Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer, Communication and Electronic Engineering

Final Dissertation

Computational Analysis of Solving the Rubik's Cube

Implementation of $\boldsymbol{H_{48}}$ *Optimal Solver*

Supervisor
Prof. Alberto Montresor

Student
Enrico Tenuti

Co-supervisor
Ph.D. Sebastiano Tronto

Academic year 2023/2024

Acknowledgements

Thanks to my family, my friends, my supervisor, my co-supervisor and all the people who helped me during this journey. This work was born from a passion for the Rubik's Cube and for computer science, and I am grateful to have had the opportunity to combine these two passions in this project. I would like to thank my supervisor, Prof. Alberto Montresor, for believing in me and for his support throughout the development of this project. I would also like to thank my co-supervisor, Ph.D. Sebastiano Tronto, for accepting me in the development of this new solver and for his valuable advice in the development of the project.

This experience made me grow as a developer and as a person, and I am very grateful to have had the opportunity to work on this project. I had the opportunity to learn many new things and to improve my skills in programming and in problem solving, using the knowledge I acquired during my studies which varied from graph theory to multithreading. The practical application of these theoretical concepts has significantly enhanced my understanding of both the mathematical and computational aspects of solution optimization. The interaction with different people around the world has been particularly enriching, allowing me to gain new perspectives on problem-solving approaches and algorithm development. This international collaboration has not only improved the technical aspects of our work but has also provided valuable insights into the global cubing community's needs and expectations.

Contents

Abstract

This thesis presents a computational study of **Rubik's Cube** solving methods, focusing on the implementation of the $\mathbf{H_{48}}$ optimal solver. After an introduction to the cube's structure and notation, the thesis provides some **basic group theory concepts** to describe the puzzle in a more formal way.

The study examines various methods used to solve the Rubik's Cube, fwith a focus on the differences between human and computer-based approaches. Humans typically rely on memory and pattern recognition, while computers use systematic algorithms and brute-force techniques. Different types of solvers are analyzed, with particular attention given to distinguishing between optimal and suboptimal methods. Additionally, the study highlights the importance of data representation through encodings that manage the cube's state and the operations on it. A specific focus is given to the implementation of these encodings using **ARM NEON intrinsics**, which leverage **SIMD** (Single Instruction, Multiple Data) instructions to enhance the performance of the $\mathbf{H_{48}}$ solver. Some core functions are presented in detail, emphasizing the use of these vectorized instructions to speed up the solving process.

In analyzing the $\mathbf{H_{48}}$ solver, advanced computational techniques that contribute to its effectiveness are explored. This includes the use of **IDA*** (Iterative Deepening A*) algorithm, which combines depth-first and heuristic search to identify optimal solutions without excessive memory usage. **Pruning tables** play a critical role in reducing the search space of the IDA* algorithm by encoding heuristic values that provide lower bounds on the number of moves required to solve each position. The thesis discusses the development of a new pruning group, never used before in optimal solvers, which aims to improve the solver's performance by reducing the number of lookups in the pruning tables.

Some other optimization techniques are discussed, such as the use of a **multithreaded approach** to parallelize the search process. This technique is particularly useful for solving the Rubik's Cube, as it allows for the simultaneous exploration of multiple branches of the search tree.

The thesis concludes by presenting the results of this computational analysis, showcasing the strengths of the employed techniques. We compared the results of the portable version of the solver with the SIMD version, highlighting the performance improvements achieved through the use of vectorized instructions. The results from using the multithreaded search are also presented and the use of different sizes of pruning tables with their respective nodes visited and time taken. Finally, the $\mathbf{H_{48}}$ solver is compared with the current reference point for optimal solvers, **VCube**, and noted the potential of the $\mathbf{H_{48}}$ solver to find new optimal solutions and serve as a basis for further research in the field of optimal solvers for the Rubik's Cube.

1 Introduction

The Rubik's Cube is a combinational puzzle invented in 1974 by Hungarian architect and professor Ernő Rubik. The professor's intention was to create a rotating object to demonstrate a structural-geometric problem.

This invention quickly became the world's best-selling toy and one of the most widespread products globally. Although the Rubik's Cube reached its peak of mainstream popularity in the 1980s, it remains widely known and used today. In fact, the past ten years have seen a true renaissance of the Rubik's Cube, bringing it back into the spotlight. It attracts the attention of both young and old with its simple six-faced, six-colored structure, yet it hides a significant challenge in solving it.

To some, it is considered a design object, while to others, it represents a scientific problem, involving various mathematical properties such as combinatorics, group theory, and symmetries. The applications of the Rubik's Cube have led to indirect benefits in various fields, thanks to its wide-ranging characteristics [1].

1.1 Structure of the Cube

The original Rubik's Cube structurally consists of a three-dimensional grid of 27 sub-cubes that we can call **cubies**. Each cubie has a specific number of visible **facelets** (or stickers), and this number characterizes the type of piece. One of the cubies is not visible, as it coincides with the core part of the cube. The remaining 26 cubies can be divided into three types of pieces: centers, edges, and corners.

- **Centers:** There are 6 center pieces, one on each face of the cube. Each center has a single color and only one possible orientation, as it is fixed relative to its face.
- **Edges:** There are 12 edge pieces, each located between two centers. Each edge has two possible orientations (since it is two-colored and can be flipped).
- **Corners:** There are 8 corner pieces, each at the intersection of three centers. Each corner can be oriented in three distinct ways (since each corner piece has three colors and can be rotated in three different orientations).

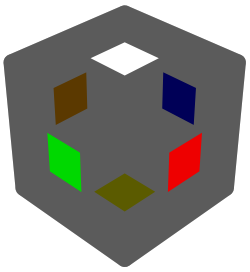

Figure 1.1: Centers

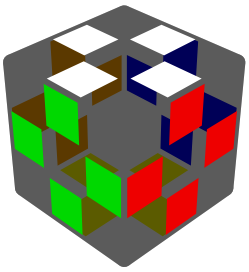

Figure 1.2: Edges

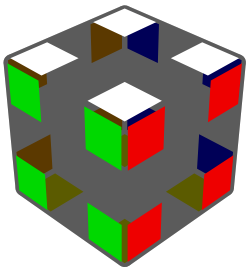

Figure 1.3: Corners

In addition to the standard 3x3x3 cube, there are larger versions available, such as the 4x4x4 (commonly known as the "Revenge Cube") and the 5x5x5 (referred to as the "Professor's Cube"). These larger cubes feature a different number of centers and edges. For example, the 4x4 cube has non-fixed centers, and its edges are composed of two smaller pieces known as wings. The 5x5 cube introduces different types of pieces, including centers composed of a fixed piece, X-centers, and +-centers, as well as edges formed by a midge and two wings. Additionally, there are shape modifications of the Rubik's Cube, with the most notable being the Platonic solids: the Megaminx (a dodecahedron) and the Pyraminx (a tetrahedron).

1.2 Cube Permutations

Using combinatorial mathematics, it is possible to calculate the total number of permutations for a Rubik's Cube. As previously mentioned, the cube is composed of three types of pieces, but for the purpose of calculating permutations, the focus will be solely on the corners and edges, assuming the cube orientation on its axis is fixed and that centers have no orientation constraint.

The calculation is divided into two components: position and orientation. The position refers to where each piece is located on the cube, while orientation describes how each piece is aligned in its given position. Consider a cube that has been disassembled, allowing one piece to be added at a time. The first corner can be placed in one of eight positions, the next in one of seven, and so forth. Therefore, the number of possible corner positions is 8!. Applying the same logic to the edges, the number of possible edge positions is 12!.

Next, the orientations are considered. As noted earlier, each corner has three possible orientations, and each edge has two. Consequently, the number of possible orientations is multiplied by the number of pieces, yielding the total number of permutations:

$$P(n) = 8! \cdot 12! \cdot 3^8 \cdot 2^{12} \approx 5.2 \times 10^{20} \tag{1.1}$$

This is an exceedingly large number. To provide some context, it can be compared to the estimated number of seconds that have elapsed since the Big Bang: approximately 4.35×10^{17} seconds over 13.8 billion years. However, this figure does not represent the number of possible solvable permutations. When starting from a solved cube and applying random moves, the number of possible permutations is reduced due to structural constraints.

To understand why not all permutations are solvable, it is necessary to analyze the effects of a move. A 90-degree rotation of a face results in a 4-cycle of corners and edges. Essentially, this move involves three swaps of corners and three swaps of edges. To return the cube to its solved state, an equal or multiple number of swaps is required. As observed, the parity of the edges and corners remains unchanged with each move, alternating between odd and even numbers of swaps for each group of pieces.

If a solved cube is altered by swapping a pair of edges or corners, the parity between the two groups changes, rendering the cube unsolvable using standard moves. To account for this positional parity in the calculation, while the corners can be placed in any position, the placement of the edges must ensure that they maintain the same parity as the corners. Therefore, the total number is divided by two, as the final two positions are constrained:

$$P(n) = \frac{8! \cdot 12! \cdot 3^8 \cdot 2^{12}}{2} \approx 2.6 \times 10^{20} \tag{1.2}$$

This calculation, however, is still incomplete as orientation parity must also be considered. In a solved cube, all corners and edges are correctly oriented. Each move affects the orientations of certain pieces, but the sum of their orientations, modulo the total number of possible orientations of the piece type, should always equal zero. By incorporating these factors, the true number of solvable permutations is:

$$P(n) = \frac{8! \cdot 12! \cdot 3^8 \cdot 2^{12}}{2 \cdot 2 \cdot 3} \approx 4.3 \times 10^{19} \tag{1.3}$$

Understanding parity is crucial for solving the Rubik's Cube. In larger variants of the cube, additional types of parity errors can occur because certain pieces may be swapped without affecting the rest of the cube.

1.3 Move Notation

To formalize the set of moves of the Rubik's Cube into a group, it is first necessary to define what constitutes a move. The widely accepted notation for the Rubik's Cube is known as the Singmaster Notation, named after its inventor, David Singmaster, an American mathematician who served as a professor at London South Bank University for several years.

In the standard orientation of a Rubik's Cube, the white face is positioned on top and the green face in front. Each face is designated by a letter: up (`U`, white), down (`D`, yellow), front (`F`, green), back (`B`, blue), right (`R`, red), and left (`L`, orange).

Each face can be moved in three ways:

1. A 90-degree clockwise turn, denoted by the plain move (e.g., `U`).
2. A 90-degree counter-clockwise turn, denoted with an apostrophe (e.g., `U'`, pronounced "U prime").
3. A 180-degree turn, denoted with a 2 following the letter (e.g., `U2`).

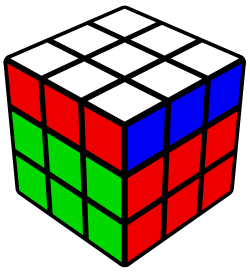

Figure 1.4: `U` move

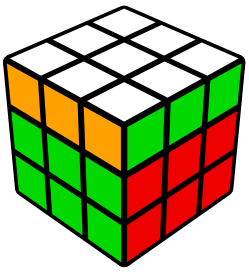

Figure 1.5: `U'` move

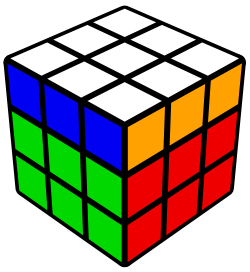

Figure 1.6: `U2` move

Additionally, rotations around the cube axes are represented by another set of letters: `x` for a rotation on the `RL` axis (following the `R` clockwise direction), `y` for a rotation on the `UD` axis (following `U`), and `z` for a rotation on the `FB` axis (following `F`).

