
A Megaminx Solver

Master thesis by Alexander Botz

Date of submission: October 26, 2025

1. Review: Prof. Dr. Pascal Schweitzer

Darmstadt



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Mathematics Department

Department 04

Mathematics

Didactics

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Alexander Botz, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 26. Oktober 2025



Alexander Botz

Contents

1	Introduction	5
2	Preliminaries	8
2.1	Twisty Puzzles	8
2.2	Computational Solving	10
2.3	Modeling the Megaminx	11
3	Group Theoretic Foundations	14
3.1	Groups	14
3.2	Generating Sets and Words	15
3.3	Cosets	16
3.4	Homomorphisms	17
3.5	Group Actions	18
3.6	Cayley Graphs	20
3.7	Wreath Products	22
3.8	Computational Group Theory	23
4	Megaminx Theory	25
4.1	Megaminx Laws	25
4.2	A Lower Bound on the Diameter of the Megaminx Group	30
5	A Naive Solution to the Factorization Problem of Permutation Groups	32
5.1	Composing Words	32
5.2	The Naive Algorithm	33
6	Breadth-First-Search Approach	36
7	Multi-Phasing	39
7.1	Solving the Word Problem on Cosets	39
7.2	The Multi-Phase Algorithm	42
8	The Megaminx Solver	44
8.1	Subset-Chain	44
8.2	Computing Cosets	45
8.3	The Intuitive Interpretation	47
8.4	Suboptimality of Multi-Phase Solutions	49

9	Implementation	51
9.1	Encoding the Megaminx	51
9.2	Coset Encodings	53
9.3	Precomputations	55
10	Experiments	57
10.1	Developing a Test Setup	57
10.2	Bigger Tables	63
10.3	Alternative Subset-Chain	65
10.4	Meet-In-The-Middle	67
11	Random State Scrambles	69
12	Conclusion	71
13	Acknowledgments	72

1 Introduction

The Rubik's Cube is one of the most popular puzzles to ever exist and has inspired the invention of countless variants, called twisty puzzles. These puzzles still fascinate people of all ages, which led to the development of various solving methods and techniques. Eventually, a competitive element was introduced, leading to a sport known as speedcubing, where people try to solve twisty puzzles as fast as possible. Another interesting discipline arising with twisty puzzles is the so-called Fewest Moves Challenge, in which the goal is to solve a given configuration in as few twists of the puzzle as possible. This challenge has inspired people to develop more efficient solving algorithms, which were often found by utilizing mathematical group theory. Eventually, people got curious as to how computers would perform in the Fewest Moves Challenge, introducing the discipline of computational puzzle solving. The resulting solving algorithms are utilized by the speedcubing community in various ways, for example, to find suitable move sequences for solving methods. They are also applied by engineers to be used in solving robots; for example, one that can solve the Rubik's Cube in only fractions of a second [1]. Beyond these practical applications, one of the main goals within computational puzzle solving is to find the God's Number of a twisty puzzle, which is the maximum number of twists required to solve any configuration of that puzzle. While this number can easily be determined for some puzzles by brute force, this is only an option when the number of possible configurations of that puzzle is sufficiently low. If that is not the case, then it is often just possible to give upper and lower bounds instead. One of the twisty puzzles that has not yet been subjected to a lot of research in this context is the Megaminx, a dodecahedral twisty puzzle. The main goal of this thesis is to develop an efficient solving algorithm for the Megaminx and to establish a lower and upper bound on its God's Number.

The scientific background of computational puzzle solving mostly stems from computational group theory, where the problem of solving twist puzzles reduces to the so-called factorization problem of permutation groups. While there are very efficient algorithms to solve this problem, such as the Schreier-Sims Algorithm, solutions obtained by these algorithms are often very long, and it is currently unknown if a shortest solution can be found efficiently [2, Section 4.8.3]. When the number of possible configurations of a puzzle is sufficiently low, then it is possible to just brute-force optimal solutions to all these configurations, which yields the God's Number of that puzzle. For example, this is the case for the Pocket Cube, a variant of the Rubik's Cube that only consists of its eight corners. The Pocket Cube can only attain 3,674,160 configurations, and its God's Number has been determined to be 11 or 14, depending on if twists of a face by 180° are allowed [3]. While

this result has been independently verified, for example in [4], it has not yet been published within a scientific environment. Generally speaking, it is an unfortunate circumstance that a lot of research on twisty puzzles is performed outside of an academic setting, in which case the findings are often loosely published on a website like Jaap’s [5] or posted in online forums like the speedsolving forum [6]. Hence, it can be very challenging to find reliable information, even regarding results that are considered common knowledge among speedcubers and puzzle enthusiasts, such as the God’s Number of the Pocket Cube. Fortunately, this is not always the case, and some questions, especially more challenging ones, have been answered within an academic environment. In the case of the Rubik’s Cube, the God’s Number has been proven to be exactly 20 in [7], if any twist of a face by a whole multiple of 90° is allowed. While 20 twists are therefore always sufficient to solve the Rubik’s Cube, it has been shown in [8] that it requires at least 26 random moves to properly scramble a Rubik’s Cube. When twists by 180° are disallowed, the God’s Number rises to 26, as shown in [9]. On a more theoretical level, it has been shown in [10] that the God’s number of an $n \times n \times n$ twisty cube is in $\Theta(n^2/\log(n))$.

Results: With the Megaminx as an example, we will work out the mathematical and computational environment in which twisty puzzles and their solving methods can be understood. We will then use this understanding to develop and implement an efficient solving algorithm for the Megaminx and to prove that its God’s Number is between 47 and 133. We will introduce several modifications to our solver and develop a standardized test setup that allows us to compare all of the resulting variants.

Technique: We develop a mathematical model of the Megaminx and find that this model can be described by a permutation group on the colored stickers of the Megaminx. A naive lower bound on the God’s Number of the Megaminx is then determined by the minimal $l \in \mathbb{N}_0$ for which the number of twist sequences of length at most l exceeds the number of existing states. This approach can be further optimized by reducing the number of counted twist sequences that lead to identical states, which provides us with the final lower bound of 47. Utilizing algorithmic group theory and computational theory, we manage to develop a general description for multi-phase solving algorithms of twisty puzzles, which we then utilize to develop and implement such a solving algorithm for the Megaminx. We find that the permutation group of the Megaminx can also be formalized with respect to the behavior of the Megaminx’s edges and corners. This approach will provide us with a more natural model that is especially helpful to use in the actual implementation of the developed Megaminx solver, as the resulting encoding is very efficient. To make the solver very fast in practice, we generate all the necessary data of the phases once in advance and store it for later use. The upper bound on the Megaminx’s God’s Number is then obtained by adding up the worst twist counts of all phases. Efficiency of our solving algorithm is very important, as the general approach is still based on brute force, and inefficiency can quickly cause blow-ups in the required storage for the pregenerated data and the runtimes. Hence, algorithmic engineering is vital to ensure an efficient implementation, as the feasibility of our solver and the chosen multi-phase approach depend on it.

We start with Chapter 2 by introducing some background and terminology on twisty puzzles, and then take a closer look at related works focusing on the Rubik’s Cube. In Chapter 3, we develop the underlying mathematical environment, where we find the problem of solving twisty puzzles to be reducible to a known

group theoretic problem. We then obtain first results for the Megaminx in Chapter 4, namely a more natural description of the Megaminx Group, as well as a lower bound on its God's Number. We develop a first naive algorithm to solve the underlying group theoretic problem in Chapter 5, which motivates us to design a more efficient algorithm based on a breadth-first search approach in Chapter 6. Introducing a divide-and-conquer approach allows us to improve this algorithm in Chapter 7. The result is a general description of solving methods, which we use to design a solving method for Megaminx in Chapter 8. We elaborate on a possible implementation of this solving algorithm in Chapter 9 and experiment with it in Chapter 10, which provides us with an upper bound on the Megaminx's God's number. Lastly, in Chapter 11 we give an example of where such a solver can be utilized in the speedcubing community.

AI was used in this thesis for typesetting and to identify improvements in the writing style. Furthermore, most computations in this thesis were performed with the Lichtenberg high-performance cluster (HPC) of the TU Darmstadt, utilizing 16 cores and 80 GB of memory.

2 Preliminaries

Before we introduce the required mathematical theory, we want to introduce the relevant terms on a more intuitive level, which will later allow us to better understand the interaction between theory and application. We also use this opportunity to give some background on the history of twisty puzzles, speedcubing, and the search for the Rubik's Cube's God's Number.

2.1 Twisty Puzzles

The Megaminx is an example of a *combination puzzle*, which is a puzzle consisting of some set of pieces that can be manipulated into different combinations, usually called *states*. Such a puzzle has a predefined, not necessarily unique, *solved state*, and the goal is to achieve such a state. There is a huge variety of combination puzzles, some of which can be seen in Figure 2.1. More specifically, the Megaminx is a *twisty puzzle*, which is a combination puzzle that possesses some sort of twisting mechanic. Referring back to Figure 2.1, all but the topmost puzzle are twisty puzzles. The puzzle on the top is not considered a twisty puzzle, as its pieces cannot be moved around, but instead the different clocks can only be rotated in place.

