replace by two if statements that check if the sum is 3 or 4 and set it to 0 or 1 respectively. The edge orientations follow the same rules, but can only have orientation values of 0 and 1, so the addition and the mod 2-operation can be replaced by an XOR-operation with a binary 1.

5.4. Duplciate State Detection

To generate the table containing the sequence indices and the pruning flags with an recursive implementation the sequence index does not have to be computed completely for every entry but rather every move applied and removed from the sequence can adjust the index by adding or subtracting the move index times the depth term of Equation 4.2. This can only be done when an iteration of IDDFS sets entries at the depth bound, because n in the equation would be variable otherwise.

The state-index-based duplicate state detection, contrary to the sequence-index-based method, not only reads but also sets flags in the lookup table. When using multiple threads racing conditions can occur that would lead to duplicate states that will be traversed further by different threads. But instead of using some techniques to prevent the racing conditions we opted to allow some states to be traversed multiple times since tests have shown that the parallelization overhead of such preventive methods have lead to worse results than just allowing the racing conditions.

6. Evaluation

This chapter begins by specifying the machine that was used for all solves presented in this work. Results of solves that did not use the full edge pattern database nor any duplicate state detection are shown first in Section 6.2 to establish a base line all other solve configurations can be compared to. Section 6.3 shows the result of solving 5 000 shuffles with the full edge pattern database. Section 6.4 first presents the results of 15 solves that did use the state-index-based duplicate state detection technique and then compares all the different techniques with different maximal sequence lengths with data of 100 shuffles for each configuration. Lastly we show the results of using solve configurations on shuffles that are known to have a distance of 20 moves to be solved. The last section discusses all the results.

We generated a list of 5 000 scrambles by applying 100 random moves for each scramble. The only rule used for generating theses moves was not to allow two consecutive turns of the same face. All results presented used the generated shuffles. All experiments that solved less than 5 000 shuffles used the first shuffles in this list. All results we show used 18 threads and parallelized the search as described in Chapter 5.

Our implementation can be found on GitHub:

https://github.com/TanGrumser/rubikscube-solver

The version we used has the commit hash:

ec8b15535214088f2a16ed7a45d7f72d0ee5a03b

6.1. Setup

The specifications of the machine used for all solves presented in this work can be found in Table 6.1.

OS	Ubuntu 20.04
CPU	AMD EPYC Rome 7702P - 64-core + HT, 2.0-3.35GHz
RAM	1024GB DDR4 ECC
Caches	L3: 256MB
Hard Disks	512GB NVMe System-SSD, Intel 7600P , 2 x 2TB NVMe Daten-SSD, Intel P4510

Table 6.1.: Specification of the Machine

solution length	nodes generated	time (seconds)
16	274 289 501	125
17	729 252 017	327
17	913 874 902	413
17	1 253 793 474	559
17	1 824 394 068	857
17	3 531 115 249	1 568
18	3 481 595 519	1 726
18	5 097 761 131	2 364
18	7 105 639 133	3 291
18	9 435 875 048	4 265
18	9 654 313 831	4 457
18	10 356 872 845	4 691
18	11 423 610 423	5 316
18	19 863 983 642	8 855
18	22 372 292 049	10 039
18	39 526 346 646	17 509

Table 6.2.: Results with 4 Pattern Databases

6.2. Baseline

To establish a baseline all other solve configurations can be compared to we solved 16 scrambles. For this we only used the corner, the two edge group and the edge permutation pattern databases and no duplicate state detection. The number of nodes generated and the time required for all solves can be found in Table 6.2. For these solves our program generated about 2 260 000 nodes per second. On average it took about 4 150 seconds and about 9 177 813 092 generated nodes to solve the scrambles.

6.3. Full Edge Pattern Database

With the full edge and the corner pattern database we solved 5 000 scrambles. The absolute and relative distance distributions of the scrambles can be found in Figures A.1 and in Figure A.2 respectively in Appendix A. The distribution fits to the findings made before during the search for God's Number [8]. The full edge pattern database decreased the average time to 78 seconds and the average nodes generated to 149 881 988 which is about 61.6 times fewer nodes than with the four lookup tables used by Botto and solved shuffles about 53.2 times faster on average. This solving configuration was able to generate about 1 897 973 nodes per second. So about 17 percent less nodes per seconds were generated.