1.4 Metrics

After establishing the notation, it is crucial to understand how moves are counted. There are three main metrics used to count the number of moves applied to a cube:

- **Half Turn Metric (HTM)**: Every move in the Singmaster notation is counted as a single move, including half turns.
- **Quarter Turn Metric (QTM)**: Counts half turns as two moves.
- **Slice Turn Metric (STM)**: Introduces three additional moves to represent the inner layers (`M`, `E`, `S`), that are counted as a single move each. Unlike HTM and QTM, STM is not center-fixed.

To illustrate the differences between these metrics, consider the sequence `R2 L2`. This sequence is counted as 2 moves in HTM, 4 moves in QTM, and 1 move in STM (equivalent to `M2 x2`).

Other metrics exist, such as the Outer Block Turn Metric (OBTM), which incorporates wide moves. OBTM is the standard used by the World Cube Association in speedcubing competitions [2].

The analysis in this study will focus solely on the Half Turn Metric (HTM), which is the most prevalent in computer and mathematical works related to the Rubik's Cube. This choice allows for a more standardized approach to move counting and aligns with the majority of computational and theoretical studies in this field.

The metric factor is crucial in terms of solving methods and optimal solutions because a solution that is optimal in one metric may be suboptimal in another. Using different types of metrics can lead to different heuristic strategies applied to the solver, a topic that will be discussed in detail in subsequent chapters.

2 Group Theory

To better understand the structure of the **Rubik's Cube puzzle** and more accurately characterize its properties, it is necessary to provide a brief explanation of some basic group theory concepts. As this study does not examine advanced group theory, the focus will be on the fundamental ideas and tools useful for describing the problem. This chapter primarily draws upon the work of Janet Chen [3].

2.1 Definitions

Definition 1 (Group)**.** *A group $(G, *)$ consists of a set G and an operation $*$ such that:*

1. *G is closed under $*$. That is, if $a, b \in G$, then $a * b \in G$.*
2. *$*$ is associative. That is, for any $a, b, c \in G$, $a * (b * c) = (a * b) * c$.*
3. *There exists an identity element $e \in G$ which satisfies $g = e * g = g * e$, $\forall g \in G$.*
4. *For any $g \in G$, there exists an inverse element $h \in G$ such that $g * h = h * g = e$.*

It is important to note that one of the properties present in most commonly used groups is not necessarily present in all groups: commutativity. For example, in the case of the set of real numbers $\mathbb{R}$, commutativity is present for the sum operation. However, in the case of a Rubik's Cube Group, commutativity does not hold: applying the moves `R U` is not the same as doing `U R`. In other words, there may exist elements g_1 and g_2 of G such that $g_1 * g_2 \neq g_2 * g_1$ [4].

Definition 2 (Group Order)**.** *The **order** of a group G, denoted $|G|$, is the number of elements in G. A group is a **finite group** if $|G| < \infty$.*

Lemma 1 (Uniqueness of Identity and Inverse)**.** *In a group $(G, *)$:*

1. *There is exactly one identity element. Suppose there are two identity elements e_1 and e_2. Then $e_1 = e_1 * e_2 = e_2$, proving that the identity is unique.*
2. *Each $g \in G$ has exactly one inverse. Assume h_1 and h_2 are two inverses of g. Then $h_1 = h_1 * (g * h_2) = (h_1 * g) * h_2 = e * h_2 = h_2$, showing that the inverse is unique.*

Definition 3 (Subgroup and Cyclic Group)**.** *Let H be a subset of a group G. If H is a group with the same operation as G, then H is a **subgroup** of G.*

*A group G is a **cyclic group** if there is some element g in G such that $G = \{g^n | n \in \mathbb{Z}\}$. The element g is a **generator** of the group G, denoted $G = \langle g \rangle$. The group C_n denotes the cyclic group of order n.*

Definition 4 (Permutation and Related Concepts)**.** *A **permutation** of a set G is an injective and surjective function from G to itself. The **permutation group** of the set S is the set of all permutations of S that form a group under composition.*

*The permutation group on a set of n elements, denoted S_n, is called the **symmetric group**.*

*A **cycle** is a permutation of the elements in a set $X = \{1, 2, 3, ..., n\}$ such that $x_1 \mapsto x_2 \mapsto ... \mapsto x_1$ where $x_i \in X$.*

*A cycle $(x_1 x_2 ... x_k)$ is called a cycle of **length** k. A permutation that can be expressed as a cycle of length 2 is called a **2-cycle** or **transposition**. A permutation is **even** if it can be expressed as an even number of 2-cycles, and **odd** if it can be expressed as an odd number of 2-cycles.*

These concepts can now be applied to the Rubik's Cube as a group. A key aspect of permutation groups is the operation of composition. If σ and τ are two permutations, their composition $\sigma \circ \tau$ means applying τ first, followed by σ.

Now, let $(G, *)$ denote the set of permutations of the Rubik's Cube. The Rubik's Cube has 54 facelets, so the set of permutations is a subset of S_{54}. The group operation $*$ is the composition of

permutations, meaning that two sequences of moves can be combined to form a new sequence. Thus, the set G is a subgroup of S_{54}, and the group G is referred to as the permutation group of the Rubik's Cube.

Theorem 2.1.1 (Rubik's Cube Group)**.** *The permutation group $G = \langle U, D, F, B, R, L \rangle \subset S_{54}$ is called the **Rubik's Cube Group**.*

Proof. We can show that the group properties are satisfied:

1. G is closed under $*$ since, for any two moves A and B, $A * B$ is also a move (or a sequence of moves).
2. Let e be the "empty" move. Then $A * e = e * A = A$ for any move A, so e serves as the identity element for $(G, *)$.
3. For any move A, its inverse A' exists such that $A * A' = A' * A = e$. For example, a clockwise quarter turn of a face has as its inverse the counterclockwise quarter turn of the same face.
4. Associativity holds because the composition of moves is always associative. Let A, B and C be moves, then $A * (B * C) = (A * B) * C$.

□

2.2 Application on the Cube

To better understand the structure of the Rubik's Cube Group, we need to introduce an additional concept:

Definition 5 (Direct Product)**.** *Let G_1 and G_2 be groups. Then the **direct product** of G_1 and G_2 is the set $G_1 \times G_2$ under the operation $(g_1, g_2) \cdot (g_1', g_2') = (g_1 g_1', g_2 g_2')$ for $g_1, g_1' \in G_1$ and $g_2, g_2' \in G_2$.*

Definition 6 (Semi-Direct Product and Wreath Product)**.** *Let G_1 and G_2 be subgroups. Then $A = G_1 \rtimes G_2$ is a **semi-direct product** if:*

1. *$A = G_1 G_2$.*
2. *$G_1 \cap G_2 = e_A$, where e_A is the identity element of A.*
3. *G_1 is a normal subgroup of A ($G_1 \triangleleft A$).*

*A **wreath product** is a special case of the semi-direct product. Let X be a finite set, G a group, and H a group acting on X. Fix a labeling of X, say $\{x_1, x_2, \ldots, x_t\}$, with $|X| = t$. Let G^t be the direct product of G with itself t times. The **wreath product** of G and H is $G^t \wr H = G^t \rtimes H$, where H acts on G^t by permuting the components of G^t according to its action on X.*

There are two different classifications of the Rubik's Cube Group: the Legal Rubik's Cube Group and the Illegal Rubik's Cube Group. The difference between the two is that the Illegal Rubik's Cube Group allows the solver to take the cube apart and rearrange the facelets. As expected, the Rubik's Cube Group is a subset of the Illegal Rubik's Cube Group.

Now, not all of the permutations of S_{54} are possible on the Rubik's Cube. Any valid permutation on the cube will send corner facelets to corner positions and edge facelets to edge positions. Any other permutations will not be physically possible on the cube. Hence, G is only a subset of S_{54} and is not isomorphic to the full permutation group.

Lemma 2 (Corner and Edge Groups)**.** *The position of all the corner facelets on the cube can be described by the group $C_3^8 \wr S_8$. The position of all the edge facelets on the cube can be described by the group $C_2^{12} \wr S_{12}$.*

This lemma arises from the definition of the wreath product and the nature of the Rubik's Cube. Each corner can be characterized by its position and the cyclic orientation of its three facelets, and each edge by its position and the cyclic orientation of its two facelets.

Lemma 3 (Illegal Rubik's Cube Group)**.** *The illegal Rubik's Cube Group is $I = (C_2^{12} \wr S_{12}) \times (C_3^8 \wr S_8)$.*

Proof. This follows from Lemma 2 and the definition of the direct product. □

With these definitions and lemmas in place and the knowledge given by Section 1.2, we can now state the two fundamental theorems of cube theory:

Theorem 2.2.1 (First Fundamental Theorem of Cube Theory)**.** *Let $v \in C_3^8$, $r \in S_8$, $w \in C_2^{12}$, and $s \in S_{12}$. The 4-tuple (v, r, w, s) corresponds to a possible arrangement (position) of the cube if and only if:*

1. *$sgn(r) = sgn(s)$ (equal parity of permutations).*
2. *$v_1 + v_2 + v_3 + \cdots + v_8 = 0 \pmod 3$ (conservation of the total number of twists).*
3. *$w_1 + w_2 + w_3 + \cdots + w_{12} = 0 \pmod 2$ (conservation of the total number of flips).*

Theorem 2.2.2 (Second Fundamental Theorem of Cube Theory)**.** *An operation of the cube is possible if and only if the following are satisfied:*

1. *The total number of edge and corner cycles of even length is even.*
2. *The number of corner cycles twisted clockwise is equal to the number of corner cycles twisted anti-clockwise (up to modulo 3).*
3. *There is an even number of reorienting edge cycles.*

With these two fundamental theorems of cube theory, any possible position and operation on the Rubik's Cube can be defined. Also, the theorems eliminate the physically impossible arrangements and moves from the group. The demonstrations of the theorems and lemmas can be found in the linked article [5].

2.3 Cube permutations with Groups

Thanks to the two theorems there are the right conditions to mathematically confirm the number of legal permutations found in the last chapter. Condition (2) of Theorem 2.2.1 states that the group determines the position of the corners and once 7 corners are arranged the orientation of the last one is automatically determined by the given formula, because the facelets can assume only one configuration in order to satisfy the conservation of the total number of twists. Similarly, condition (3) determines the orientation of the edges: once 11 edges are positioned, the last one is already determined. Condition (2) reduces the group by a factor of C_3 and condition (3) by a factor of C_2.

Now, to obtain the Legal Rubik's Cube Group it is needed one other reduction. Condition (1) of Theorem 2.2.2 states that the number of even permutations is equal to the odd permutations, so the group must be reduced by a factor of C_2.

It is possible to determine the number of permutations.

Theorem 2.3.1. *Let I be the illegal Rubik's Cube Group and G the Legal Rubik's Cube Group, with $G \subset I$. It can be expressed as:*

$$G = (C_3^7 \wr S_8) \times (C_2^{10} \wr S_{12}). \tag{2.1}$$

and $|G|$ the order of the group as:

$$|G| = |C_3^7||S_8||C_2^{10}||S_{12}| = 3^7 \cdot 8! \cdot 2^{10} \cdot 12! = 4.3 \times 10^{19} \tag{2.2}$$

The two fundamental theorems provide a framework that will be useful in the next chapters as properties for the development of different solving methods and heuristics. A mathematical representation of the cube gives a better understanding of the problem and a way to find a general approach to the puzzle itself. In the scientific field, the Rubik's Cube is often used as a representation of problems that may initially seem completely unrelated. However, through group theory it is frequently possible to reduce their complexity and solve different problems with a similar strategy.