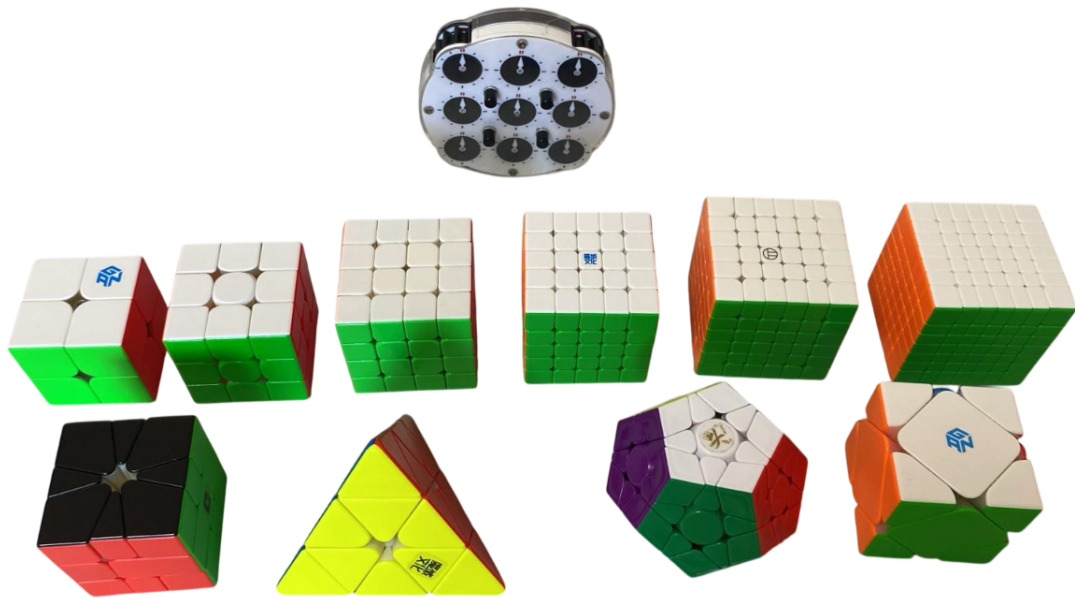


Figure 2.1: Various combination puzzles

The most popular twisty puzzle is the Rubik's Cube, which was invented in 1974 by Ernő Rubik and internationally sold under that name by 1980 [11]. It sparked a worldwide hype, and methods were invented to solve the Rubik's Cube soon after. This development eventually led to a sport known as *speedcubing*, where the goal is to solve the Rubik's Cube as fast as possible. The popularity of speedcubing originally peaked in 1982, when the first Speedcubing World Championship was held in Hungary [12], the origin country of the Rubik's Cube. While the original hype eventually faded away, speedcubing experienced a revival in 2003 when another world championship was held in Canada [13]. However, this time the Rubik's Cube was not the only discipline that was offered, as many more twisty puzzles had been developed in the meantime, such as the Pocket Cube seen in Figure 2.2 or the Rubik's Revenge seen in Figure 2.3. Additionally, variations of regular solving disciplines were offered, such as one-handed solving or blindfolded solving. The interest did not fade away this time, and in 2004 the World Cubing Association was founded to facilitate speedcubing competitions all over the world. To this day, more than 14,000 competitions [14] have been held for over 140,000 competitors in 143 countries [15].

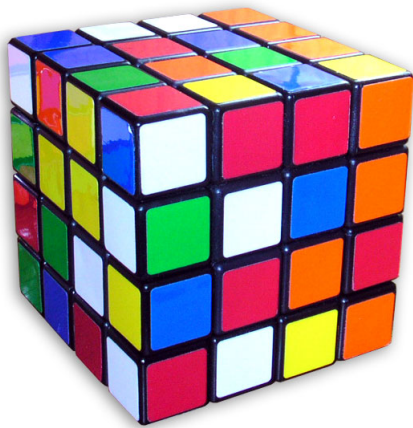


Figure 2.2: The Rubik's Revenge [16]

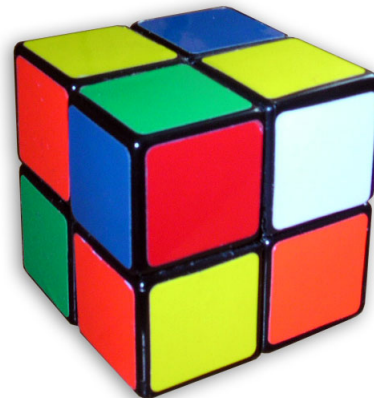


Figure 2.3: The Pocket Cube [17]

Another variation of the Rubik's Cube offered at speedcubing competitions is the Fewest Moves Challenge, where competitors are tasked to solve a given state with as few so-called *moves* as possible. In this case, a move is defined as any twist of a face by a whole multiple of 90° . A twist of the middle layer is not considered a move and instead has to be performed by two suitable moves of the adjacent faces. Rotations of the full puzzle are allowed but not considered moves, so they do not count towards the number of twists used. Such a definition of what constitutes a move is called a *move metric*, and this specific definition is called the *half-turn metric*. Coming from the half-turn metric, we can additionally exclude twists by 180° , which yields the *quarter-turn metric*, or we can include twists of the middle layers, which yields the *slice-turn metric*. These are the three most common move metrics for the Rubik's Cube, and similar move metrics exist for other twisty puzzles.

2.2 Computational Solving

Naturally, people got curious as to how computers could be utilized to push the Fewest Moves Challenge beyond human limits. While this development was not directly relevant for speedcubing competitions, as such assistance is not allowed, it sparked a new kind of challenge and introduced the task of finding the Rubik's Cube's *God's Number*, which is the maximum number of moves required to solve any possible state of the Rubik's Cube. Here, a lot of research has already been done by analyzing the underlying mathematical structures and utilizing a vast amount of computational power. The peak of this research was reached when it was proven in [7] that the Rubik's Cube's God's Number is exactly 20 in the half-turn metric. The authors of that paper first established that the so-called *superflip* configuration, which is depicted in Figure 2.4 and can be obtained from a solved Rubik's Cube by twisting all edges in place, requires exactly 20 moves to be solved. After that, all possible states were solved in at most 20 moves, which required about a billion seconds of CPU

time and a lot of mathematical expertise to optimize the runtime sufficiently. With the same approach, it was proven later on in [9] that the God's Number of the Rubik's Cube in the quarter-turn metric is exactly 26.

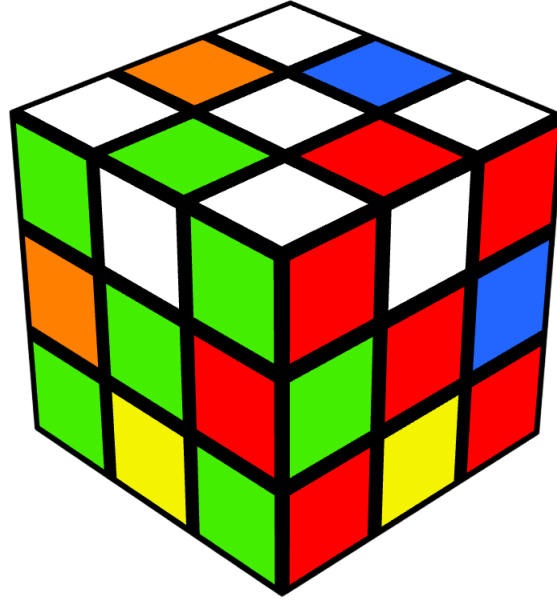


Figure 2.4: The superflip configuration of the Rubik's Cube

2.3 Modeling the Megaminx

We are already familiar with the Rubik's Cube, which is depicted in Figure 2.1. It consists of a core to which all of the 6 centers are attached, as well as 12 edges and 8 corners. The centers, edges, and corners have one, two, and three colored stickers (or tiles) attached to them, respectively, and each piece is uniquely identified by the stickers attached to it. The Rubik's Cube is considered to be solved when the colors on each of the 6 faces are matching up. It can then be scrambled by twisting the faces, which will move around the colored stickers, posing the challenge of getting the puzzle back into a solved state. The core itself does not possess a twisting mechanic, and relative to it, the centers can only be twisted in place. Fixing the core's orientation also fixes the position of all centers, in which case the solved state becomes unique. We can then describe states by the orientation and position of all corners and edges.

The Megaminx, depicted in Figure 2.5, is a dodecahedron twisty puzzle. Its mechanical structure is very similar to that of the Rubik's Cube and can be seen in Figure 2.6. It consists of

- 30 *edges*, each of which has 2 colored stickers attached to it.
- 20 *corners*, each of which has 3 colored stickers attached to it.

- 12 *centers*, each of which has 1 colored sticker attached to it.
- A *core*, to which all the centers are attached.

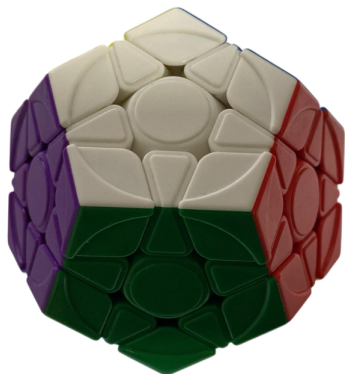


Figure 2.5: The Megaminx

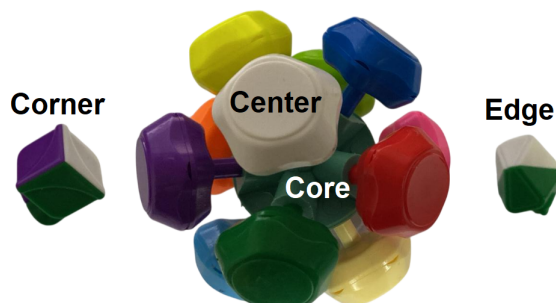


Figure 2.6: The pieces of the Megaminx

As was the case for the Rubik's Cube, each piece can be uniquely identified by its attached stickers. We fix the orientation shown in Figure 2.5, where the front center is dark green and the top center is white. Again, we can now identify faces by their center's sticker color and describe states by the orientation and position of all corners and edges. Furthermore, we again have a unique solved state, and we define an edge or corner to be solved when its position and orientation match that of it in the solved state.