The number of nodes generated and the time required for a solve with respect to the distance of a shuffle to the solved state are shown Figures 6.1 an 6.2 respectively.

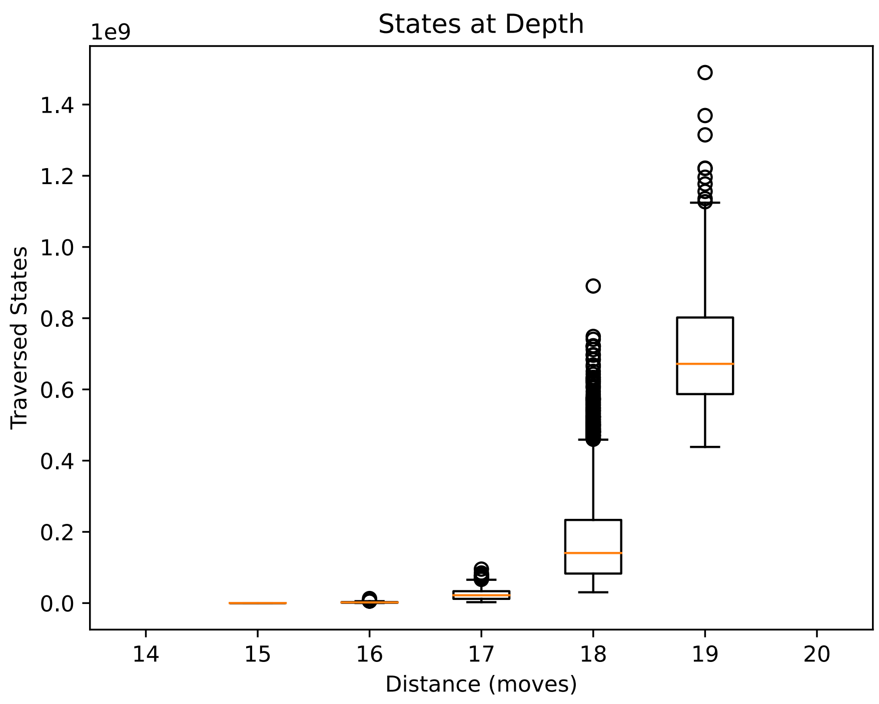

Figure 6.1.: Generated Nodes w.r.t. the Distance

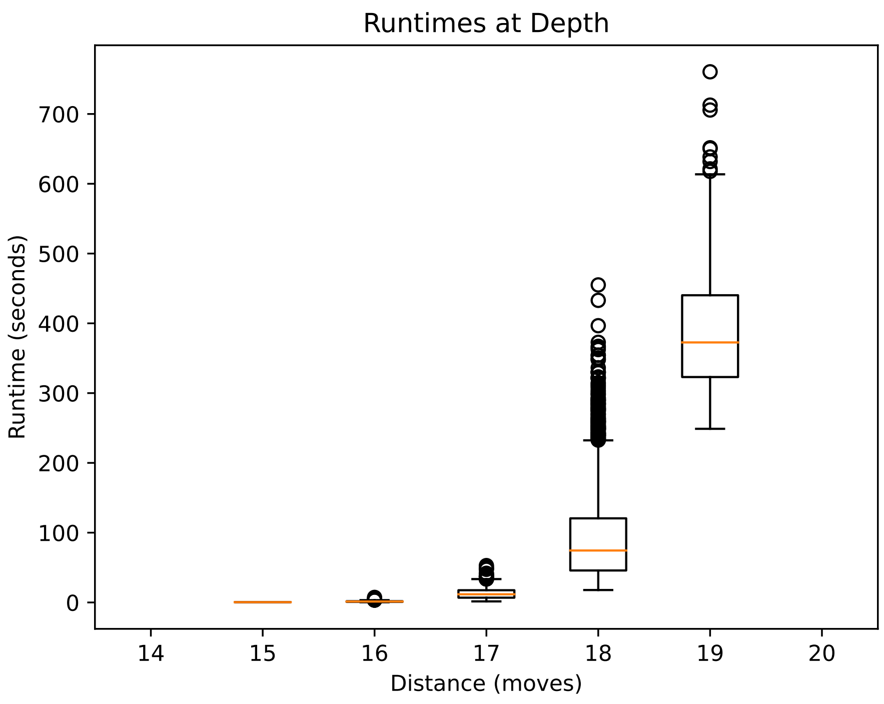

Figure 6.2.: Solving Time w.r.t. the Distance

6.4. Duplicate State Detection

We introduces two duplicate state detection techniques. Both have a variable depth limit, to which duplicate states are generated and detected. We show the results of various solving configurations with and without the use of the full edge pattern database.

6.4.1. 4 Pattern Databases

We solved the first ten of the five thousand scrambles with both duplicate state detection techniques, each with depth limits of 7 and 8. These solves did not use the full edge pattern database. The results of the different solving configurations are shown in Table 6.4.1. The right most column only shows results of the first ten out the 16 solves presented in Section 6.2.

DS Detection	Inverse state index		turn index		Off
Max Depth	7	8	7	8	-
Median Time	2 785	2 819	2 755	2 718	4 266
Average Time	3 913	3 961	3 834	3 751	5 026
Median Nodes	5 940 419 084	5 905 975 072	5 883 089 555	5 784 436 571	9 435 875 048
Average Nodes	8 563 754 644	8 581 022 002	8 387 809 751	8 184 571 366	11 203 560 198
Nodes / second	2 188 633	2 166 309	2 187 918	2 181 974	2 229 318

Table 6.3.: 4 Pattern Databases - Duplicate State Detection