3 Solving Methods

Following the characterization of the cube structure and permutations, we can now address the central problem: how to solve the puzzle. When Ernő Rubik first conceptualized the cube, he needed over a month to develop a reliable strategy for transitioning it from a random state to a solved one. For the average individual, solving the cube demands problem-solving skills and the ability to decompose the problem into simpler components. The use of memorized sequences, commonly referred to as **algorithms**, can significantly simplify this process.

3.1 Human solving approach

Typically, an individual encountering the problem of solving the cube for the first time adopts a simple, yet often ineffective strategy: attempting to construct one face and then proceeding face by face. This approach can lead to frustration as it relies on a two-dimensional strategy for a puzzle with a three-dimensional nature. The average person usually succeeds in building the first face but often abandons the attempt when the problem becomes more complex, as preserving the progress already made becomes challenging.

Over time, two main paths emerge in the learning curve of the average person: either abandoning the attempt or seeking guidance from external resources to find a comprehensible method that provides step-by-step instructions to achieve the solved state.

Throughout the years, mathematicians and speedcubers have developed methods that are comprehensible to the human mind and can be learned by most individuals within a few hours. The fundamental concept underlying most of these methods is to solve the majority of pieces intuitively (following a preset schema) and then utilize a set of algorithms to manipulate the remaining pieces.

The most widely recognized method is the **CFOP method**, also known as the Fridrich Method, invented by Czech engineering professor Jessica Fridrich [6]. This method consists of four steps: Cross, First two layers (F2L), Orient Last Layer (OLL), and Permute Last Layer (PLL). This approach is similar to the "face-by-face" strategy but focuses on the layer structure of the cube. The average move count for each step is as follows: 7 for Cross, 21 for F2L, 9 for OLL, and 12 for PLL, totaling 56 moves. The method is intuition-based for the first two steps (although some experienced speedcubers employ algorithmic optimizations in these steps) and relies on a set of algorithms for the last two. OLL comprises 57 different cases, while PLL involves 21 cases. With sufficient practice and analysis, this method can enable a speedsolver to achieve average solving times well under 10 seconds. For the average person, simplified versions of CFOP exist, allowing for cube solution with just a few algorithms.

The cubing community has developed various other methods that exploit different properties of the cube. These include **Roux** and **Petrus**, which focus on building blocks instead of layers, and **ZZ**, which incorporates the concept of edge orientation. Additionally, methods for blindfolded solving employ a wider range of intuitive algorithms called **commutators** (a concept directly derived from mathematics) to solve one piece at a time while preserving the rest of the cube intact.

The human mind requires patterns to simplify problems, which often leads to longer solutions compared to the optimal ones. World-class speedcubers average around 5 seconds per solve and continuously strive to improve their techniques by learning new algorithms, enhancing efficiency, and practicing fingertricks [7].

When the objective is to minimize the number of moves rather than the time, the World Cube Association hosts a specific event called the **Fewest Moves Challenge (FMC)**. In this event, competitors must find the shortest possible solution within a one-hour time limit. At high levels, the techniques employed in this event differ significantly from those used during speedsolving. World-class FMC solvers can average around 21 moves, which is just 3 moves away from the median optimal solution of 18 moves.

3.2 Computer solving approach

As one might expect, computers hold a significant advantage over human analysis in solving the Rubik's Cube, similar to other games like chess. They can outscore human abilities by a large margin, primarily through brute-force methods. A computer can apply a human-type method in fractions of seconds and find multiple solutions, leading to better move counts.

Before exploring the problem of solving the Rubik's Cube, it is essential to establish a good scrambling algorithm. A perfect scrambling algorithm takes a random permutation from the Rubik's Cube Group and generates a sequence of moves that leads to that state. Notably, taking the cube from the solved state to another fixed state is as challenging as performing the inverse operation, given that one can find a solution for that state and invert it (similarly to an inverse maze strategy, a concept we will explore in subsequent chapters).

A simpler approach for scrambling could involve generating a sequence of random moves, but this can lead to inconsistencies: it is not always clear how many moves are sufficient to achieve a fairly scrambled state. Even if we determined a suitable number, it could lead to parity problems if we apply the same number of moves for every scramble. Obviously, this scrambling method must have a way to ensure that moves do not cancel each other out.

An alternative method to evenly scramble a cube without reverse solving involves randomly permuting corners and edges and then implementing a routine that fixes parities to ensure a legal state. The routine that fixes parity also needs to be randomized to achieve an uniform scrambling algorithm. This approach requires selecting a random corner to twist, a random edge to flip, and a random pair of corners or edges to swap (if required).

```
cube_t scramble(cube_t c){
    random_permutation(c.corners);
    random_permutation(c.edges);
    fix_parity(c);
    return c;
}
```

The solving methods for computers can be divided into two main categories: sub-optimal solvers and optimal solvers.

3.2.1 Sub-optimal solvers

A sub-optimal solver typically follows a method that either solves pieces or relies on structural properties of the cube that ensure certain states are closer to the solved state. One simple solver could strictly apply the CFOP method and find a human-like solution. As mentioned earlier, the length of the solution is not close to being optimal with this method, which is why some mathematicians have sought alternative strategies.

In the early 1980s, Morwen Thistlethwaite developed a clever method of solving the cube. He decided to split the problem into four sub-tasks, each of which is analogous to solving a simpler puzzle and can be accomplished using a lookup table.

Simple puzzles with a small order of permutations can be stored in a database containing the information to solve them. One can apply every single move to the puzzle and proceed recursively until all permutations have been visited and store the distance from the solved state of each one in a table. Solving these puzzles becomes quite simple after building the table: one just has to apply a move to the cube and determine which state brings it closer to the solved state by performing a lookup in the table. This approach is not feasible for the standard Rubik's Cube due to its vast number of permutations, which cannot be compressed sufficiently to be useful.

To create his method, Thistlethwaite used a sequence of nested groups. The method takes advantage of successive restricting groups:

$$G_0 = \langle \texttt{U}, \texttt{D}, \texttt{R}, \texttt{L}, \texttt{F}, \texttt{B} \rangle$$
$$G_1 = \langle \texttt{U}, \texttt{D}, \texttt{R2}, \texttt{L2}, \texttt{F}, \texttt{B} \rangle$$
$$G_2 = \langle \texttt{U}, \texttt{D}, \texttt{R2}, \texttt{L2}, \texttt{F2}, \texttt{B2} \rangle$$
$$G_3 = \langle \texttt{U2}, \texttt{D2}, \texttt{R2}, \texttt{L2}, \texttt{F2}, \texttt{B2} \rangle$$
$$G_4 = I$$

This method differs from a human-like approach because it does not rely on solving pieces but progresses in the solving state through move constraints. The idea is to start from the scrambled state and apply moves that restrict the cube to a simpler group. For example, the first stage requires orienting edges on one axis, which can be accomplished by performing a lookup on a table with $2^{11} = 2048$ configurations, a relatively small number that can be generated quite easily. For each group $G_i \subset G_{i-1}$, and each step reduces the number of allowed moves excluding quarter turns on one axis at time: $18 \to 14 \to 10 \to 6 \to 0$. This method uses lookup tables to iterate through the states and find the solution. Thistlethwaite's algorithm has a worst-case scenario of $7 + 10 + 13 + 15 = 45$ moves and an average case of $4.6 + 7.8 + 8.8 + 10.1 = 31.3$ moves [8].

3.2.2 Optimal solvers

The problem of finding an optimal solution for a randomly scrambled state of the cube is closely related to the problem of determining "God's Number", which represents the maximum number of moves required to solve a cube optimally. We can visualize the states of the cube as a graph: the nodes represent the states, and the edges represent the moves required to transition from one state to another.

God's number becomes the diameter of this graph. This can be confirmed by the fact that the graph is symmetric for every node, allowing us to place the solved state at any node and maintain symmetry while finding the diameter.

In 1992, Kociemba introduced a two-phase algorithm that could rapidly find near-optimal solutions using only one megabyte of memory. In 1995, it was proven that the lower bound is 20 by demonstrating that the "superflip" state requires 20 moves to solve optimally. In 1997, Korf introduced the first optimal solver that took approximately a day to find an optimal solution [9].

In subsequent years, the upper bound was gradually lowered until the upper and lower bounds converged at 20 moves. The work done by a team comprising Rokicki, Kociemba, Davidson, and Detridge determined the God's number using a supercomputer and a highly optimized solving program [10].

Optimal solvers are based on the concept of **IDDFS** (Iterative Deepening Depth First Search), which combines two algorithms: **DFS** (Depth First Search) and **BFS** (Breadth First Search). DFS is a recursive algorithm that explores as deeply as possible along each branch before backtracking. BFS, in contrast, visits all nodes at the same level before proceeding deeper. IDDFS combines these approaches: it starts with a depth of 1 and applies DFS, then increases the depth and applies another DFS until it finds the solution. This algorithm is employed in optimal solvers to find the optimal solution in a reasonable timeframe and with a manageable amount of memory.

Modern optimal solvers are based on the **IDA*** (Iterative Deepening A*) algorithm, which combines IDDFS with the A* algorithm. The A* algorithm is a graph traversal algorithm that uses a heuristic to find the optimal path between nodes. The most widely used programs today are **Kociemba's Cube Explorer** for simple optimal solution searching [11], **Sebastiano Tronto's Nissy** for FMC analysis [12], **Thomas Rokicki's Nxopt**, which is considered the most advanced and fastest optimal solver available [13] and **Andrew Skalski's Vcube** which is a reimplementation of Nxopt and the solver which we consider in the benchmarking comparison [14].

The iterative process of the IDA* search and the implementation adopted in the **H48** solver will be explained in detail in subsequent chapters.

4 Data Encoding and Manipulation

This chapter examines the encoding of data in the context of the Rubik's Cube. We will first discuss the representation of the cube in memory as a data structure, followed by an analysis of the actions that can be applied to this structure to manipulate the cube, both for executing moves and creating particular cases.

4.1 Data Memorization

Multiple methods exist for storing Rubik's Cube information in memory. Some representations are more intuitive, while others are more space-efficient or computationally efficient.

4.1.1 Facelets Representation

The Rubik's Cube can be conceptualized as six 3x3 grids, one for each face, resulting in a three-dimensional vector of 54 elements representing the facelets. Each element in the vector is a character value representing the color of the facelet. While this is the most intuitive way to picture the cube, it can lead to complex code for cube manipulation.

Applying a move to the cube necessitates updating the corresponding facelets in the array. This can be achieved by cycling the facelets, but it is inefficient as each move affects 21 facelets (nine on the face being moved and 12 on the adjacent faces). This structure relies on a two-dimensional representation of the cube, resulting in more complex and slower implementations of move composition functions and properties.

Given that the color is a value between 0 and 5, it can be represented using only 3 bits. Consequently, the space required for one cube is $3 \cdot 54 = 162$ bits. Further space reduction is possible by excluding certain facelet information, but this would necessitate an algorithm to infer the remaining facelets from the stored information. The operation of applying a move can be conceptualized as a permutation, as described in Chapter 2.

4.1.2 Cubie Representation

The cube is inherently a three-dimensional object, and the cubie representation capitalizes on this by focusing on edges and corners. This representation comprises two vectors: one with 8 elements for the corners and another with 12 elements for the edges.