Next, we introduce notation for the twists that can be applied to the puzzle. As we only allow twists of the faces, we only need notation for those twists. We assign a name to each face, as can be seen in Figure 2.7, which allows us to define the set of allowed twists.

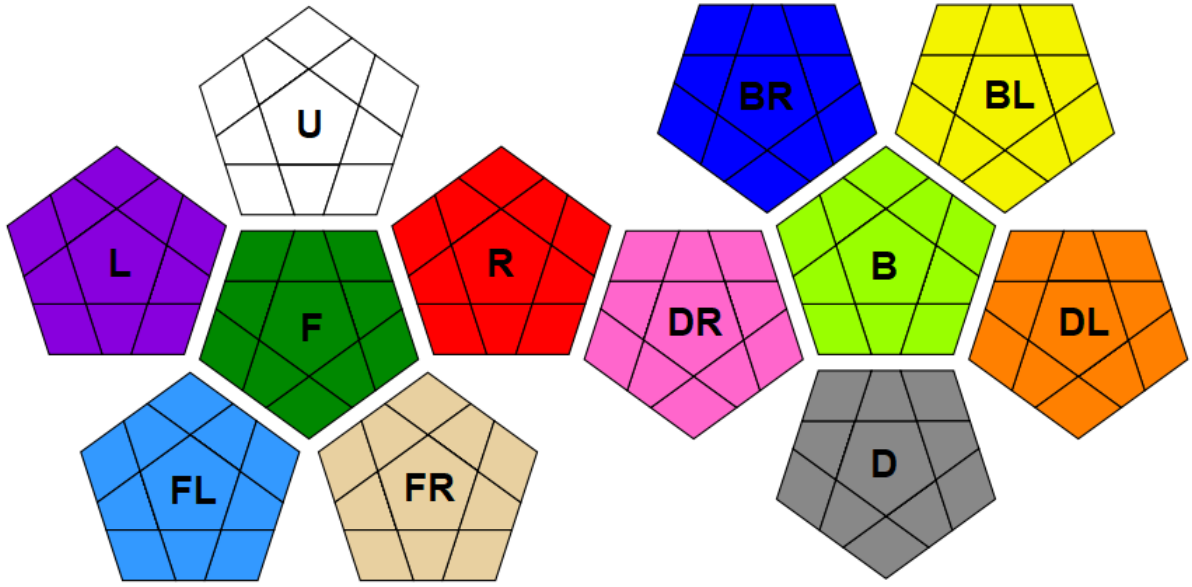


Figure 2.7: Names of the Megaminx's faces depicted on a two-dimensional unfolding

Definition 1 (Moves and Move-Sequences on the Megaminx).

We denote by F_M the set $\{U, R, F, L, BR, BL, FR, FL, DR, DL, B, D\}$ of faces of the Megaminx, and a twist of any such face by a whole multiple of 72° is called a move. For some $x \in F_M$ we denote

- a 72° clockwise twist of x by x ,
- a 144° clockwise twist of x by x^2 ,
- a 216° clockwise twist of x by x^2' and
- a 288° clockwise twist of x by x' .

The set of all such moves is denoted by S_M . By writing them in order from left to right, we can compose moves to move sequences, and the number of moves in such a sequence is called its length. A solution to some state c of the Megaminx is a move-sequence that yields the solved state when applied to c . A solution to c of shortest possible length is said to be optimal, and the length of any optimal solution is called the depth of c .

The notation established here aligns with the one used by the online tool Twizzle [18], which I have utilized to create all digital visualizations of the twisty puzzles in this thesis. Definition 1 yields a model of the Megaminx that we can now analyze in a mathematical environment.

3 Group Theoretic Foundations

We collect some basic definitions and results from [2, Chapter 2] that will be required for later proofs. For some $n \in \mathbb{N}_0$ we denote by $[n]$ the set $\{1, \dots, n\}$ and by $[n]_0$ the set $\{0, \dots, n\}$.

3.1 Groups

Definition 2 (Groups).

A group $(G, \cdot)$ is a set G together with a composition map $\cdot : G^2 \rightarrow G$, for which the following conditions hold.

1. For all $a, b, c \in G$ we have $(a \cdot b) \cdot c = a \cdot (b \cdot c)$, i.e., the composition map is associative.
2. There exists a unique neutral element $e \in G$ such that for all $a \in G$ we have $a \cdot e = e \cdot a = a$.
3. For all $a \in G$ there exists a unique inverse element $a^{-1} \in G$ such that $a^{-1} \cdot a = a \cdot a^{-1} = e$.

We usually just introduce a group $(G, \cdot)$ by the set G and let the composition map $\cdot$ be given implicitly. If there is more than one group, then we refer to the composition map of G by $\cdot_G$ to avoid confusion. The following example depicts one of the most crucial types of groups.

Definition 3 (Symmetric Groups).

For a set Ω , we call a bijective map $\sigma : \Omega \rightarrow \Omega$ a permutation of the domain Ω . The set of all permutations of Ω forms a group together with the natural composition of maps, the Symmetric group of Ω , denoted by $\text{sym}(\Omega)$. For $\alpha \in \Omega$ and $g \in \text{sym}(\Omega)$ we denote $\alpha^g := g(\alpha)$. We denote by S_n the symmetric group on $\Omega = [n]$.

Note here that every symmetric group on a finite domain of size $n < \infty$ can be treated as S_n by renaming the elements of the domain. Oftentimes we do not have all permutations on a set available, in which case we have a substructure of a symmetric group.

Definition 4 (Subgroups).

A subgroup H of a group G is a group such that $H \subseteq G$ and $\cdot_G|_H = \cdot_H$ hold. We denote the subgroup relation by $H \leq G$, and if the subset relation is strict, we similarly write $H < G$.

Definition 5 (Permutation Groups).

Subgroups of symmetric groups are called permutation groups.

Definition 6 (Fixed Points).

For a domain Ω and some $\pi \in \text{sym}(\Omega)$, we call $\alpha \in \Omega$ a fixed point of π if $\alpha^\pi = \alpha$ holds true. We denote by $\text{fix}(\pi)$ the set of fixed points of π . Similarly, we call $\alpha \in \Omega$ a fixed point of a set $A \subseteq \text{sym}(\Omega)$ if it is a fixed point of each element in A , and we denote with $\text{fix}(A)$ the set of fixed points of A .

One important takeaway here is that fixed points of permutation groups are redundant and can be omitted from the domain for the sake of simplicity.

3.2 Generating Sets and Words

Definition 7 (Generating Sets).

For a group G and a subset $S \subseteq G$, we denote by $\langle S \rangle$ the smallest subgroup of G that contains all elements in S . We then call S a generating set of $\langle S \rangle$ and the elements of S generators of $\langle S \rangle$.

We assume all generating sets to be closed under inverses. Since each piece of the Megaminx is unique, we can identify all of its stickers to obtain a domain.

Definition 8.

We denote by Ω_M the set of all 120 stickers of the Megaminx that are not placed on a center.

The moves defined in Definition 1 permute Ω_M , and since the set of moves S_M is closed under inverses, it is a generating set of a group.

Definition 9.

We denote by G_M the permutation group generated by S_M and call it the Megaminx Group. Furthermore, we call S_M the standard generating set of the G_M .

The stickers placed on the centers of the Megaminx could be included in Ω_M , but since they would be fixed points anyways, it makes sense to omit them for the sake of simplicity. Note that we cannot swap a sticker from an edge and a corner, so G_M is not a symmetric group.

Definition 10 (Words).

For a group G with generating set S , some $n \in \mathbb{N}_0$ and $s_j \in S$ for all $j \in [n]$, we call $s_1 \cdots s_n$ a word over S of length n . We denote by W_S the set of all words over S and define the map $\phi : W_S \rightarrow G : s_1 \cdots s_n \mapsto s_1 \cdots s_n$. For some $g \in G$ we call $w \in W_S$ a word of g if $\phi(w) = g$ holds, and we consider two words to be equivalent if they are words of the same group element. We denote by ϵ the empty word, which is a word of e . The word length of some $g \in G$ is the length of a shortest word of g .

Lemma 11.

For a group G and a subset S of G , we have $g \in \langle S \rangle$ if and only if there is a word of g over S .

Corollary 12.

Every element in the Megaminx Group is a composition of moves.

Definition 13 (The Factorization Problem in Permutation Groups).

Let G be a group with generating set S , and let $g \in G$. The task of finding a word of g over S is called the Factorization Problem in Permutation Groups (FPPG).

FPPG can be considered the main problem of this thesis, which can be seen by considering the Megaminx as an example.

Example 14 (Finding Solutions).

For a given state c of the Megaminx, let $g_c \in G_M$ be the permutation that permutes c into the solved state. Then a word $s_1 \cdots s_n \in W_{S_M}$ of length $n \in \mathbb{N}_0$ is a solution to FPPG of g_c if and only if the move sequence $s_n \cdots s_1$ is a solution to c . Hence, the problem of finding solutions reduces to FPPG.

The order of letters in the word has to be inverted, as move sequences are read from left to right. We note that FPPG can be solved efficiently with the Schreier-Sims Algorithm, but the resulting words tend to be very long [2, Section 4.8.3]. While the word lengths can be reduced by optimizing the strong generating sets [19], we want even shorter words, and hence, this approach is not suitable for us.

3.3 Cosets

Definition 15 (Order).

For a set A , we denote by $|A|$ the number of elements in A , and we call A finite if $|A| < \infty$ holds. When A is a group, we also call $|A|$ the order of A . For a group G and some $g \in G$, we denote $|\langle g \rangle|$ by $\text{ord}(g)$ and call it the order of g .

Definition 16 (Cosets).