These results show that the duplicate state detection techniques introduce can decrease the time required to solve scrambles with Korf's algorithm. With the state-index-based method, the solving time was decreased by over 22 percent on average with a depth bound of 7. Increasing the depth bound even further finds more duplicate states and decreases the number generated states in total, but the overhead of the the shifted state index computations are too high to improve the performance. With the sequence-index-based technique the performance could be improved by nearly 24 percent in exchange for more memory usage. With this method also a depth bound of 7 yielded better results, since the sequence index calculation again raised too much overhead.

6.4.2. Full Edge Pattern Database

We solved the first one hundred of the five thousand generated scrambles with four different solving configurations. Table 6.4.2 shows the results of these solves. The right most column shows the results of the five thousand solves shown in Section 6.3 for comparison.

DS Detection	Inverse state index		turn index		Off
Max Depth	7	8	7	8	-
Median Time	50.08	55.3	47.08	48.3	51.82
Average Time	66.38	70.92	63.7	64.16	78.83
Median Nodes	85 767 000	85 238 026	83 977 156	82 856 152	94 201 241
Average Nodes	123 556 034	123 245 932	122 908 362	121 287 346	149 530 656
Nodes / second	1 861 444	1 737 827	1 929 565	1 890 457	1 896 959

Table 6.4.: Edge group configurations distribution per depth

The table shows the same characteristics as the results without the full edge pattern database. This means that the duplicate state detection method still works for greater heuristic values.

6.4.3. Nodes Generated and Time required

We solved all five thousand scrambles with the state-index-based duplicate state detection and the full edge pattern database. The number of nodes generated and the time required for a solve with respect to the distance are shown in Figures 6.3 and 6.4 respectively.