The elements in these vectors are more complex than those in the facelets representation. A corner piece is represented by a notation such as UFR0, indicating the white, green, and red corner piece in solved orientation. An edge piece is denoted for example as UF1, representing the white and green edge piece in flipped orientation. The orientation is encoded in an integer between 0 and 2 for corners, and 0 or 1 for edges.

This structure facilitates more efficient computation as the operations are simpler to implement and execute more rapidly. For this representation, we allocate 3 bits for the corner type and 2 bits for its orientation, 4 bits for edges, and 1 bit for edge orientation. Consequently, the space required for one cube is $5 \cdot 8 + 5 \cdot 12 = 100$ bits.

4.1.3 Coordinate Representation

The coordinate representation offers a more efficient approach for specific types of computation. This method associates each possible cube position with a natural number.

The structure is encoded in four coordinates: Corner Orientation (CO): a number from 0 to 2186 ($|CO| = 3^7 - 1$), Edge Orientation (EO): a number from 0 to 2047 ($|EO| = 2^{11} - 1$), Corner Permutation (CP): a number from 0 to 40319 ($8! - 1$), Edge Permutation (EP): a number from 0 to 479001599 ($12! - 1$).

The memory space required for one cube in this representation is:

$$\lceil \log_2(|CO|) \rceil + \lceil \log_2(|EO|) \rceil + \lceil \log_2(|CP|) \rceil + \lceil \log_2(|EP|) \rceil = 12 + 11 + 16 + 29 = 68 \text{ bits} \quad (4.1)$$

While this level of abstraction is beneficial for computing pruning tables, it necessitates the use of transition tables to apply moves to the cube. This process can be relatively slow on modern computers due to the disparity between RAM access and register or cache operations [11].

4.1.4 H48 Implementation

In the implementation of **H48**, the cube is represented using the cubie representation. This choice was made due to the simplicity of the operations and the ease of implementation. The cube is stored in memory as a 20-byte struct, with a vector of 8 bytes for the corners and 12 bytes for the edges. Each byte is divided into two parts: the first 4 bits represent the piece position, and the remaining 4 bits represent the orientation. This structure is larger than the original cubie representation and contains some unused bits; however, it is more intuitive and easier to manipulate using bitwise operations. The implementation is as follows:

```
typedef struct {
    uint8_t corners[8];
    uint8_t edges[12];
} cube_t;
```

This structure differs when we consider architecture-oriented implementation. In the **H48** core code [15], we can find two implementations: one for Intel x86 architecture that leverages SIMD intrinsics through the AVX2 library, and one for ARM architecture that utilizes NEON intrinsics:

AVX2

```
typedef __m256i cube_t;
```

NEON

```
typedef struct {
    uint8x8_t corners;
    uint8x16_t edges;
} cube_t;
```

4.2 String Representations of Cube Data Structure

H48 employs three different string representation for cube states: H48, LST, and B32. These formats assume a fixed-centered cube in a specific orientation and leverage the cubie representation. The cube state is encoded as a string of characters, with each element representing a specific piece of the cube [16].

4.2.1 H48 Format

The H48 format represents each edge with two letters denoting its position and a number indicating orientation (0 for oriented, 1 for misoriented). Similarly, each corner is represented by three letters for position and a number for orientation (0 for oriented, 1 for clockwise rotation, 2 for counterclockwise rotation).

```
UF0 UB0 DB0 DF0 UR0 UL0 DL0 DR0 FR0 FL0 BL0 BR0 UFR0 UBL0 DFL0 DBR0 UFL0 UBR0 DFR0 DBL0 // Solved
FL1 UB0 DB0 FR1 UR0 UL0 DL0 DR0 UF1 DF1 BL0 BR0 UFL1 UBL0 DFR1 DBR0 DFL2 UBR0 UFR2 DBL0 // F move
```

4.2.2 LST Format

The LST format represents the cube as a list of unsigned 8-bit integers. Each piece is encoded by a single integer, where the 4 least-significant bits determine the piece expected position, and the 4 most-significant bits indicate orientation.

```
0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 // Solved
36, 1, 38, 3, 66, 5, 64, 7, 25, 1, 2, 24, 4, 5, 6, 7, 16, 19, 10, 11 // F move
```

4.2.3 B32 Format

The B32 format offers a compact representation using base-32 encoding. Each piece is encoded as a number between 0 and 31, then converted to an uppercase letter (0-26) or a lowercase letter (27-31). Edges and corners are separated by a single equals sign. A cube state in B32 format is consistently 21 characters long (22 including the terminating null character).

```
ABCDEFGH=ABCDEFGHIJKL // Solved
MBODSFQH=ZBCYEFGHQTKL // F move
```

4.3 ARM NEON and SIMD Processes

ARM NEON is a powerful extension for ARM processors designed to accelerate data processing, particularly in applications requiring high-performance computing. It provides Single Instruction, Multiple Data (SIMD) capabilities, enabling the execution of a single operation on multiple data points in parallel.

The evolution of ARM architecture has introduced several SIMD optimization techniques. The ARMv6 architecture introduced a small set of SIMD instructions that operate on multiple 16-bit or 8-bit values packed into standard 32-bit ARM general-purpose registers. These instructions allowed certain operations to execute two or four times as fast without adding additional computation units. NEON technology extends the ARM processor pipeline and utilizes the ARM core registers for memory addressing.

ARM NEON, introduced in the ARMv7 architecture, provides a significantly more powerful SIMD instruction set. SIMD through NEON operates by grouping data into vectors, typically 64-bit or 128-bit, and applying operations across all elements of the vector simultaneously, rather than iterating through each element individually. For example, instead of performing an addition on each element of an array sequentially, SIMD allows the processor to perform additions on multiple elements simultaneously, resulting in significant performance improvements. This is particularly advantageous for ARM NEON, as it can work with both integer and floating-point data types, enabling efficient processing across a wide range of applications.

We will not consider vector floating-point operations (VFP), as the cube is represented using integers. The NEON unit is fully integrated into the processor and shares the processor resources for integer operations, loop control, and caching. This significantly reduces the area and power cost compared to a hardware accelerator. It also employs a simpler programming model, as the NEON unit uses the same address space as the application. The components of the NEON unit include: NEON register file, NEON integer execute pipeline, NEON single-precision floating-point execute pipeline, and NEON load/store and permute pipeline [17].

In the context of Rubik's Cube manipulation, using NEON intrinsics helps optimize operations such as permuting cube pieces and applying moves. By leveraging SIMD, multiple cubies can be manipulated in parallel, significantly enhancing the performance of functions like move composition and comparison. Intrinsic functions and data types provide access to low-level instructions, offering almost as much control as writing assembly code, but with the convenience of C syntax. The allocation of registers is left to the compiler, allowing the programmer to focus on the logic of the program. The NEON intrinsics are defined in the `arm_neon.h` header file, which contains a wide range of functions for performing SIMD operations on ARM processors.

The contents of the NEON registers are vectors of elements of the same data type. A vector is divided into lanes, and each lane contains a data value called an element. The number of lanes in a vector depends on the data type. A 128-bit NEON vector can contain the following element sizes:

- Sixteen 8-bit elements (operand suffix .16B, where B indicates byte)
- Eight 16-bit elements (operand suffix .8H, where H indicates halfword)
- Four 32-bit elements (operand suffix .4S, where S indicates word)
- Two 64-bit elements (operand suffix .2D, where D indicates doubleword)

A 64-bit NEON vector can contain the following element sizes (with the upper 64 bits of the 128-bit register cleared to zero):

- Eight 8-bit elements (operand suffix .8B, where B indicates byte)

- Four 16-bit elements (operand suffix .4H, where H indicates halfword)
- Two 32-bit elements (operand suffix .2S, where S indicates word)

The naming of the functions is consistent with the ARM architecture, with the prefix v indicating vector operations. For example, `vaddq_u8()` performs a vector addition on 8-bit unsigned integers, while `vst1_u8()` stores 8-bit values to memory.

Intrinsics that use the 'q' suffix usually operate on Q registers. Intrinsics without the 'q' suffix usually operate on D registers, but some of these intrinsics might use Q registers. For example, the function `vaddq_u8()` adds two 128-bit vectors of 8-bit integers, while `vadd_u8()` adds two 64-bit vectors of 8-bit integers [18].

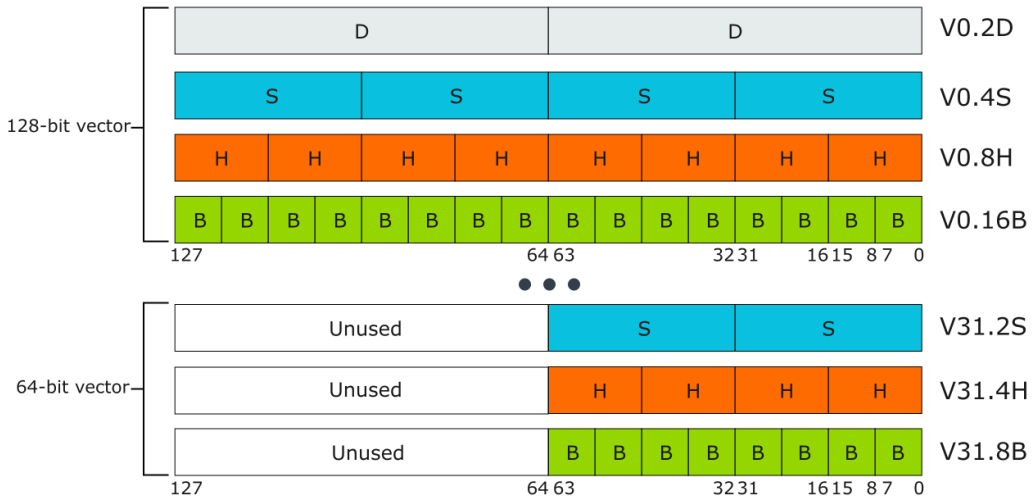

Figure 4.1: NEON registers

4.4 Core Operations

Having established the data structure for the cube, we can now examine the core operations that can be applied to it. The implementation shown is for the NEON architecture. The primary operation is the application of moves to a cube, the foundamental functions for doing that are: `pieces()` which converts an array of pieces into a new cube, `equal()` which checks if two cubes are equal, and `compose()` which composes two cubes and is used to apply moves to a cube.

4.4.1 Pieces function

```
STATIC void
pieces(cube_t *cube, uint8_t c[static 8], uint8_t e[static 12]){
    // 8 bytes of the corner vector are copied from the c array
    vst1_u8(c, cube->corner);

    // 12 bytes of the edge vector are copied from the e array (8+4 bytes)
    vst1_u8(e, vget_low_u8(cube->edge));
    vst1_lane_u32((uint32_t *)(e + 8),
                  vreinterpret_u32_u8(vget_high_u8(cube->edge)), 0);
}
```

We can see that the information about the cube is stored in chunks of 8 bytes, which are then copied to the cube structure. This is done using the `vst1_u8()` function, which stores the 8-bit values in the first argument to the memory address specified in the second argument. The `vget_low_u8()` function extracts the lower 8 bytes from the cube structure, while `vget_high_u8()` extracts the higher 8 bytes. The `vst1_lane_u32()` function stores the 32-bit value in the first argument to the memory address specified in the second argument, using the lane specified in the third argument [19].