For a group G and a subgroup $H \leq G$, we call $gH := \{gh \mid h \in H\}$ the left coset of H in G containing g and $Hg := \{hg \mid h \in H\}$ the right coset of H in G containing g .

The underlying theory of this thesis can be phrased for both left and right cosets equivalently, but we will only focus on right cosets; therefore, we assume every coset to be a right coset unless explicitly stated otherwise.

Theorem 17 (Coset Structure).

For a group G and a subgroup $H \leq G$, the cosets of H in G are pairwise either disjoint or equal. Furthermore, group elements $g, g' \in G$ lie in the same coset if and only if $g'g^{-1} \in H$.

Theorem 18 (Lagrange).

For a finite group G and a subgroup $H \leq G$, the numbers of left cosets and right cosets coincide. It is denoted by $[G : H]$ and called the index of H in G . Furthermore, it holds that $[G : H] = \frac{|G|}{|H|}$.

Definition 19 (Normal Subgroup).

For a group G and a subgroup $N \leq G$, we call N a normal subgroup if $gN = Ng$ holds for all $g \in G$, in which case we write $N \trianglelefteq G$.

Theorem 20 (Factor Groups).

For a group G , some $g \in G$, and a normal subgroup $N \trianglelefteq G$, we obtain a group structure on the set of cosets with the composition map defined by

$$gN \cdot g'N = (g \cdot g')N \quad (3.1)$$

for all $g, g' \in G$. We call this the factor group of G modulo N , denoted by G/N .

Example 21.

For the group $(\mathbb{Z}, +)$ of integers and every $n \in \mathbb{N}$, we have $(n\mathbb{Z}, +) \trianglelefteq (\mathbb{Z}, +)$. We denote by $\mathbb{Z}_n$ the group $\mathbb{Z}/n\mathbb{Z}$, which is called the factor group of integers modulo n .

3.4 Homomorphisms

Definition 22 (Homomorphisms).

For groups G, H we call a map $\varphi : G \rightarrow H$ a (group-)homomorphism, if for all $g, g' \in G$ it holds that $\varphi(g \cdot_G g') = \varphi(g) \cdot_H \varphi(g')$.

Example 23.

For some group G and a generating set S of G , the set W_S of words over S becomes a group together with the map that concatenates two words. In that case the map ϕ from Definition 10 is a group homomorphism from W_S to G .

Definition 24 (Isomorphisms and Automorphisms).

A homomorphism φ is called an isomorphism if it is bijective. If there exists an isomorphism between groups G and H , we call them isomorphic, and we denote that relation with $G \cong H$. An isomorphism from a group into itself is called an automorphism.

Definition 25.

The set of automorphisms of a group G together with the natural composition of maps forms a group, the automorphism group, denoted by $\text{Aut}(G)$.

Definition 26 (Kernel and Image of a Group Homomorphism).

For groups G, H and a homomorphism $\varphi : G \rightarrow H$, we denote by $\ker(\varphi)$ the kernel $\{g \in G \mid \varphi(g) = e\}$ and by $\text{im}(\varphi)$ the image $\varphi(G)$ of φ .

Theorem 27.

For groups G, H and a homomorphism $\varphi : G \rightarrow H$, the kernel of φ always forms a normal subgroup of G and the image of φ always forms a subgroup of H .

Example 28.

Let $\pi \in S_n$; then we call a pair $(a, b) \in [n]^2$ an inversion if $a < b$ and $\pi(a) > \pi(b)$ holds. We can count the number of inversions in a permutation and determine if this number is even or odd, which induces a homomorphism from S_n to $\mathbb{Z}_2$. The kernel of this homomorphism is denoted by A_n and called the alternating group. The elements of A_n are also called even permutations, and any subgroup of an alternating group is called an even group.

3.5 Group Actions

Definition 29 (Group Actions).

For some set Ω and a group G , we call a group homomorphism $\psi : G \rightarrow \text{sym}(\Omega)$ a group action. In this case we say that G acts on Ω via ψ .

When ψ is known from context, we just treat the elements of G as permutations on Ω and omit ψ .

Definition 30 (Orbit).

Let $\psi : G \rightarrow \text{sym}(\Omega)$ be a group action and $\alpha \in \Omega$; then we call $\alpha^G := \{\alpha^g \mid g \in G\}$ the orbit of α under ψ .

Definition 31 (Image).

Let $\psi : G \rightarrow \text{sym}(\Omega)$ be a group action and $\Delta \subseteq \Omega$; then we denote by $\Delta^g := \{\alpha^g \mid \alpha \in \Delta\}$ the image of Δ under g .

Definition 32 (Stabilizer).

Let $\psi : G \rightarrow \text{sym}(\Omega)$ be a group action and $\alpha \in \Omega$; then we call the subgroup $\{g \in G \mid \alpha^g = \alpha\}$ of G the stabilizer of α and denote it by G_α .

Theorem 33 (Orbit-Stabilizer Theorem).

Let G be a finite group, $\psi : G \rightarrow \text{sym}(\Omega)$ a group action, and $\alpha \in \Omega$; then it holds that $|G| = |\alpha^G| |G_\alpha|$.

Definition 34 (Pointwise and Setwise Stabilizer).

For a group G and some $B \subseteq \Omega$, we call

- $G_B := \{g \in G \mid \beta^g = \beta \text{ for all } \beta \in B\}$ the Pointwise Stabilizer of B and
- $G_{\{B\}} := \{g \in G \mid B^g = B\}$ the Setwise Stabilizer of B .

Definition 35 (Faithfulness).

A group action is said to be faithful if its kernel is the trivial group $\{e\}$.

Definition 36 (Right Regular Action).

For a group G we obtain a faithful action $\psi : G \rightarrow \text{sym}(G)$ by letting $g^{\psi(g')} := g \cdot g'$ for all $g, g' \in G$. This action is called the right regular action.

Definition 37 (Coset Action).

For a group G , a subgroup $H \subseteq G$, and Ω as the set of cosets of H in G , we can define an action $\psi : G \rightarrow \text{sym}(\Omega)$ by letting $(Hg)^{\psi(g')} := H(g \cdot g')$ for all $g' \in G$ and $Hg \in \Omega$, which we call the coset action.

The right regular action corresponds to the coset action of $H = \{e\}$.

3.6 Cayley Graphs

Definition 38 (Graphs).

A graph G is a tuple (V, E) that consists of a set of nodes V and a set of edges $E \subseteq V^2$. A graph is said to be finite when V is finite. A graph is said to be undirected when E is closed under transpositions.

We assume all graphs to be undirected for the remainder of this thesis.

Definition 39 (Walks and Paths).

For a graph (V, E) , $s, t \in V$ and some $l \in \mathbb{N}_0$, we call a finite sequence of adjacent edges $(v_i, v_{i+1})_{i \in [l]}$ an s - t -walk of length l , if $v_1 = s$ and $v_{l+1} = t$. A walk is called a path if the v_i are all distinct.

Definition 40 (Distance).

For a graph (V, E) and some $a, b \in V$, we call the length of a shortest path between a and b the distance between a and b . If no such path exists, the distance between a and b is ∞ .

Definition 41 (Connected Graphs).

A graph (V, E) is said to be connected if there is an s - t -path for all $s, t \in V$.

Definition 42 (Diameter of a Graph).

The diameter of a finite connected graph (V, E) is the maximum distance between any two nodes of (V, E) .

Definition 43 (Cayley Graph).

For a group G and a generating set S of G , we call (G, E_S) with $E_S := \{(g, g \cdot s) \mid g \in G, s \in S\}$ a Cayley graph. We call the distance between e and some element $g \in G$ in the Cayley graph the depth of g .

This graph is undirected, since we assumed for generating sets to be closed under inverses. The Cayley graph encapsulates the right regular action's effect of the elements of S on the elements of G . Clearly, a Cayley graph is finite if and only if the group G is finite, which is the case for all twisty puzzle groups.

Theorem 44 (Words in the Cayley Graph).

Let G be a group with generating set S and (G, E_S) its Cayley graph. Then every word $s_1 \cdots s_l$ of some $g \in G$ can be associated with the walk

$$(e, s_1)(s_1, s_1 \cdot s_2) \cdots (s_1 \cdot \dots \cdot s_{l-1}, s_1 \cdot \dots \cdot s_l) \quad (3.2)$$

of length l from e to g in the Cayley graph and vice versa.

Hence, the depth of some $g \in G$ is also the word length of g . We obtain the following result as an immediate consequence.

Theorem 45.

For a group G , a generating set S of G , and an element $g \in G$, the Factorization Problem of Permutation Groups 13 is equivalent to finding a walk from e to g in the Cayley graph (G, E_S) . Such a walk is a shortest path if and only if the associated word of g is minimal.

Hence, the problem of finding minimal words reduces to the problem of finding shortest paths. From Theorem 44 we also obtain that Cayley graphs are connected, which motivates another definition.

Definition 46 (Diameter of a Group).

Let G be a group with generating set S and (G, E_S) its Cayley graph. The diameter of G with respect to S is the diameter of (G, E_S) .

We can conclude that the diameter of a group G is also the maximum word length of all elements in G . Hence, the diameter of G_M is the maximum number of moves required to solve any possible state of the Megaminx, also called the God's Number of the Megaminx.

When a group's order is known, we can easily find a lower bound on its diameter.

Theorem 47 (Lower Bound on a Group's Diameter).

For a group G with generating set S , we have that $\min\{l \in \mathbb{N}_0 \mid \sum_{i=0}^l |S|^i \geq |G|\}$ is a lower bound on the diameter of G with respect to S .