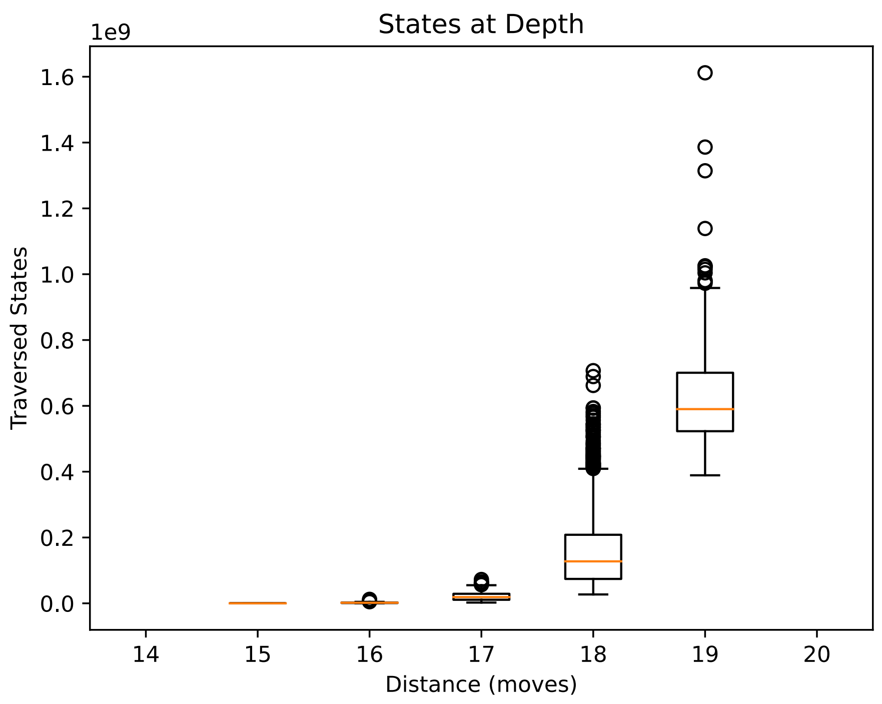

Figure 6.3.: Generated Nodes w.r.t. the Distance (Edge Pattern Database)

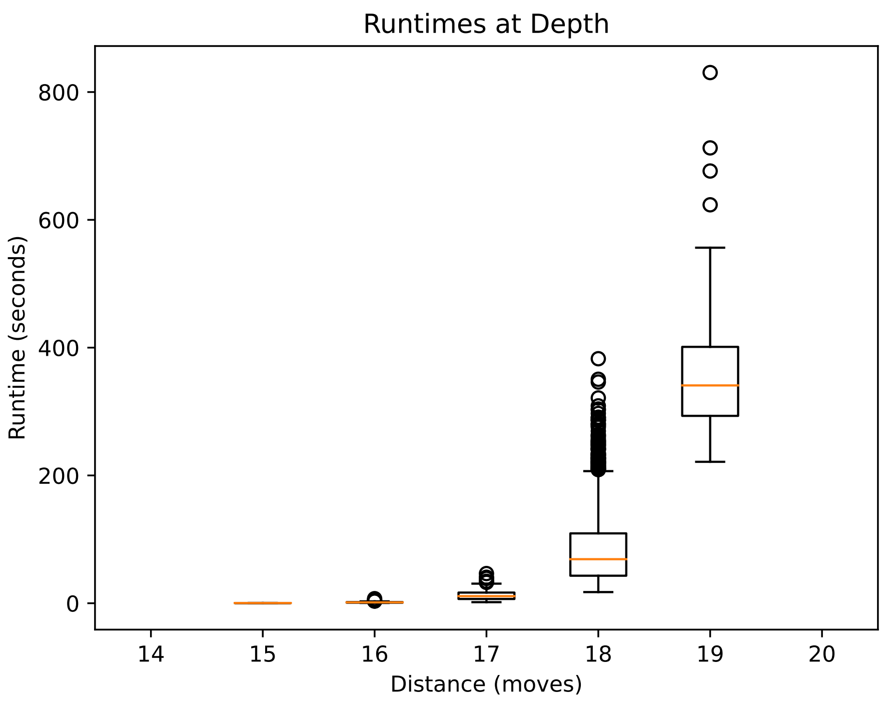

Figure 6.4.: Solving Time w.r.t. the Distance (Edge Pattern Database)

The figures show, that the statistical outliers with a high number of generated nodes and seconds of solving time occur less often. The duplicate state detection therefore is able to normalize these metrics over many solves.