4.4.2 Equal function

```
STATIC_INLINE bool
equal(cube_t c1, cube_t c2){
    uint8x8_t cmp_corner;
    uint8x16_t cmp_edge;
    uint64x2_t cmp_corner_u64, cmp_edge_u64, cmp_result;

    // compare the corner vectors and the edge vectors
    cmp_corner = vceq_u8(c1.corner, c2.corner);
    cmp_edge = vceqq_u8(c1.edge, c2.edge);

    // convert the comparison vectors to 64-bit vectors and combine them
    cmp_corner_u64 = vreinterpretq_u64_u8(vcombine_u64(cmp_corner, cmp_corner));
    cmp_edge_u64 = vreinterpretq_u64_u8(cmp_edge);
    cmp_result = vandq_u64(cmp_corner_u64, cmp_edge_u64);

    // check if all the bits are set
    return vgetq_lane_u64(cmp_result, 0) == ~0ULL &&
           vgetq_lane_u64(cmp_result, 1) == ~0ULL;
}
```

The `equal()` function compares two cubes and returns a boolean value indicating whether they are equal. This is done by comparing the corner and edge vectors of the two cubes using the `vceqq_u8()` function, which compares the 8-bit values in the two vectors. The comparison vectors are then converted to 64-bit vectors using the `vreinterpretq_u64_u8()` function. The comparison vectors are combined using the `vandq_u64()` function, which performs a bitwise AND operation on the two vectors. The function then checks if all the bits in the combined vector are set to 1, indicating that the two cubes are equal [19].

4.4.3 Compose function

```
STATIC_INLINE cube_t
compose(cube_t c1, cube_t c2){
    cube_t ret = {0};
    // Masks (initiated from the macro definitions)
    uint8x16_t p_bits_e, eo_bit;
    uint8x8_t p_bits_c, cobits, cobits, twist_cw;

    // EDGES, find the index and permutation
    uint8x16_t p0_e = vandq_u8(c2.edge, p_bits_e);
    uint8x16_t p1_e = vqtbl1q_u8(c1.edge, p0_e);

    // Calculate the orientation through XOR
    uint8x16_t orien = vandq_u8(veorq_u8(c2.edge, p1_e), eo_bit);

    // Combine the results
    uint8x16_t edges = vorrq_u8(vandq_u8(p1_e, p_bits_e), orien);

    // Mask to clear the last 32 bits of the result
    uint8x16_t mask_last_32 = vsetq_lane_u32(0, vreinterpretq_u32_u8(edges), 3);
    edges = vreinterpretq_u8_u32(mask_last_32);

    // CORNERS, find the index and permutation
    uint8x8_t p0_c = vand_u8(c2.corner, p_bits_c);
    uint8x8_t p1_c = vtbl1_u8(c1.corner, p0_c);
```

```
    // Calculate the orientation
    uint8x16_t aux = vadd_u8(vand_u8(c2.corner, cobits), vand_u8(p1_c, cobits));
    uint8x16_t auy = vshr_n_u8(vadd_u8(aux, twist_cw), 2);
    orien = vand_u8(vaddq_u8(aux, auy), cobits2);

    // Combine the results
    uint8x8_t corners = vorr_u8(vand_u8(p1_c, p_bits_c), orien);

    ret.corner = corners;
    ret.edge = edges;
    return ret;
}
```

The `compose()` function applies the edge and corner permutations and orientations of the second cube to the first one. The function calculates the edge permutation and orientation by extracting the relevant bits from the two cubes and performing bitwise operations on them. It combines the results to obtain the final edge vector. Subsequently, the function calculates the corner permutation and orientation in a similar manner and combines the results to obtain the final corner vector. In the end, the function does some masking to clear the unused bits of the result and returns the final cube.

This property of applying moves through composition is crucial for the implementation of the Rubik's Cube solver. Moves are applied by composing the specified cube with some macros which represent cubes with one move applied.

There is a strict correlation between a state and the moves from the solved state to that state, so the composition of two states is equivalent to the composition of the moves from the solved state to those states. This is a fundamental property of the Rubik's Cube Group, as described in Chapter 2.

The `compose()` function allows us to visualize the cube solving procedure in its entirety. A useful way to conceptualize the cube solving process is through graph representation. We initially have a solved cube and a sequence of moves to scramble it. This sequence is correlated with the state itself, and it is not necessary to have the sequence. We can construct the solution by applying moves to the cube until we reach the solved state. Another strategy used in Fewest Moves solving is to use premoves, which in composing terms is simply taking the move cube and composing it with the scrambled cube. With this approach, the final solution is as follows:

Let P be the premove cube, S the scrambled cube and M the move cube. The solution is given by the composition of the three cubes:

$$P * S * M = I \tag{4.2}$$

4.4.4 Additional Core Functions

The architecture implementation includes several other core functions, a notable example being the `inverse()` function. Given a cube C, this function returns the inverse cube C^{-1} such that $C * C^{-1} = I$. This function plays a crucial role in the process of finding optimal solutions.

Premoves can be conceptualized as an application of the inverse function, effectively applying moves to the inverted cube. One approach to solving the cube involves applying moves to the inverse cube until it reaches the solved state. The solution for the original cube can then be obtained by inverting the sequence of moves applied to the inverse cube [20].

5 IDA* and Pruning

This chapter explores the IDA* algorithm and its application in solving the Rubik's Cube. We will discuss how pruning techniques can be employed to reduce the number of states that need to be explored. Additionally, we will examine how multithreading can be utilized to enhance the performance of the IDA* algorithm.

5.1 DFS and IDDFS

The Depth-First Search (DFS) is a fundamental algorithm that explores a graph by traversing as deeply as possible along each branch before backtracking. DFS is often preferred over Breadth-First Search (BFS) for large graphs because the memory requirements of BFS become prohibitive at deeper levels. The DFS algorithm can be implemented using either a stack or recursion, and it only needs to maintain the current path from the root to the current node. However, applying pure DFS to the Rubik's Cube Graph can be problematic due to the vast number of states that need to be explored and the presence of cycles within the graph.

A potential improvement is to combine the DFS and BFS approaches in an Iterative Deepening Depth-First Search (IDDFS) algorithm. The IDDFS algorithm employs a depth limit to control the scope of the search. This depth limit is incrementally increased until a solution is found. While IDDFS does revisit the same states multiple times as the search is restarted at each iteration, this cost is offset by the pruning that can occur at each iteration. The repeated executing parts are the fastest, because the number of positions to check grows exponentially with the depth, so asymptotically repeating the first phases is negligible.

5.2 Move Redundancy Pruning

The Rubik's Cube Graph presents a significant challenge due to the exponential growth in the number of states explored at each level. Starting from 18 moves at level 1, the number of states explored expands to $18^2 = 324$ at level 2, $18^3 = 5832$ at level 3, and so forth. However, some of these moves can be pruned as they are redundant with previous ones. For example, if the previous move was `R`, a subsequent `R` move is redundant because it is equivalent to `R2`.

To address this, we introduce a function that identifies redundant moves and prunes them accordingly. There are two types of redundant moves: the first case occurs when we have a previous move and attempt to apply a move on the same face; the second case arises when the current move is on the same axis and opposite face as the second to last move, necessitating the pruning of both on that axis. For instance, given a partial solution $S =$ `R U2 R L`, the set of allowed moves M_A is:

$$M_A = \{\texttt{U}, \texttt{U2}, \texttt{U'}, \texttt{D}, \texttt{D2}, \texttt{D'}, \texttt{F}, \texttt{F2}, \texttt{F'}, \texttt{B}, \texttt{B2}, \texttt{B'}\} \tag{5.1}$$

In the **H48** implementation we can find a function that checks for move redundancy. The function `allowednextmove_h48()` returns an integer value representing allowed moves. The function is implemented in the following way:

```
STATIC uint32_t
allowednextmove_h48(uint8_t *moves, uint8_t n, uint8_t h48branch){
    uint32_t result = MM_ALLMOVES;
    // Disable half turns in specific branching cases
    if (h48branch & MM_NORMALBRANCH)
        result &= MM_NOHALFTURNS;
    if (n < 1) return result;
```

```
    // Disable moves on the same face
    uint8_t base1 = movebase(moves[n-1]);
    uint8_t axis1 = moveaxis(moves[n-1]);
    result = disable_moves(result, base1 * 3);

    // Disable moves on the same axis for half of the moves to avoid redundancy
    if (base1 % 2)
        result = disable_moves(result, (base1 - 1) * 3);

    if (n == 1) return result;

    // Disable moves on the same axis when the second previous was on the same axis
    uint8_t base2 = movebase(moves[n-2]);
    uint8_t axis2 = moveaxis(moves[n-2]);

    if(axis1 == axis2)
        result = disable_moves(result, base2 * 3);

    return result;
}
```

The function `allowednextmove_h48()` accepts as input parameters the previous moves, the number of moves, and the branching value. The return value is initialized with all moves enabled. Initially, the function verifies if the branching value is enabled, and if so, it disables half turns (the branching value is presented in the next section). The function then performs these actions: it disables the moves on the same face of the previous move, it disables the moves on the same axis of the previous if the second previous move was on the same axis, and it applies and additional pruning for axis redundancy. For instance, given a partial solution $S =$ `R U2 L`, the set of allowed moves M_A is identical to the previous example 5.1. The moveset `R` is excluded because the previous move was on the same axis. This eliminates cases of solutions that differ only by sequential moves on the same axis, such as `LR` and `RL` occurring simultaneously.

The function returns the set of allowed moves as a one-hot encoded value, where bit i is set if move i is allowed. Consequently, the number of moves checked for each level is: 18 states at level 1, 243 states at level 2, and 3240 states at level 3. This approach reduces the number of states explored at depth 3 by $3240/5832 = 55.6\%$.

5.3 IDA*

The IDA* algorithm is a variant of the IDDFS algorithm that employs heuristics to prune the search space. Heuristics estimate the cost of reaching the goal from the current state and are utilized to prune the search when the estimated cost of reaching the goal from the current state exceeds the bound given by the deepening factor.

Let d be the deepening factor, n the number of moves already applied, and h the heuristic value obtained from the pruning table. The IDA* algorithm prunes the search when:

$$n + h > d \tag{5.2}$$

The IDA* algorithm guarantees finding the optimal solution if the heuristic is admissible. An heuristics is admissible if it never overestimates the cost of reaching the goal from the current state. It is memory-efficient due to its use of depth-first search to explore the graph.

There are several approaches to creating heuristics, with the most commonly used methods taking a subset of pieces and calculating the cost of reaching the solved position through precalculated tables known as pruning tables [21].

5.3.1 H48 IDA* Implementation

The IDA* algorithm is implemented in **H48** using the `solve_h48()` function, which covers the iterative deepening part of the algorithm and calls the `solve_h48_dfs()` function to start the recursive search [22]. The function is implemented as follows:

```
STATIC int64_t
solve_h48(cube_t cube, int8_t minmoves, int8_t maxmoves, int8_t maxsolutions,
            uint64_t data_size, const void *data, uint64_t solutions_size,
            char *solutions, long long stats[static NISSY_SIZE_SOLVE_STATS]){
    _Atomic int64_t nsols = 0;
    dfsarg_solveh48_t arg;
    tableinfo_t info, fbinfo;

    // Read the pruning table information
    if(readtableinfo_n(data_size, data, 2, &info) != NISSY_OK)
        goto solve_h48_error_data;
    arg = (dfsarg_solveh48_t){ /* Initialization with the input state */ };

    // Some pruning table information checking is performed here

    // Cycle in the iterative deepening search
    for (arg.depth = minmoves;
         arg.depth <= maxmoves && nsols < maxsolutions;
         arg.depth++) {
        LOG("Found %" PRId64 " solutions, searching at depth %"
            PRId8 "\n", nsols, arg.depth);
        arg.nmoves = 0;
        arg.npremoves = 0;
        solve_h48_dfs(&arg);
    }
    **arg.nextsol = '\0';
    (*arg.nextsol)++;
    return nsols;

    stats[0] = arg.nodes_visited;
    stats[1] = arg.table_fallbacks;
    LOG("Nodes visited: %lld\nTable fallbacks: %lld\n",
        arg.nodes_visited, arg.table_fallbacks);

    solve_h48_error_data: LOG("solve_h48: error reading table\n");
        return NISSY_ERROR_DATA;
}
```

The `solve_h48()` function takes as input the current state, the minimum and maximum number of moves, the maximum number of solutions, the pruning table, the solutions printing string and the stats array. It first reads the pruning table information and initializes the arguments for the depth-first search. The function then iterates over the depth levels and calls the depth-first search function. It returns the number of solutions found.