Proof. Let $l \in \mathbb{N}_0$; then $\sum_{i=0}^l |S|^i$ is the number of words of length at most l and hence an upper bound to the number of group elements with a word length of at most l . If there are more group elements than that, then a group element with higher word length must exist, which concludes the proof. $\square$

This is generally only a lower bound, as distinct words can be equivalent. Working with cosets instead of group elements, we can generalize the notion of a Cayley graph.

Definition 48 (Coset Graph).

For a group G with generating set S , a subgroup $H \leq G$, and Ω as the set of cosets of H in G , we call (Ω, E_S) with $E_S := \{(\omega, \omega^s) \mid \omega \in \Omega, s \in S\}$ a coset graph. The distance between H and some coset gH with $g \in G$ is called the depth of gH .

For $H = \{e\}$ we obtain the Cayley graph of G with respect to S , so this is indeed a generalization of Cayley graphs.

3.7 Wreath Products

This section mostly follows [20, Chapter 2.5-2.6].

Definition 49 (Direct Product).

For groups G and H , we can obtain a group structure on the Cartesian product $G \times H$ with elementwise composition. This group is called the direct product of G and H .

If G and H are finite, it holds that $|G \times H| = |G||H|$.

Definition 50 (Semidirect Product).

For groups G, H and a group action $\phi : H \rightarrow \text{Aut}(G)$ we define the semidirect product of G by H on the set $G \times H$ with composition given by

$$(g_1, h_1) \cdot (g_2, h_2) = (g_1^{\phi(h_2)} g_2, h_1 h_2) \quad (3.3)$$

and denote this group by $G \rtimes_{\phi} H$.

If G and H are finite, it holds that $|G \rtimes_{\phi} H| = |G||H|$.

Definition 51 (Group of Functions).

Let A, B be nonempty sets; then we denote by $\text{Func}(A, B)$ the set of functions from A to B . If B is a group, then we obtain a group structure on $\text{Func}(A, B)$ with the elementwise composition

$$(fg)(\alpha) = f(\alpha) \cdot g(\alpha) \text{ for all } f, g \in \text{Func}(A, B), \alpha \in A.$$

If additionally A is finite, then the group $\text{Func}(A, B)$ is isomorphic to the group $B^{|A|}$.

Definition 52 (Wreath Product).

Let G, H be groups, and suppose H acts on some nonempty set Ω . Let $\phi : H \rightarrow \text{Aut}(\text{Func}(\Omega, G))$ be the group action defined by

$$f^h(\alpha) := f(\alpha^h) \text{ for all } f \in \text{Func}(\Omega, G), \alpha \in \Omega, h \in H.$$

Then we call $\text{Func}(\Omega, G) \rtimes_{\phi} H$ the wreath product of G by H and denote it by $G \wr_{\Omega} H$. Furthermore, we call

$$B := \text{Func}(\Omega, G) \cong \text{Func}(\Omega, G) \times \{e\}$$

the base group of the wreath product.

By definition, the elements of $G \wr_{\Omega} H$ are elements in $\text{Func}(\Omega, G) \times H$, so we can write them as (f, h) for some $f \in \text{Func}(\Omega, G), h \in H$. If Ω is finite of size $n \in \mathbb{N}$, we have $B \cong G^n$, and we can rename the elements of Ω such that $\Omega = [n]$ holds true. In that case we get for some $(g_1, \dots, g_n) \in B, h \in H$ that $(g_1, \dots, g_n)^{\phi(h)} = (g_{1^h}, \dots, g_{n^h})$, so $\phi(h)$ permutes the components of B . If G, H are finite as well, we obtain $|G \wr_{\Omega} H| = |G|^{| \Omega |} |H|$.

3.8 Computational Group Theory

The goal of this chapter is to establish a computational framework in which we can handle permutation groups. This will later allow us to analyze our algorithms and argue their correctness. For the entirety of this chapter let Ω be a domain of size $n < \infty$ and let every group be a (finite) permutation group on Ω . We can encode Ω with space $\mathcal{O}(\log(n))$, a permutation of Ω with space $\mathcal{O}(n \log(n))$, and a group G by a generating set S with space $\mathcal{O}(n \log(n) |S|)$. In permutation groups we can perform the basic group-related tasks efficiently.

Lemma 53 (Basic Operations).

For a group G and some $g, g' \in G$, the following computations are in $\mathcal{O}(n)$.

1. Computing $g \cdot g'$.
2. Computing g^{-1} .
3. Checking if $g = g'$ holds true.

Proof. This immediately follows from the fact that all of these can be performed elementwise on Ω . □

We consider these computations as *elementary actions* and express runtimes by the number of performed elementary actions. This simplifies calculations, but we have to keep the missing factor of n in mind. Furthermore, we need to make an assumption on how we can work with cosets.

Assumption 54 (Encoding and Computing of Cosets).

For a group G generated by a set S_G , a subgroup $H \leq G$ generated by a set S_H , and some $g \in G$, we assume the existence of an algorithm `compCoset`, which takes S_G, g, S_H as an input and returns the coset Hg of H in G containing g . We further assume that `compCoset` has a runtime in $\mathcal{O}(n)$ and that the encoding of the output requires space in $\mathcal{O}(n \log(n))$.

Since the runtime is the same as that of the basic group-related tasks in Lemma 53, it makes sense to consider an execution of *compCoset* an elementary action as well. It is not at all clear that this assumption is generally true, and we will later have to show that it applies to our situation. Lastly, we want to state some problems that we already know how to solve efficiently.

Theorem 55.

For a group G given by a generating set S_G , we can with a runtime polynomial in $n, |S_G|, \log(|G|)$

1. compute the orbit of an element in Ω ,
2. test membership in a subgroup,
3. compute the order of G ,
4. compute the pointwise stabilizer of some $B \subseteq \Omega$,
5. draw elements from G uniformly at random.

If H is another group given by a generating set S_H , we can verify $H \leq G$, $H < G$, or $H = G$ in time polynomial in $n, |S_G|, |S_H|, \log(|G|), \log(|H|)$.

Proof. For items 1 to 5 we refer to [21, Section 2.1., 3.1., 2.2.]. For the relationships between G and H , we first determine $H \subseteq G$ by verifying $S_H \subseteq \langle S_G \rangle$, which only adds a factor of $|S_H|$ to the runtime of the third item. Strictness and equality can then be checked by computing and comparing the orders of H and G . $\square$

These computations can be performed with the tool *Groups Algorithms Programming* [22], or GAP for short, which we will utilize for some proofs. I have written an encoding of the Megamix in GAP for this thesis, which can be found in [23].

4 Megaminx Theory

Having introduced the required mathematical background on permutations and groups, we can now develop a more detailed understanding of the Megaminx Group and prove our first results. In this chapter, we will learn about an alternative way to describe the Megaminx Group with respect to the behavior of its pieces, as opposed to its stickers. This description will later allow us to efficiently encode elements of the Megaminx Group for our solving algorithm. Furthermore, we set a starting point for the search of the diameter of the Megaminx Group by proving a lower bound for it.

4.1 Megaminx Laws

While the elements of the Megaminx Group are naturally expressed as move sequences, their permutation on Ω_M can also be achieved by disassembling and reassembling the Megaminx. More precisely, we can remove all the corners and edges from the core and suitably place them back into the core again. The set of permutations achieved by such a process forms a group $\hat{G}_M$, of which G_M has to be a subgroup. When reassembling the Megaminx, we have to place 20 corners in 3 possible orientations each and 30 edges in 2 possible orientations each, which yields a total of $|\hat{G}_M| = 20! \cdot 3^{20} \cdot 30! \cdot 2^{30} \approx 2.42 \cdot 10^{69}$ possibilities to assemble the Megaminx. However, we also obtain the following result:

Theorem 56.

The Megaminx Group has order $|G_M| = \frac{|\hat{G}_M|}{24} \approx 1.00 \cdot 10^{68}$.

Proof. We can compute $|G_M|$ with GAP, which already yields the desired result. □

Corollary 57.

It holds that $G_M < \hat{G}_M$ with $[\hat{G}_M : G_M] = 24$.

We can give some examples of permutations in $\hat{G}_M \setminus G_M$.

Definition 58.

Let $P_1, P_2, P_3, P_4 \in \hat{G}_M$ be the permutations visualized in Figure 4.1.

We can verify with GAP that these permutations are in $\hat{G}_M \setminus G_M$, and it turns out that they are archetypes of the cosets of G_M in $\hat{G}_M$.

Theorem 59.