6.5. Solving Distance 20 Shuffles

Since none of the randomly generated shuffles had a distance of 20, we took 10 random distance 20 scrambles from an online database and solved them without the duplicate state detection and with all four other solving configurations. The result of these solves can be found in Table 6.5

DS Detection	Inverse state index		turn index		Off
Max Depth	7	8	7	8	-
Median Time	3 036	3 067	3 026	2 991	4 266
Average Time	3 052	3 085	3 036	3 004	5 026
Median Nodes	5 551 051 059	5 531 074 376	5 502 933 823	5 430 530 683	9 435 875 048
Average Nodes	5 570 734 478	5 550 685 951	5 526 246 069	5 453 420 207	11 203 560 198
Nodes / second	1 824 797	1 798 889	1 819 825	1 814 959	2 229 318

Table 6.5.: Distance 20 Shuffles - Duplicate State Detection

The table shows that for solves with a distance of 20 the sequence-index-based duplicate state detection performed better with a depth limit of 8. But since distance 20 scrambles are extremely unlikely [12], an overall better performance can be reached when using a depth limit of 7.

6.6. Discussion

We found that with the two duplicate state detection methods we introduced and the full edge pattern database we were able to solve random cube instances in under 65 seconds on average. This is over 19 percent faster than without any duplicate state detection. With a smaller memory overhead the state-index-based duplicate state detection improved solving times over 15 percent. Without the full edge pattern databases the duplicate state detection methods could decrease the solve time by about 22 percent. With a smaller memory overhead the method still improved the times by nearly 24 percent.

We showed that the with a greater depth bound for the duplicate state detection number of nodes generated are decreased even further but these methods decreased the nodes that could be processed per second. We also encountered some counter intuitive results. The number of nodes generated per second with the full edge pattern database is smaller then the number of node generated without the edge pattern database. This is caused by the size ratio of the database and the cache of the machine the experiments were run on. Since the pattern databases show no signs of cache coherence nearly every access will yield a L3 cache miss. There is no known way to address this problem. The smaller pattern databases do not exceed the cache sizes of the machine in the same way so cache misses occur less often which leads to more nodes that can be generated per second.

Some tables show less nodes generated for the sequence-index-based duplicate state detection as the state-index-based duplicate state detection. As mentioned in Section 5.4 we do not use any techniques to avoid racing conditions when accessing the duplicate state hash table. This results in some duplicates states, that are not pruned. The turn index bases approach cannot miss duplicate states through racing conditions since the duplicate sequence table is only read but never written to. This explains why the number of nodes generated is lower for some experiments using the sequence-index-based duplicate state detection.

7. Conclusion

This work highlighted some techniques for optimally solving the Rubik's Cube. Korf's algorithm was presented in more detail together with the results of the best known implementation of it. An insight on the the method for generating pattern databases was given, together with methods for improving the performance of the generation and the performance of the other parts of the program. Using this foundation we introduced the results of using the biggest pattern database generated yet were shown. Then we introduces two techniques for detecting duplicate states. These techniques in isolation and together with the full edge pattern database used to solve 5 000 random scrambled cubes. Now we present some problems these techniques could be used for to try to improve the performance of the best implementations. Lastly we present some thoughts on how heuristics can be combined better on other puzzles like the 4x4x4 cube.

7.1. Future Work

The duplicate state detection is not bound to the problem space of the Rubk's Cube and may improve the performance of solvers for other combinatorial puzzles as well. Linked with the Rubik's Cube are the 15 tile puzzle, the Top-Spin puzzle and the Tower of Hanoi. All of them are combinatorial puzzles with similar properties of the problem space. But even without leaving the Rubik's Cube problem domain further research on applications of the presented techniques can be made. Other solvers that use different approaches for solving the cube may benefit from these techniques. The usage of symmetries of the cube may have the potential to be well combined with the duplicate state detection.

For Korf's algorithm an obvious next step is to think about other pattern databases that use the state of corners and edges combined instead of considering them in isolation. A pattern database that uses 6 edges and 4 corners, for example, would only be about six times larger then the pattern database of all edges. The pattern database containing all permutations of edges and corners (disregarding the orientation) would be about 20 times bigger.