The depth-first search occurs in child functions called recursively from the main `solve_h48()`. The depth-first search function `solve_h48_dfs()` is implemented as follows:

```
STATIC int64_t
solve_h48_dfs(dfsarg_solveh48_t *arg){
    dfsarg_solveh48_t nextarg;
    uint32_t allowed;
```

```
    int64_t ret = 0;
    uint8_t m;

    // Check if the maximum number of solutions has been reached
    if (*arg->nsols == arg->maxsolutions) return 0;

    // Check the cost function
    if (solve_h48_stop(arg)) return 0;

    // Check if the cube is solved
    if (issolved(arg->cube)) {
        if (arg->nmoves + arg->npremoves != arg->depth) return 0;
        solve_h48_appendsolution(arg);
        return 1;
    }

    // Recursively call the depth-first search
    nextarg = *arg;
    if(!(arg->nissbranch & MM_INVERSE)) {
        allowed = allowednextmove_h48(arg->moves, arg->nmoves, arg->nissbranch);
        for (m = 0; m < 18; m++) {
            if (allowed & (1 << m)) {
                nextarg.nmoves = arg->nmoves + 1;
                nextarg.moves[arg->nmoves] = m;
                nextarg.cube = move(arg->cube, m);
                nextarg.inverse = premove(arg->inverse, m);
                ret += solve_h48_dfs(&nextarg);
            }
        }
    } else { /* Equivalent for the inverse branch */ }

    arg->nodes_visited = nextarg.nodes_visited;
    arg->table_fallbacks = nextarg.table_fallbacks;
    return ret;
}
```

The function first checks if the maximum number of solutions has been reached and then evaluates the puzzle state to obtain the lower bound through the cost function `solve_h48_stop()`. It then checks if the current state is solved and if the depth is reached; if so, the function appends the solution to the solutions string. The function then initializes the next arguments for the depth-first search and iterates through the recursive branch selecting to add moves on normal or inverse based on the `arg->nissbranch` variable, which is changed by the cost function and is explained in the next section. It then iterates over the allowed moves and calls the depth-first search recursively. Finally, the function returns the number of solutions found.

5.4 Pruning Tables

A pruning table, or pattern database, associates a value to a cube position that represents a lower bound for the number of moves required to solve that position. To access this value, the cube must be converted into an index through coordinates. Coordinates are often derived from the cosets of a target subgroup. For example, considering the corner positions (equivalent to the 2x2 cube solution table), this group consists of $12! \cdot 2^{10}$ positions, and the set of cosets of this group has a size of $8! \cdot 3^7$.

The pruning table can be populated by exploring all positions starting from the solved state using a Breadth-First Search, then assigning the depth to each coordinate position in the table. Larger pruning tables contain more information, enabling faster solvers. The information in the table can be

stored using 4 bits per position to store the lower bound, or 2 bits with less information. While 1-bit storage is possible, it only provides information on whether the position requires more or less than a predetermined number of moves to be solved, which is generally not efficient enough.

Another effective method for reducing the size of a pruning table is by utilizing symmetries. For instance, all positions that are one quarter-turn away from the solved state are equivalent under cube rotation and optional mirroring transformation. Every position belongs to a set of up to 48 positions. Storing entries for symmetric positions is redundant, and it is possible to reduce the table size without losing information.

Most solvers do not take advantage of the full group of 48 symmetries but only 16 of them that fix the UD axis. However, they compensate for this by performing 3 pruning table lookups, one for each axis of the cube [8].

5.4.1 H48 Coordinates

The pruning tables used in the **H_{48}** solver leverage a specific target group that is invariant under all 48 symmetries. This group is defined as follows:

- Each corner is placed in its respective corner tetrad and oriented. The tetrads on a cube are 2: $\{\texttt{UFR}, \texttt{UBL}, \texttt{DFL}, \texttt{DRB}\}$ and $\{\texttt{UFL}, \texttt{UBR}, \texttt{DFR}, \texttt{DBL}\}$.
- Each edge is placed in its respective slice. The edge slices are 3: $M = \{\texttt{UF}, \texttt{UB}, \texttt{DB}, \texttt{DF}\}$, $E = \{\texttt{UR}, \texttt{UL}, \texttt{DL}, \texttt{DR}\}$, $S = \{\texttt{FR}, \texttt{FL}, \texttt{BL}, \texttt{BR}\}$.

The coordinate obtained from the target subgroup described above is the **h0** coordinate. With symmetry reduction, it has a total of $3393 \cdot C(12, 8) \cdot C(8, 4) \approx 1.18 \cdot 10^8$ values. This is relatively small for a pruning table, as with 2 bits per entry it would occupy less than 30MB of memory. To enhance its effectiveness, we add information with partial edge orientation. By modifying the target group to impose oriented edges, we obtain the **h11** coordinate, whose full size with symmetry-reduced corners is $3393 \cdot C(12, 8) \cdot C(8, 4) \cdot 2^{11} \approx 2.41 \cdot 10^{11}$ values. With 2 bits per entry, this table would occupy almost 60GB of memory, which is too large for both storage and loading into memory on most personal computers. To address this issue, we calculate intermediate tables that ignore the orientation of some edges, creating the coordinates from **h1** to **h10**. One approach to constructing smaller coordinates is to take the **h11** table and "forget" some of the edge orientation values.

Considering the 4-bit version (**k4**) and the 2-bit version (**k2**), the pruning values have the following meanings:

- **k4**: The value (between 0 and 15) can be directly read from the table.
- **k2**: The value can be between 0 and 3. If the base value is b, a pruning value can be used directly as a lower bound of b+1, b+2, b+3 respectively. A value of 0 could mean that the actual lower bound is anything between 0 and b. Since we cannot take b as the lower bound, we must read a value from another table to obtain the actual lower bound, this is called as table fallback.

5.4.2 Pruning enhancements

We can employ several techniques to improve our estimation using the pruning value. One effective method is to check the pruning table on the inverse position. If the normal estimate is insufficient to prune the branch, the inverse of the position can be evaluated to obtain a new estimate.

Another approach to enhance pruning is to reduce the branching factor in cases with tight bounds. We can leverage the fact that the **h0** coordinate is invariant under the subgroup $\langle \texttt{U2}, \texttt{D2}, \texttt{R2}, \texttt{L2}, \texttt{F2}, \texttt{B2} \rangle$. In the special case where the solution length is m, the current depth is d, and the pruning value for the inverse is h, with a strict bound of $d + p = m$, we can make an optimization. From the inverse perspective, the solution cannot end with a 180-degree move, which is the first next move in the normal branch. In this particular case, the `arg->nissbranch` variable is activated with the `MM_NORMALBRANCH` value. This value is passed to the `allowednextmove_h48()` function, which disables half turns. We apply the same technique for the inverse branch by activating the variable to the `MM_INVERSEBRANCH` value in the case which tight bounds are verified, and the next move is going to be selected on inverse and saved as a premove [23]. The tight bound technique is similar to the one used in Nxopt solver but adapted to the **H_{48}** group.

5.5 Multithreading

The performance of IDA* can be significantly improved by implementing a multithreaded version of the algorithm, known as PMIDA* (Parallel Multithreaded IDA*). PMIDA* utilizes multiple threads to explore the graph concurrently [24]. The threads are synchronized using a barrier that is triggered when all threads have completed exploring the current level. In the $\mathbf{H_{48}}$ solver, PMIDA* is implemented using the `pthread.h` library which provides the necessary functions for thread creation, synchronization, and management. The implementation can be found in the `solve_h48_multithread()` function [25].

A basic multithreaded approach could take the number of threads as input and divide the search space equally among them. The threads would then explore the graph in parallel and synchronize at the end of each level. We could apply the first move of the search with a breadth-first approach, resulting in 18 branches to explore. However, this approach is computationally inefficient as the search space is not evenly distributed, leading to some threads finishing earlier than others.

The multithreaded search in the $\mathbf{H_{48}}$ solver is structured using the thread pool approach. A thread pool is a group of threads created at the start of the application and reused to execute tasks. Tasks are placed in a queue, and threads retrieve and execute them. The tasks are created once by performing a breadth-first search at depth 2, generating 243 tasks that are placed into a provisional queue. Threads are then created and assigned the empty thread pool queue. At each iteration of the iterative deepening search, tasks are moved from the provisional queue to the thread pool queue. The threads, which poll the queue, are triggered and begin executing tasks. When the queue is empty, threads that have completed their tasks enter a wait state until all threads are finished. The next iteration then begins, and the process repeats until a solution is found [26].

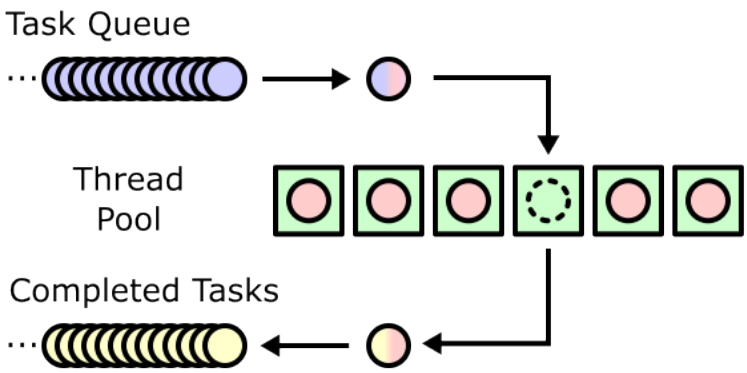

Figure 5.1: Thread Pool

Using a breadth-first search at depth 2 to create tasks is convenient as it generates a suitable number of tasks and ensures a good distribution of the search space. The number of tasks equals the number of possible moves (with `allowednextmove_h48()` pruning), as all 2-move sequences need to be considered as potential solution candidates. Threads do not interact during task execution, with only the queue, barrier, solutions counter, and solutions string being shared variables.

The performance gain is not linear with the number of threads due to the overhead of thread management and communication, as the use of mutexes for thread synchronization impacts performance. In the worst-case scenario, where the optimal solution begins with moves at the end of the queue, a bottleneck occurs in the last thread executing the final task. While significant time is saved by checking all tasks before the last one, the final task is executed by only one thread.

A potential improvement to the thread pool approach, which could prevent threads from waiting passively, is to implement a **work-stealing algorithm**. In this approach, threads that have completed their tasks can "steal" tasks from other threads if the queue is empty. This ensures that threads remain busy, potentially improving overall performance. The performance of the multithreaded solver version will be evaluated in the subsequent chapter.