A set of coset representatives of G_M in $\hat{G}_M$ is given by the set

$$\{P_1^{n_1} \cdot P_2^{n_2} \cdot P_3^{n_3} \cdot P_4^{n_4} \mid 0 \leq n_i < \text{ord}(P_i) \text{ for all } i \in [4]\}. \quad (4.1)$$

Proof. We have $\text{ord}(P_1) = \text{ord}(P_2) = \text{ord}(P_3) = 2$ and $\text{ord}(P_4) = 3$, so there are at most 24 such elements. Utilizing Theorem 17, we can verify with GAP that all these elements lie in pairwise different cosets, and the statement now follows from $[\hat{G}_M : G_M] = 24$. $\square$

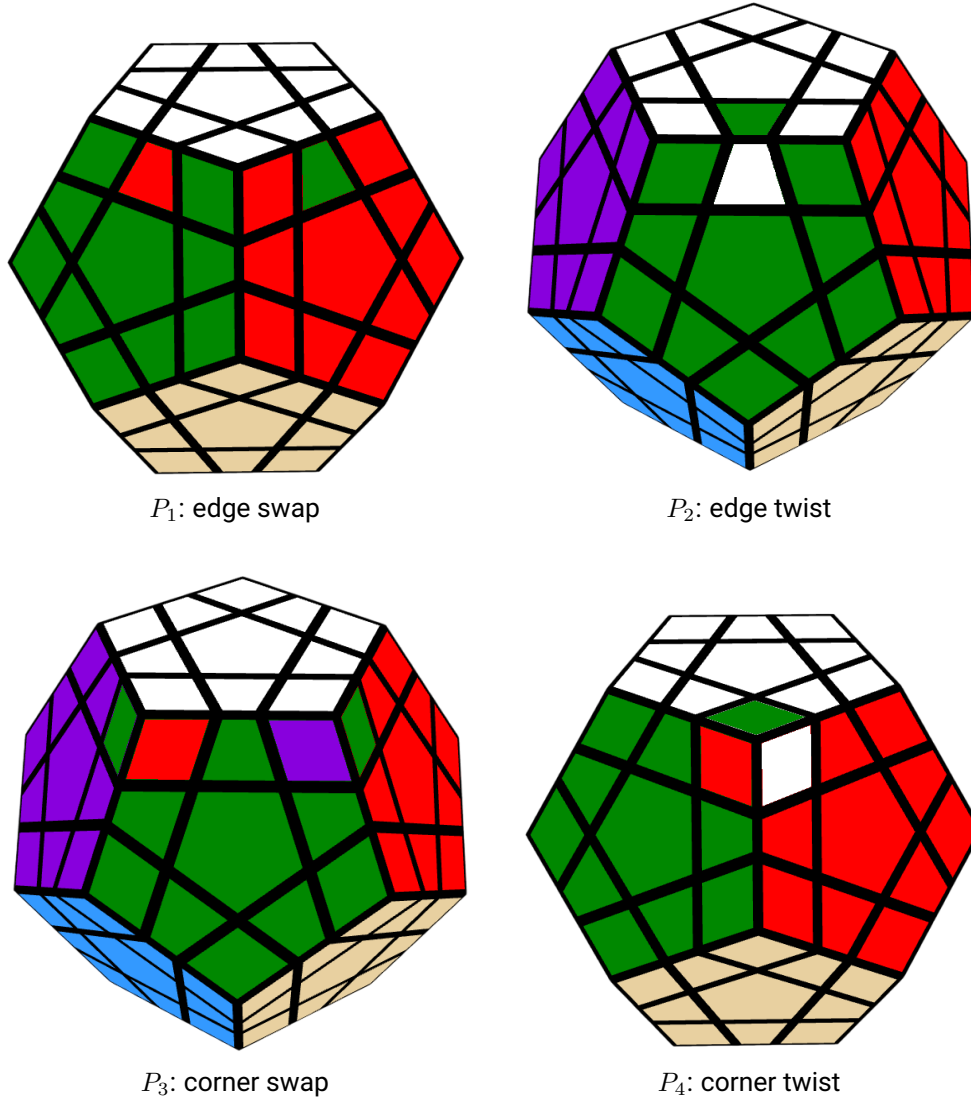


Figure 4.1: States obtained by application of P_i onto the solved state

It follows immediately from the definition of $\hat{G}_M$ that it can be described as a Cartesian product.

Theorem 60.

Suppose $\hat{G}_E < \hat{G}_M$ is the subgroup of edge-sticker permutations and $\hat{G}_C < \hat{G}_M$ is the subgroup of corner-sticker permutations; then it holds that

$$\hat{G}_M \cong \hat{G}_E \times \hat{G}_C. \quad (4.2)$$

Proof. This immediately follows from the fact that corners and edges are independent in $\hat{G}_M$. □

The permutations P_1 and P_2 only affect edges, so we have $P_1, P_2 \in \hat{G}_E$, and similarly it holds that $P_3, P_4 \in \hat{G}_C$. The structure of $\hat{G}_E$ and $\hat{G}_C$ can be understood using wreath products.

Theorem 61.

It holds that $\hat{G}_E \cong \mathbb{Z}_2 \wr_{[30]} S_{30}$ and $\hat{G}_C \cong \mathbb{Z}_3 \wr_{[20]} S_{20}$.

Proof. We only do the proof for $\hat{G}_E$, as the proof for $\hat{G}_C$ follows similarly. We obtain a bijection $\varphi : \hat{G}_E \rightarrow \mathbb{Z}_2^{30} \times S_{30}$ from the natural interpretation that for some $(a_1, \dots, a_{30}, a') \in \mathbb{Z}_2^{30} \times S_{30}$ the a_i describe the orientation change to the edge in the i -th position and the a' describes the permutation of the edges.

Now let $a, b \in \hat{G}_E$ with $(a_1, \dots, a_{30}, a') := \varphi(a)$ and $(b_1, \dots, b_{30}, b') := \varphi(b)$ and consider their composition $((a \cdot b)_1, \dots, (a \cdot b)_{30}, (a \cdot b)') := \varphi(a \cdot b)$. The permutations of the pieces can just be composed naturally, so we have $(a \cdot b)' = a' \cdot b'$. However, when we want to determine $(a \cdot b)_i$ for some $i \in [30]$, we have to keep in mind that b permutes the edge in the i -th position to $i^{b'}$, so we have to compose $a_{i^{b'}} \cdot b_i$ instead of $a_i \cdot b_i$ to obtain $(a \cdot b)_i$. Since this describes the action of the wreath product, we can conclude that φ is an isomorphism from $\hat{G}_E$ to $\mathbb{Z}_2 \wr_{[30]} S_{30}$. $\square$

Interestingly, we can decompose G_M as well. Figure 4.2 depicts a Megaminx state for which the corners are solved and the edges have been permuted according to a U move. Giving this state to the solving algorithm that was developed in the scope of this thesis, we obtain the move sequence

$$R' U' R2 U2' R' U R U' R U2' R U2' R' U2 R' U2 R2 U R2 U R' \quad (4.3)$$

as a solution, from which we conclude that this permutation is an element of the Megaminx group. On the other hand, we can append a U move to the end of this sequence to obtain a U move for only corners, depicted in Figure 4.3.

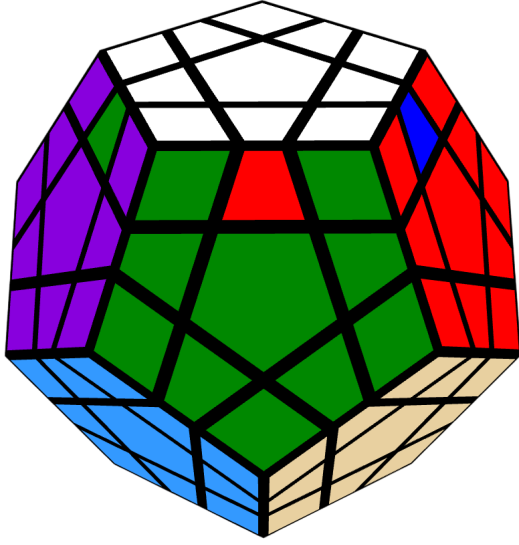


Figure 4.2: U move for edges

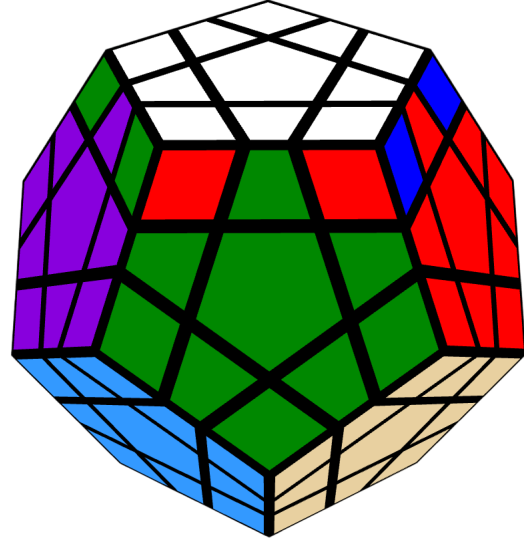


Figure 4.3: U move for corners

Since this can be done for all moves, the edges and corners of the Megaminx are also independent.

Theorem 62.

Suppose $G_E < G_M$ is the subgroup of edge-sticker permutations and $G_C < G_M$ is the subgroup of corner-sticker permutations; then it holds that

$$G_M = G_E \times G_C. \quad (4.4)$$

Note that this is not true for the Rubik's Cube, as can be verified, for example, with GAP. Since G_C and G_E are subgroups of $\hat{G}_C$ and $\hat{G}_E$, respectively, we immediately obtain the following theorem.

Theorem 63.

Up to isomorphism it holds that $G_E < \mathbb{Z}_2 \wr_{[30]} S_{30}$ and $G_C < \mathbb{Z}_2 \wr_{[20]} S_{20}$.

This can be strengthened, since we know from Theorem 59 that the permutation groups of corners and edges are even groups.

Theorem 64.

Up to isomorphism it holds that $G_E < \mathbb{Z}_2 \wr_{[30]} A_{30}$ and $G_C < \mathbb{Z}_2 \wr_{[20]} A_{20}$.

Corollary 65.

Up to isomorphism it holds that $G_M < (\mathbb{Z}_2 \wr_{[30]} A_{30}) \times (\mathbb{Z}_2 \wr_{[20]} A_{20})$.

We further obtain from Theorem 59 that if the orientation of all but one of the edges is determined, then this determines the orientation of the last edge, and the same can be said for corners.

4.2 A Lower Bound on the Diameter of the Megaminx Group

Theorem 47 yields a lower bound of 40 for the diameter of the Megaminx Group with respect to the standard generating set S_M . This theorem was established through the fact that the number of group elements with a word length of at most $l \in \mathbb{N}_0$ is naturally bounded by the number of words of length at most l . We can therefore improve this lower bound by reducing the number of equivalent words counted. However, we have to make sure that we always count at least one minimal word of each group element, as the recursive argument of Theorem 47 ceases to work otherwise. This optimization approach is especially helpful in cases where equivalence of words can be determined without actually having to compute their permutations.