7.1.1. Hash Tables

Neither the hash table we used to store the duplicate states nor the hash function used had special properties, but since all elements that will lay in the hash table are known beforehand. So special algorithms for handling static hash tables and better hash functions could further improve the performance of the state index based duplicate state detection.

7.1.2. Pattern Databases Combination

From the different pattern databases the maximum is used, but somehow combining the different information to obtain greater approximations for the heuristic would greatly improve the performance. Since every non trivial combination of moves on the Rubik's Cube does not transform any pieces in isolation, no intuitive groups of pieces could be used for pattern databases that can be combined without just taking the maximum. In early iterations of this work we worked on some concepts to improve the lower bound of the $4 \times 4 \times 4$ Rubik's Cube, also known as the Rubik's Revenge. A move on this cube can not only turn the outer layers but also inner layers. Since rotations of the inner layers do not affect the corners and rotations of the outer layers do not affect the centers relative to each other on on one side heuristics that combine information of the two subsets of pieces could be used obtain greater heuristic values than the pattern databases for the 3x3x3 Rubik's Cube could provide.

There is also ongoing research on how to combine heuristics that do not affect each other. The knowledge of these works could be used to attempt to find optimal solutions to harder scrambles on the 4x4x4 cube and improve the lower bound for its God's Number.

Bibliography

[1] Blai Bonet. "Efficient algorithms to rank and unrank permutations in lexicographic order". In: *AAAI-Workshop on Search in AI and Robotics*. 2008, pp. 142–151.

[2] Ben Botto. *Implementing an Optimal Rubik's Cube Solver using Korf's Algorithm*. URL: `https://medium.com/@benjamin.botto/implementing-an-optimal-rubiks-cube-solver-using-korf-s-algorithm-bf750b332cf9` (visited on 07/27/2022).

[3] Ben Botto. *Sequentially Indexing Permutations: A Linear Algorithm for Computing Lexicographic Rank*. URL: `https://medium.com/@benjamin.botto/sequentially-indexing-permutations-a-linear-algorithm-for-computing-lexicographic-rank-a22220ffd6e3` (visited on 07/27/2022).

[4] Teresa Maria Breyer and Richard Korf. "1.6-bit pattern databases". In: *Twenty-Fourth AAAI Conference on Artificial Intelligence*. 2010.

[5] George Webster CNN. *The little cube that changed the world*. 2012. URL: `https://edition.cnn.com/2012/10/10/tech/rubiks-cube-inventor` (visited on 07/15/2022).

[6] Tom Davis. *Group theory via Rubik's cube*. 2006. URL: `http://www.geometer.org/rubik/group.pdf`.

[7] *Definition:Singmaster Notation*. URL: `https://proofwiki.org/wiki/Definition:Singmaster_Notation` (visited on 07/15/2022).

[8] *God's Number is 20*. URL: `https://cube20.org` (visited on 07/27/2022).

[9] *God's Number is 26 in the Quarter-Turn Metric*. URL: `https://cube20.org/qtm/#:~:text=God's%5C%20Number%5C%20is%5C%2026%5C%20in%5C%20the%5C%20Quarter%5C%2DTurn%5C%20Metric` (visited on 07/27/2022).

[10] *How to Solve a Rubik's Cube in 5 Seconds—or Less*. URL: `https://www.wired.com/story/how-to-solve-a-rubiks-cube-in-5-seconds-or-less/` (visited on 07/27/2022).

[11] David Joyner. "The man who found God's number". In: *The College Mathematics Journal* 45.4 (2014), pp. 258–266.

[12] Herbert Kociemba. *The Mathematics behind Cube Explorer*. URL: `http://kociemba.org/cube.htm` (visited on 07/27/2022).

[13] Richard E Korf. "Depth-first iterative-deepening: An optimal admissible tree search". In: *Artificial intelligence* 27.1 (1985), pp. 97–109.

[14] Richard E Korf. "Finding optimal solutions to Rubik's Cube using pattern databases". In: *AAAI/IAAI*. 1997, pp. 700–705.