6 Benchmark Results

This chapter presents the benchmarking results of the $\mathbf{H_{48}}$ solver implementations. The different types of benchmarks conducted are:

- **SIMD Optimization Benchmark:** Measures performance gains achieved through SIMD optimization by comparing architecture-optimized code with portable code.
- **Multithread Benchmark:** Evaluates algorithm performance across different thread counts.
- **Pruning Memory Benchmark:** Analyzes algorithm performance with varying pruning table sizes to determine optimal solutions.
- **Nodes Benchmark:** Measures the number of nodes visited by the algorithm at different depths. We also analyze the average time spent per node.
- **Performance Benchmark:** Compares execution times against another modern solver (VCube) implemented with similar-sized pruning tables.

It is important to note that the benchmarking was conducted on a working but preliminary version of the solver, which is still under development with room for performance improvements. Our research explored a new group configuration in the pruning tables. The idea was to take advantage of the symmetries and low memory accesses of this group while excluding other optimizations that cannot be applied to this group. This coordinate system represents an innovative approach never previously explored before.

The benchmarks focus primarily on the components I was most involved with: ARM NEON optimization, the IDA* algorithm, and the multithreaded implementation. The benchmarking was done on a set of different machines in order to compare the architecture performance, but they should not be taken into comparison with each other. The machines used for benchmarking are:

- **Apple M1 (ARM):** MacBook Air M1 (2020) with 8-core CPU (8 threads), 8GB RAM, macOS Sequoia 15.0
- **AMD Ryzen 7 7700 (x86):** Custom-built Desktop with 8-core CPU (16 threads), 32GB RAM, Debian 12.0
- **HPC Cluster (x86):** HPC UniTn Cluster with 64-core CPU (64 threads), 128GB RAM, CentOS 7.0

Due to inconsistent results observed on the HPC machine, likely caused by high system load, we performed multiple benchmark runs and used average values. Some results remain anomalous and do not correlate with our personal machine results. The pruning tables examined include h6k2 (2GB, suitable for average users), h8k2 (7GB, for powerful machines), and h11k2 (56GB, largest available). We focused exclusively on k2 tables as k4 tables are still under development. These table sizes also represent the maximum available memory on the machines used for benchmarking. The solver was mostly tested on 100 random positions (which are on average 18 optimal), with some additional tests on specific positions.

6.1 SIMD Optimization Benchmark

The SIMD optimization benchmark quantifies performance gains achieved through SIMD optimization by comparing architecture-optimized code with portable code. The results below represent solver execution on 100 random positions using a single thread:

Machine	Portable [s]	SIMD Optimized [s]	Gain
ARM M1 h6	57.8413	31.5337	83.43%
x86 DESK h6	43.2173	25.2266	71.32%
x86 DESK h8	15.8987	9.0389	75.89%
x86 HPC h6	54.8843	41.5590	32.63%
x86 HPC h11	6.5420	4.5995	42.23%

Figure 6.1: SIMD Optimization Benchmark

The ARM Intrinsics implementation achieved nearly a 2x speedup on the ARM M1 machine, with significant speedup also observed on x86 machines. Despite not having the same amount of instructions as the x86 architecture, ARM intrinsics still provide a significant speedup.

6.2 Multithread Benchmark

The multithread benchmark evaluates algorithm performance across different thread counts. Results show solver execution on 100 random positions with varying thread counts:

Machine\Threads [s]	1	2	4	8	16	32	64
ARM M1 h6	31.5337	15.741	8.2442	5.5265	5.4262	5.5982	5.7329
x86 DESK h6	25.2266	12.7536	6.4031	3.3338	2.0516	2.0862	2.2263
x86 DESK h8	9.0389	4.4937	2.2730	1.1949	0.7546	0.7972	0.8111
x86 HPC h6	41.5590	37.5083	29.3422	11.5611	5.1033	2.5110	1.5201
x86 HPC h8	21.7909	10.6266	5.8580	2.9664	1.4652	0.7826	0.5329
x86 HPC h11	4.5995	2.4973	1.1761	0.5062	0.2437	0.1459	0.1286

Figure 6.2: Multithread Benchmark

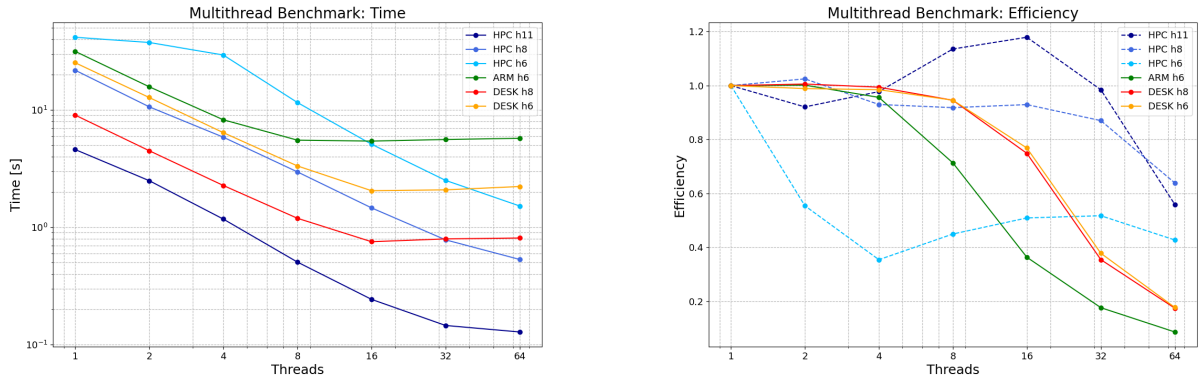

Figure 6.3: Time and Efficiency Multithread Graph

Each machine demonstrates a performance plateau after exceeding its available thread count. Given the multithread parallelization, the efficiency is defined as: $E = \frac{t_1}{t_n \cdot n}$, where t_1 is the time taken by the solver with one thread, t_n is the time taken by the solver with n threads, and n is the number of threads. As said before the HPC machine results are not as expected, but the trend is still visible. The efficiency graph shows that the solver efficiency fluctuates with thread count by surpassing the

100% efficiency threshold. The HPC machine encounters another type of plateau when using the h11k2t64 configuration which is probably caused by the speed of the solving algorithm, the time of allocation of the table affects the efficiency of the solver. We rely on the personal machine results for the most accurate data. The multithreading approach, while not using the work-stealing algorithm, still demonstrates good scalability with low overhead of the mutexes and condition variables.

6.3 Pruning Memory Benchmark

The pruning memory benchmark analyzes algorithm performance with different pruning table sizes to determine optimal solutions. Results show solver execution on 100 random positions using 8 threads (chosen to speedup the benchmarking process for smaller tables):

Machine\Table [s]	**2**	**3**	**4**	**5**	**6**	**8**	**11**
ARM M1	107.3360	40.3601	23.9781	11.3042	5.4890	-	-
x86 DESK	40.3006	22.2812	13.6408	6.6936	3.3243	1.1949	-
x86 HPC	72.0414	39.0187	23.8777	12.3914	8.8530	2.8326	0.4974

Figure 6.4: Pruning Memory Benchmark

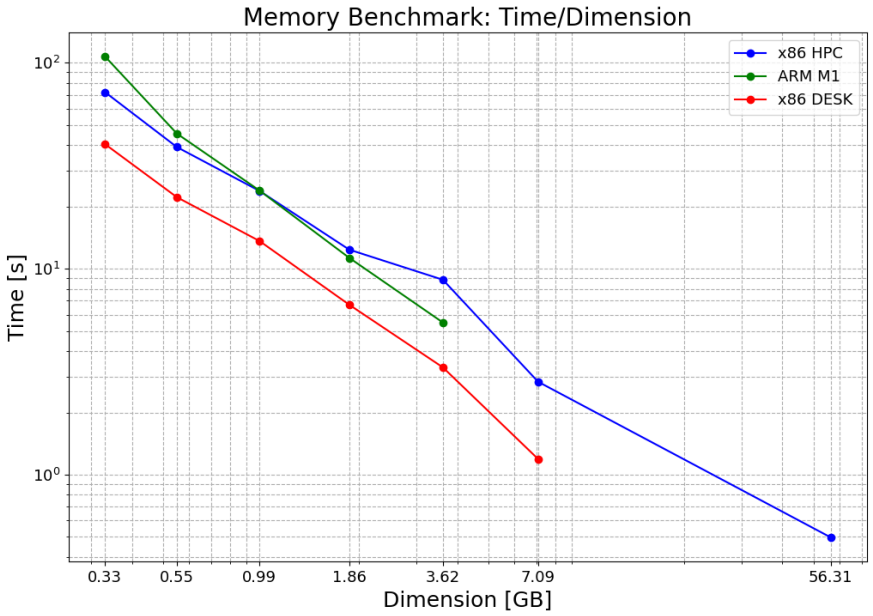

Figure 6.5: Time and Pruning Table Dimension Benchmark Graph

Results demonstrate that solver performance directly correlates with pruning table size and with that the average time per scramble. Theoretically the average time per node should lower with the increase of the pruning table size given that a bigger table means less table fallbacks but we did not have the ability to test this property given the time constraints of smaller tables and the inconsistency of using the HPC machine for bigger ones.

6.4 Nodes Benchmark

The nodes benchmark calculates the number of nodes visited by the algorithm at different depths, with fewer nodes indicating better performance. We also analyze the average time spent per node. We thank Chen Shuang for providing tools and data for comparison with VCube [27]. Results show solver execution on 100 random positions and various depth sets of 25 positions each:

Scrambles\Tables	5	6	8	11
16 optimal	5312696	2685093	1097003	234447
17 optimal	81532628	39169415	14841550	2015182
18 optimal	442192722	215537099	86746820	10647711
19 optimal	2186562107	1031351493	389311907	50150550
100 random	463392390	220212187	89784999	12919092

Figure 6.6: Nodes Benchmark

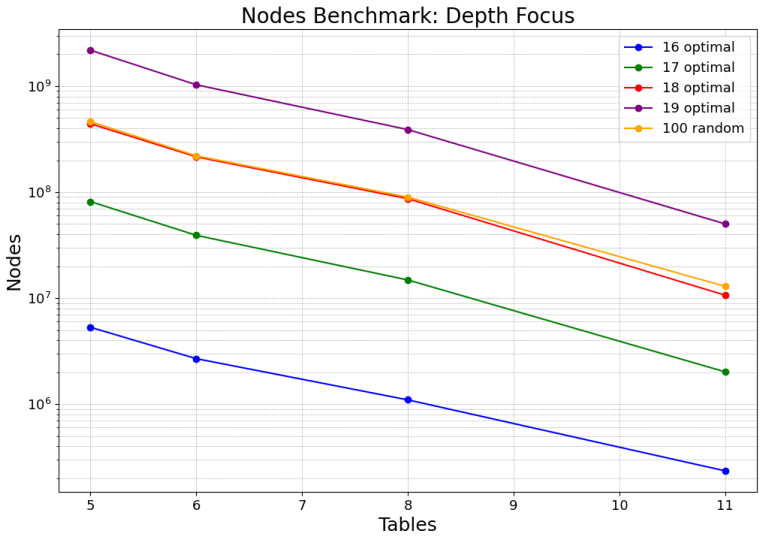

Figure 6.7: Nodes Benchmark Graph

Our analysis reveals that our pruning approach achieves comparable node visits to other modern solvers, but we get lower average time per node, attributable to fewer memory accesses. For VCube, the average pruning table queries per node is approximately 2.1, while for h48, it is 1.1.