As an example, we can consider the equivalence between the move sequences $U U$ and U^2 , which stems from the fact that moves on the same face can be summarized. Another example is the equivalence between the move sequences $U D$ and $D U$, which is rooted in the non-adjacency of the faces D and U , which causes moves on them to commute pairwise. We can define a reduced set of move sequences, where each move sequence avoids consecutive moves on the same face and has a unique order of consecutive commuting moves.

Definition 66 (Canonical Sequences).

Let $<$ be the strict total order

$$U < R < F < L < BR < BL < FR < FL < DR < DL < B < D \quad (4.5)$$

on F and let f be the map that maps a move to its face, then we denote by $c(G_M)$ the set

$$\{s_1 \cdots s_l \in W_{S_M} \mid f(s_i) < f(s_{i+1}) \text{ for all } i \in [l-1] \text{ with } \phi(s_i s_{i+1}) = \phi(s_{i+1} s_i)\} \quad (4.6)$$

of canonical move sequences.

As stated before, we have to show that this set still contains at least one minimal word of each group element.

Theorem 67.

For every $m \in G_M$ there is a minimal word of m in $c(G_M)$.

Proof. For every $m \in G_M$ we can choose a minimal word $w = s_1 \cdots s_l$ of m that is also minimal with respect to the lexicographic ordering implied by $<$. We can now assume $w \notin c(G_M)$; then there is some $i \in [l-1]$ such that s_i and s_{i+1} commute and $f(s_i) \geq f(s_{i+1})$ holds. In case $f(s_i) = f(s_{i+1})$, the moves s_i and s_{i+1} are on the same face, and there exists a move s with $\phi(s) = \phi(s_i s_{i+1})$, which contradicts the minimality of w . Hence, we have $f(s_i) > f(s_{i+1})$, and we can transform w into a word w' by swapping s_i and s_{i+1} in w . Since s_i and s_{i+1} commute and because w' has the same length as w , we know that w' is also a minimal word of m . However, with respect to the lexicographic ordering implied by $<$, we have $w' < w$, which is a contradiction. $\square$

Corollary 68.

For $n \in \mathbb{N}_0$, let $c_n(G_M)$ be $c(G_M)$ restricted to move sequences of length n ; then

$$\min\{l \in \mathbb{N}_0 \mid \sum_{n=0}^l |c_n(G_M)| \geq |G_M|\} \quad (4.7)$$

is a lower bound on the diameter of G_M with respect to S_M .

We can recursively compute $|c_n(G_M)|$, which allows us to obtain an improved lower bound.

Theorem 69.

The diameter of the Megaminx Group is at least 47.

Proof. For $f \in F$ and $n \in \mathbb{N}$, let $c_n^f(G_M) \subseteq c_n(G_M)$ be the set of move sequences of length n that end with a move on f , and let $F_f \subseteq F$ be the set of faces f' such that f and f' are adjacent or satisfy $f' < f$. Then we have $c_n^f(G_M) = \bigcup_{f' \in F_f} c_{n-1}^{f'}(G_M) \cdot \{f^i \mid i \leq \text{ord}(f)\}$ for $n \geq 2$, and since every f has order 4, we can conclude $|c_n^f(G_M)| = 4 \sum_{f' \in F_f} |c_{n-1}^{f'}(G_M)|$. We also have $|c_1^f(G_M)| = 4$ for each $f \in F$, which allows us to compute $|c_n^f(G_M)|$ for all $n \geq 1$ and $f \in F$. Additionally, we have $c_0(G_M) = 1$ and $|c_n(G_M)| = \sum_{f \in F} |c_n^f(G_M)|$ for all $n \in \mathbb{N}$, which allows us to compute $|c_n(G_M)|$ for all $n \in \mathbb{N}$ with dynamic programming. Computing $|c_n(G_M)|$, we obtain

$$2.25 \cdot 10^{67} \approx |c_{46}(M)| < |G_M| < |c_{47}(M)| \approx 6.45 \cdot 10^{68}, \quad (4.8)$$

which yields the desired bound. $\square$

We can choose a different order of F to see if the result changes. For example, we can choose

$$U < D < R < DL < F < B < L < DR < BR < FL < BL < FR, \quad (4.9)$$

which gives

$$1.09 \cdot 10^{67} \approx |c_{45}(M)| < |G_M| < |c_{46}(M)| \approx 3.34 \cdot 10^{68}. \quad (4.10)$$

Hence, the obtained bound is not independent of $<$ and testing more orders could possibly yield a better bound. A similar approach was independently found and used by C. H. Liang in [24] to establish a lower bound for the God's Number of the Kilominx, a Megaminx variant without centers and edges. We want to mention here that a lower bound of 48 has been argued for in [25] by expanding the considered equivalency beyond consecutive moves. However, we will not go into detail with this approach, as that would go beyond the scope of this thesis.

5 A Naive Solution to the Factorization Problem of Permutation Groups

In this chapter we develop a first naive algorithm to solve the Factorization Problem of Permutation Groups 13 (FPPG) in the computational framework of Section 3.8. Since we assumed for any generating set S to be finite, we can enumerate the words over S and test them individually for equality with the given group element g . We have that $|S| = 0$ implies $G = \{e\}$, and since generating sets are closed under inverses, $|S| = 1$ implies $|G| = 2$. Because FPPG is trivial in those cases, we can assume $m := |S| > 1$. Now let $S = \{s_1, \dots, s_m\}$; then we can enumerate words over S like we enumerate numbers in base $|S|$ with the pattern

$$\epsilon, s_1, s_2, \dots, s_m, s_1 s_1, s_1 s_2, \dots, s_1 s_m, s_2 s_1, \dots, s_m s_m, s_1 s_1 s_1, \dots \quad (5.1)$$

Until we find a word of g , we have to compute $\phi(w)$ for each word w that we generate in this sequence, and if w is of length $l \in \mathbb{N}_0$, this computation is generally in $\Theta(l)$. This runtime can be improved by utilizing the fact that our enumeration is lexicographic for words of equal length, and hence we are mostly only updating the rightmost letters, which allows us to reduce the number of compositions we have to compute.

5.1 Composing Words

The number of letters we have to update for the next word in the sequence depends on the number of letters from the right that are equal to s_m .

Definition 70.

For a word $w := s_{i_1} \cdots s_{i_l}$ of length $l \in \mathbb{N}_0$, we define $n(w)$ to be the number

$$\max\{n \in [l]_0 \mid s_{i_j} = s_m \text{ for all } l - n < j \leq l\} \quad (5.2)$$

of consecutive s_m from the right in w .

Suppose now that we are currently at the word $w = s_{i_1} \cdots s_{i_l}$ of length $l \in \mathbb{N}_0$ and we have already computed the respective group element $\phi(w)$. We can determine the next word w' in the enumeration by updating the first $n(w)$ letters from the right to s_1 and $s_{i_{l-n(w)}}$ to $s_{i_{l-n(w)}+1}$. Hereby we use the convention $i_0 := 0$ and $s_0 := e$, so the word s_m^l is updated to s_1^{l+1} . Note here that we have $i_{l-n(w)} < m$ by definition of $n(w)$, so this definition is well-defined. Now we obtain the group element of w' by computing

$$\phi(w') = \phi(w) \cdot \phi(s_m)^{-n(w)} \cdot \phi(s_{i_{l-n(w)}})^{-1} \cdot \phi(s_{i_{l-n(w)}+1}) \cdot \phi(s_1)^{n(w)}. \quad (5.3)$$

We formalize this with the pseudocode given in Algorithm 1.

Algorithm 1 NextWord

Require: Generating set S of group G , word $w = s_{i_1} \cdots s_{i_l}$ over S , group element $g = \phi(w)$.

Ensure: Next word $w' = s'_{i_1} \cdots s'_{i'_l}$ in Enumeration 5.1, group element $g' = \phi(w')$.

- 1: Determine $n(w)$
 - 2: $w' \leftarrow s_{i_1} \cdots s_{i_{l-n(w)-1}} s_{i_{l-n(w)}+1} s_1^{n(w)}$
 - 3: $g' \leftarrow g \cdot \phi(s_m)^{-n(w)} \cdot \phi(s_{i_{l-n(w)}})^{-1} \cdot \phi(s_{i_{l-n(w)}+1}) \cdot \phi(s_1)^{n(w)}$
 - 4: **return** w', g'
-

Lemma 71.

For a word w and its group element $\phi(w)$, Algorithm 1 computes the successor w' of w in Enumeration 5.1 along with $\phi(w')$ with a runtime in $\Theta(n(w))$.

Proof. We have already argued correctness, so we are only left with determining the runtime. Line 1 reads w from the right until it finds a letter that is not s_m , which is in $\Theta(n(w))$. Generating w' in Line 2 requires us to update $n(w) + 1$ letters, which is also in $\Theta(n(w))$. Lastly, we have to compute g' , which requires $2n(w) + 2$ compositions and 2 inversions and is hence also in $\Theta(n(w))$. Therefore, we get the total runtime to be in $\Theta(n(w))$. $\square$

While it would not improve the resulting runtime class, we can precompute powers of $\phi(s_m)^{-1}$ and $\phi(s_1)$, in which case Line 3 requires at most 4 compositions.

5.2 The Naive Algorithm

We now just iterate through Enumeration 5.1 with Algorithm 1 until we find a word of g . A pseudocode is given in Algorithm 2.

Algorithm 2 A Naive Algorithm

Require: Generating set S of group G , element $g \in G$

Ensure: Minimal word w of g over S

```
1:  $w \leftarrow \epsilon$ 
2:  $g_w \leftarrow e$ 
3: while  $g_w \neq g$  do
4:    $(w, g_w) \leftarrow \text{NextWord}(S, w, g_w)$ 
5: end while
6: return  $w$ 
```

Lemma 72.