[15] Richard E Korf. “Linear-space best-first search: Summary of results”. In: *AAAI*. 1992, pp. 533–538.

[16] Richard E Korf and Peter Schultze. “Large-scale parallel breadth-first search”. In: *AAAI*. Vol. 5. 2005, pp. 1380–1385.

[17] Brian Glen Patrick. “An analysis of iterative-deepening-A*”. In: (1991).

[18] Tomas Rokicki. “Twenty-five moves suffice for Rubik’s cube”. In: *arXiv preprint arXiv:0803.3435* (2008).

[19] Tomas Rokicki et al. “The Diameter of the Rubik’s Cube Group Is Twenty”. In: *SIAM Journal on Discrete Mathematics* 27.2 (2013), pp. 1082–1105. DOI: `10.1137/120867366`. eprint: `https://doi.org/10.1137/120867366`. URL: `https://doi.org/10.1137/120867366`.

[20] Jerry Slocum. *The Cube. The Ultimate Guide to the World’s Bestselling Puzzle. Secrets – Stories – Solutions.* New York: Black Dog & Leventhal, 2009.

[21] Massachusetts Institute of Technology. *The Mathematics of the Rubik’s Cube*. 2009. URL: `https://web.mit.edu/sp.268/www/rubik.pdf` (visited on 05/15/2022).

[22] *Thistlethwaite’s 52-move algorithm*. URL: `https://www.jaapsch.net/puzzles/thistle.htm` (visited on 07/27/2022).

[23] BBC News Magazine Tom de Castella. *The people who are still addicted to the Rubik’s Cube*. 2014. URL: `https://www.bbc.com/news/magazine-27186297` (visited on 06/29/2022).

A. Appendix

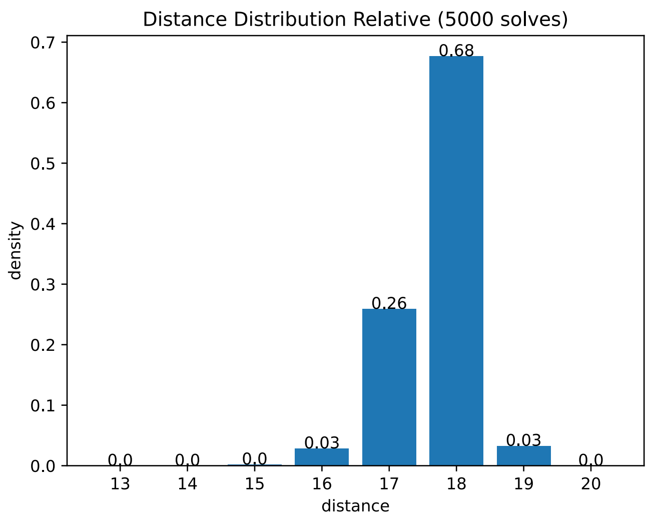

Figure A.1.: Distance Distribution Density

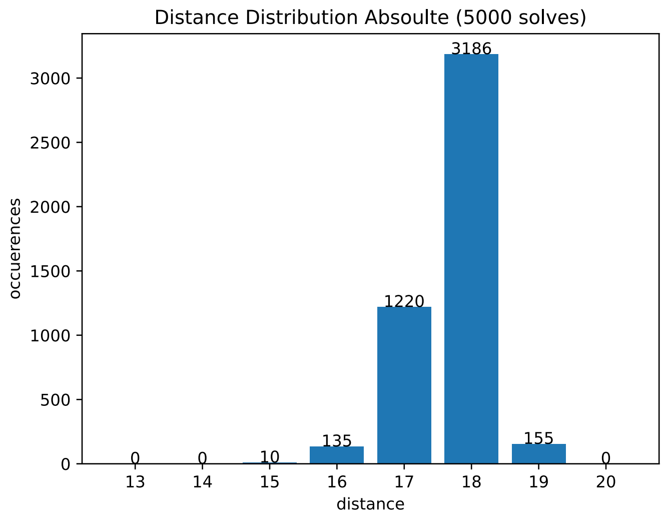

Figure A.2.: Absolute Distance Distribution