6.5 Performance Benchmark

The performance benchmark compares the **H48** solver with VCube using similar-sized pruning tables. Comparisons were all conducted on the same machine (x86 DESK) for compatibility reasons. We need to consider that, the multithreading approaches differ between the two solvers: VCube divides scramble sets among threads for single-threaded execution per scramble, while **H48** executes each scramble with multiple threads. Results show solver execution on 100 random positions and various depth sets:

Solver\Depth [s]	16 optimal	17 optimal	18 optimal	19 optimal	100 random
VCube 2GB t1	0.12013	0.96483	9.43876	41.57013	6.02418
h48 h6k2t1	0.31931	4.59250	25.31363	121.40652	25.22660
VCube 2GB t8	0.02540	0.19332	2.22263	7.45382	1.02503
h48 h6k2t8	0.04346	0.60092	3.35924	16.0101	3.33380
VCube 2GB t16	0.02155	0.16704	1.81945	6.11274	0.85859
h48 h6k2t16	0.02608	0.36420	2.08380	10.22683	2.05160
VCube 7GB t1	0.05063	0.57250	4.48019	20.98285	2.91961
h48 h8k2t1	0.10253	1.26344	6.78263	29.83010	6.81637
VCube 7GB t8	0.01307	0.10493	1.06548	3.69392	0.44242
h48 h8k2t8	0.01424	0.16484	0.90504	4.01291	0.88573
VCube 7GB t16	0.01257	0.08894	0.87754	2.71938	0.38541
h48 h8k2t16	0.01172	0.10508	0.58354	2.65523	0.55320

Figure 6.8: Performance Benchmark

VCube demonstrates superior performance with equivalent pruning tables, attributed to its more optimized implementation and additional heuristics. However, **H48** approaches VCube's performance at higher thread counts. We consider our parallel technique more suitable for the daily user because normally the user does not have to solve sets of scrambles but single scrambles.

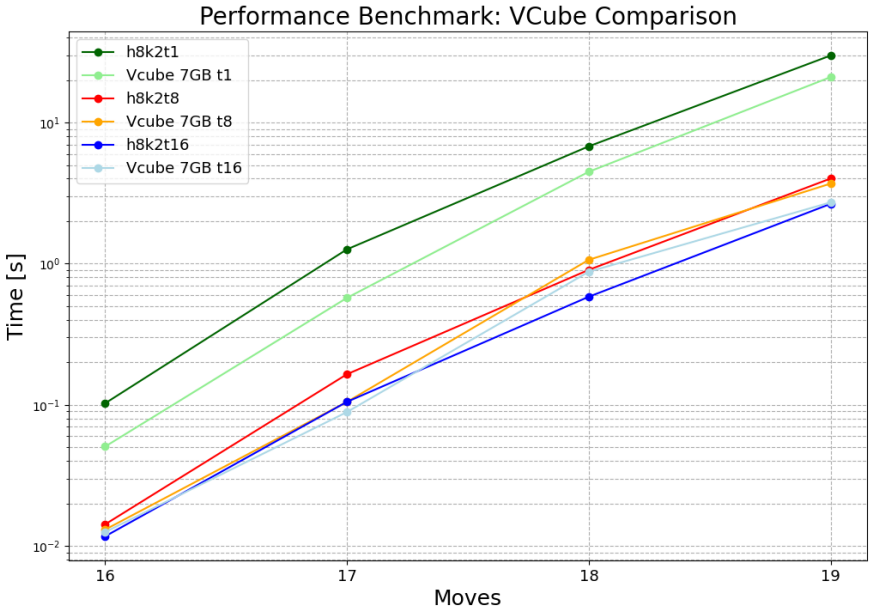

Figure 6.9: Performance Benchmark Graph - h48 h8 and VCube 7GB

6.5.1 Special Scrambles Benchmark

We evaluated the solver against known challenging scrambles. **H48** shows reduced performance on symmetric scrambles and particularly struggles with scrambles having corners in HTR/Fake HTR states (the configuration used in the pruning group). However scrambles with at least one symmetry are rare, occurring in approximately one in 200 million cases (just under 200 billion total cases).

A very challenging scramble is the Superflip, which is a configuration where all the pieces are in the solved position but all the edges are flipped. We consider this scramble the hardest for the solver because it has a very high depth being 20 optimal and the solver has to visit a lot of nodes to solve it and the pruning technique is not very effective. While the VCube solver is able to solve it in a reasonable time, 4:15 minutes with 7GB table, the **H48** solver does very poorly taking above 17 minutes on the most powerful configuration, h11k2t64. Interestingly, our solver performs slightly worse than VCube on random depth-18 scrambles but shows modest improvements at depths 19 and 20. Our approach is more inconsistent but we are able to solve some scrambles faster than VCube. We are confident that with more optimization we can reach the performance of VCube and maybe even surpass it.

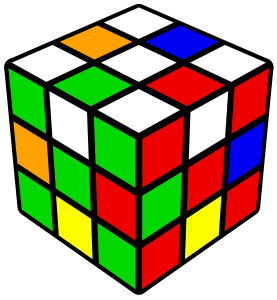

Figure 6.10: Superflip

7 Conclusions

The goal of this thesis was to take part in the development of an advanced optimal solver for the Rubik's Cube. The pruning tables used in this research represent an experimental approach compared to traditional pruning tables. The results obtained are quite promising, and with the new heuristics found for our pruning group, we are confident that we can improve the solver's performance even further. We believe that the use of this new coordinate system can help the community find new 20-optimal solutions and possibly discover previously unexplored positions.

One interesting field that can be exploited is the development of the solver for HPC systems. Currently, the solver demonstrates good performance through its implementation of parallel computing using primitive thread libraries, allowing it to be executed effectively on cluster systems. However, there is significant potential for optimization through the integration of specialized libraries such as MPI and OpenMP. These technologies could enhance the solver's capability to handle complex computations more efficiently, particularly when dealing with extensive search spaces.

Our next goals are to provide a more user-friendly interface for the solver and to further improve its performance. The idea is to use this basis as an API interface for a program that can be used by the community to find optimal solutions and perform FMC analysis faster and more efficiently.

In comparison with VCube, which is the current reference point for optimal solvers, our solver is slightly slower. However, we believe that with the new heuristics we can improve our solver's performance and make it competitive with VCube. The $\mathbf{H_{48}}$ pruning approach has more potential than traditional pruning tables, as it performs fewer lookups in the pruning tables, which, given equal optimization, becomes the bottleneck of the solver. $\mathbf{H_{48}}$ has some features that are more user-oriented, such as the ability to find all optimal solutions for a given position, whereas VCube can only find one. Additionally, VCube is x86 oriented and needs a specific compiler configuration, our solver is more flexible and is optimized for different architectures, which makes it more versatile than VCube. The multithreading part of the solver, which I was most involved in, performs better than VCube, and it allows for the parallelization of a single cube search, which is a feature that VCube does not have.

The graph exploring is a wide field and our work represents just a small part of it. We hope that our work will help the community find new optimal solutions and possibly develop new algorithms to solve the Rubik's Cube.

Bibliography

[1] D.X. Zeng, M. Li, J.J. Wang, B. Wang, and X. Wang. Overview of rubik's cube and reflections on its application in mechanism. *Chinese Journal of Mechanical Engineering*, 31:77, 2018. https://doi.org/10.1186/s10033-018-0269-7.

[2] Speed Solving Wiki. Metric, January 2022. https://www.speedsolving.com/wiki/index.php/Metric.

[3] Janet Chen. Group theory and the rubik's cube, 2004. https://people.math.harvard.edu/~jjchen/docs/Group%20Theory%20and%20the%20Rubik's%20Cube.pdf.

[4] Tom Davis. Group theory via rubik's cube, December 2006. http://geometer.org/rubik/group.pdf.

[5] Lindsey Daniels. Group theory and the rubik's cube. *Lakehead University*, 2014. https://www.lakeheadu.ca/sites/default/files/uploads/77/docs/Daniels_Project.pdf.

[6] Jessica Fridrich. My system for solving rubik's cube, 2006. http://www.ws.binghamton.edu/fridrich/system.html.

[7] Kepler Boyce and Cornelis Storm. Rubik's cube: What separates the fastest solvers from the rest? *Journal of Emerging Investigators*, 2022. https://doi.org/10.59720/21-189.

[8] Jaap Scherphuis. Computer puzzling, 2014. https://www.jaapsch.net/puzzles/compcube.htm.

[9] Richard E. Korf. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI/IAAI*, pages 700–705, 1997. https://www.cs.princeton.edu/courses/archive/fall06/cos402/papers/korfrubik.pdf.

[10] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge. The diameter of the rubik's cube group is twenty. *SIAM Journal on Discrete Mathematics*, 27(2):1082–1105, 2013. https://doi.org/10.1137/140973499.

[11] Herbert Kociemba. Cube explorer, 1997. http://kociemba.org/cube.htm.

[12] Sebastiano Tronto. Nissy, a rubik's cube solver and fmc assistant, 2023. https://nissy.tronto.net.

[13] Thomas Rokicki. Nxopt is an optimal solver for the rubik's cube, 2018. https://github.com/rokicki/cube20src/blob/master/nxopt.md.

[14] Andrew Skalski. Vcube, optimal rubik's cube solver, 2024. https://github.com/Voltara/vcube/.

[15] Sebastiano Tronto and Enrico Tenuti. H48: prototype for a new optimal solver and nissy backend, 2024. https://github.com/enricotenuti/h48/tree/v0.1-thesis.

[16] Sebastiano Tronto and Enrico Tenuti. H48, readme file, 2024. https://github.com/enricotenuti/h48/tree/v0.1-thesis/README.md.

[17] ARM. Neon intrinsics reference, 2024. https://developer.arm.com/documentation/den0018.

[18] ARM. Neon, programmer's guide, 2013. http://www.heenes.de/ro/material/arm/DEN0018A_neon_programmers_guide_en.pdf.

[19] Sebastiano Tronto and Enrico Tenuti. H48, arm neon implementation, 2024. https://github.com/enricotenuti/h48/tree/v0.1-thesis/src/arch/neon.h.

[20] Speed Solving Wiki. Premove, August 2020. https://www.speedsolving.com/wiki/index.php?title=Premoves.

[21] S. Sutradhar, S. Sharmin, and S. Islam. A review on ida* - iterative deepening algorithm heuristics search. In *2022 6th International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 286–288, 2022. https://doi.org/10.1109/ICOEI53556.2022.9776667.

[22] Sebastiano Tronto and Enrico Tenuti. H48, ida* implementation, 2024. https://github.com/enricotenuti/h48/tree/v0.1-thesis/src/solvers/h48/solve.h.

[23] Sebastiano Tronto and Enrico Tenuti. H48, documentation, 2024. https://github.com/enricotenuti/h48/tree/v0.1-thesis/doc/h48.md.

[24] Basel A. Mahafzah. Performance evaluation of parallel multithreaded a* heuristic search algorithm. *Journal of Information Science*, 40(3):363–375, 2014. https://doi.org/10.1177/0165551513519212.

[25] Sebastiano Tronto and Enrico Tenuti. H48, multithreaded implementation, 2024. https://github.com/enricotenuti/h48/tree/v0.1-thesis/src/solvers/h48/solve_multithread.h.

[26] Dongping Xu. Performance study and dynamic optimization design for thread pool systems. *Ames Lab and USDOE Office of Science (SC)*, 2004. https://www.osti.gov/biblio/835380.

[27] Chen Shuang. Cube solver test, a simple benchmark for rubik's cube optimal solvers, 2024. https://github.com/cs0x7f/cube_solver_test.