Algorithm 2 solves FPPG of g , and the resulting word is minimal.

Proof. Since there always exists a word of g , such a word has to be generated eventually, terminating the algorithm. The returned word has to be minimal because words of shorter length are generated first. $\square$

The runtime of Algorithm 2 depends on the number of words generated until termination, and that number is generally not known in advance. However, if we can make an estimate on the word length of g , then we get at least some idea of the runtime.

Lemma 73.

For a word g with word length $l \in \mathbb{N}_0$, the runtime of Algorithm 2 is in $\Theta(|S|^l)$.

To prove this, we need a quick lemma to help us with the calculations.

Lemma 74.

Let $l \in \mathbb{N}_0$, and suppose W_S^l is the set of words over S of length at most l ; then it holds that

$$\sum_{w \in W_S^l} n(w) = l + (|S| - 1) \sum_{k=0}^{l-1} k |S|^{l-k-1}. \quad (5.4)$$

Proof. Let $k \in [l]_0$, and let c_k^l be the number of words of length l with $n(w) = k$. By definition of $n(w)$, words $w = s_{i_1} \cdots s_{i_l}$ counted by c_k^l satisfy $s_{i_j} = s_m$ for $j > l - k$ and $s_{i_{l-k}} \neq s_m$. Since each remaining s_{i_j} can be an arbitrary element from S , we can conclude that $c_1^l = 1$ and $c_k^l = (|S| - 1) |S|^{l-k-1}$ for $k \in [l - 1]_0$. With that we already obtain

$$\sum_{w \in W_S^l} n(w) = \sum_{k=0}^l k c_k^l = l + (|S| - 1) \sum_{k=0}^{l-1} k |S|^{l-k-1}. \quad (5.5)$$

$\square$

Proof of Lemma 73. The first two lines are in $\Theta(1)$. Line 3 performs an elementary action, and according to Lemma 71, Line 4 runs in time $\Theta(n(w))$, so each iteration of the while loop is in $\Theta(n(w))$.

In the worst case we have to generate and test all words of length at most l until a word of g is found, so with Lemma 74 we get a runtime in

$$\mathcal{O} \left(l + (|S| - 1) \sum_{k=0}^{l-1} k |S|^{l-k-1} \right) = \mathcal{O}(|S|^l). \quad (5.6)$$

Similarly, in the best case we have to generate and test all words of length at most $l - 1$, as well as one word w of length l . That word w would satisfy $n(w) = 0$, and we again have with Lemma 74 a runtime in

$$\Omega \left(l - 1 + (|S| - 1) \sum_{k=0}^{l-2} k |S|^{l-k-2} \right) = \Omega(|S|^l). \quad (5.7)$$

Hence, we have a runtime in $\Theta(|S|^l)$. □

We can optimize this approach by skipping words that do not represent the canonical move sequences defined in Theorem 66. This would allow us to omit some computations of group elements, as the property of being canonical can be extracted from a word without the computation of its group element. Since this approach would still be too slow, we now take a look at a different idea.

6 Breadth-First-Search Approach

In Theorem 45 we have seen that FPPG is equivalent to the problem of finding paths in the Cayley graph, in which case path lengths translate to word lengths. Since the Cayley graph is unweighted, we can therefore utilize a BFS algorithm to help us out. A pseudocode inspired by [26, Section 20.2] is given in Algorithm 5, where we first generate depths of all nodes in the Cayley graph with Algorithm 3 and then use those depths to find a shortest path with backtracking in Algorithm 4.

Algorithm 3 GenDepths

Require: Generating set S of group G

Ensure: Group elements of G with their respective depths

```
1:  $e.depth \leftarrow 0$ 
2:  $G \leftarrow \{e\}$ 
3:  $Q \leftarrow \emptyset$ 
4:  $Q.enqueue(e)$ 
5: while  $Q \neq \emptyset$  do
6:    $v \leftarrow Q.dequeue$ 
7:   for  $s \in S$  do
8:      $u \leftarrow v \cdot s$ 
9:     if  $u \notin G$  then
10:        $u.depth \leftarrow v.depth + 1$ 
11:        $G.add(u)$ 
12:        $Q.enqueue(u)$ 
13:     end if
14:   end for
15: end while
16: return  $G$ 
```

Algorithm 4 Backtrack

Require: Group G with depths of all elements, generating set S of G , $g \in G$

Ensure: Path from e to g in the Cayley graph

```
1: if  $g.depth == 0$  then
2:   return  $\epsilon$ 
3: else
4:   for  $s \in S$  do
5:      $u \leftarrow g \cdot s^{-1}$ 
6:     if  $g.depth > u.depth$  then
7:       return  $Backtrack(G, S, u)s$ 
8:     end if
9:   end for
10: end if
```

Algorithm 5 BFS on a Cayley Graph

Require: Generating set S of group G , $g \in G$

Ensure: Minimal word of g over S

```
1: return  $Backtrack(GenDepths(S), S, g)$ 
```

Lemma 75.

Algorithm 5 finds a minimal word of g over S .

Proof. Since Cayley graphs are connected, there is always a path between any two nodes, and therefore the proof follows from [26, Theorem 20.5]. $\square$

For a group element with word length $l \in \mathbb{N}_0$, we can again prove a runtime in $\mathcal{O}(|S|^l)$ [27, Section 3.4.1]; however, we can also find a runtime depending only on $|G|$ and $|S|$. This is generally a more telling estimate, because $|S|$ is already known and $|G|$ can be computed efficiently, as seen in Theorem 55, while word lengths are currently not known to be efficiently computable [2, Section 4.8.3].

Lemma 76.

For a finite group G with generating set S , the runtime of Algorithm 5 is in $\mathcal{O}(|G||S|)$.

Proof. Let g be a word with word length $l \in \mathbb{N}_0$; then the analysis in [26, Section 20.2] yields a runtime in $\mathcal{O}(|G| + |E_S|)$ for Algorithm 3 and a runtime in $\mathcal{O}(|S|l)$ for Algorithm 4. Since we have $|E_S| = |S||G|$ by definition of Cayley graphs and $l < |G|$, we obtain a runtime of Algorithm 5 in

$$\mathcal{O}(|G| + |E_S| + |S|l) = \mathcal{O}(|G| + |S||G| + |S|l) = \mathcal{O}(|S||G|). \quad (6.1)$$

□

Unfortunately, the runtime of Algorithm 5 is still far from feasible for the Megaminx Group, and we have also added a space requirement linear in $|G|$.

7 Multi-Phasing

In order to improve upon the BFS approach, we can utilize a divide-and-conquer technique to split up the solution process. Let $H \leq G$ be some subgroup with generators given as words over S , in which case words over H are also words over S . Then we first search for a word w_1 with $\phi(w_1) \in Hg$ and then solve FPPG for $g \cdot \phi(w_1)^{-1}$ over H to obtain a word w_2 with $\phi(w_2) = g \cdot \phi(w_1)^{-1}$. Because ϕ is a homomorphism, this is equivalent to $\phi(w_2w_1) = g$, and since w_2 is a word over H , it is also a word over S . We can conclude that w_2w_1 is a solution to FPPG of g .

The word w_1 can be found by utilizing our BFS approach on the coset graph. Since the coset graph has $[G : H]$ nodes and the Cayley graph of H has $|H|$ nodes, we can expect a runtime in $\mathcal{O}((|H| + [G : H])|S|)$. From Lagrange's Theorem 18 we have $|G| = |H|[G : H]$, so this is a substantial improvement if H is a proper subgroup, especially if it has an order close to $\sqrt{|G|}$.

We can utilize this approach again by choosing a subgroup of H whose generators are given as words over the generating set of H , which ensures that the resulting word remains a word over S . Hence, the process can be iterated. This approach is also used when FPPG is solved with the Schreier-Sims Algorithm, in which case the subgroup chain consists of pointwise stabilizers [21, Section 4.1]. Since we know that the Schreier-Sims Algorithm produces rather long words, we can conclude that we need to choose these subgroups and their generators carefully in order to obtain shorter words.

7.1 Solving the Word Problem on Cosets

First of all, we need to generalize FPPG.

Definition 77 (The Factorization Problem of Cosets).

Let G, H be groups with $H \leq G$, S be a generating set of G , and Ω be the set of cosets of H in G . For some $\omega \in \Omega$ we call the task of finding a word w over S , such that $\phi(w) \in \omega$ holds the Factorization Problem of Cosets (FPC).

For $H = \{e\}$ we obtain FPPG, so this is indeed a generalization. We now want to design a generalized version of Algorithm 5 that works on coset graphs. A pseudocode is given in Algorithm 8.

Algorithm 6 GenDepthsCoset

Require: Generating set S_G of group G , generating set S_H of subgroup $H \leq G$

Ensure: Depths of all cosets $\omega \in \Omega$

```

1:  $He \leftarrow \text{compCoset}(S_G, S_H, e)$ 
2:  $He.depth \leftarrow 0$ 
3:  $\Omega \leftarrow \{He\}$ 
4:  $Q \leftarrow \emptyset$ 
5:  $Q.enqueue(e)$ 
6: while  $Q \neq \emptyset$  do
7:    $v \leftarrow Q.dequeue$ 
8:   for  $s \in S_G$  do
9:      $u \leftarrow v \cdot s$ 
10:     $Hu \leftarrow \text{compCoset}(S_G, S_H, u)$ 
11:    if  $Hu \in \Omega$  then
12:       $Hv \leftarrow \text{compCoset}(S_G, S_H, v)$ 
13:       $Hu.depth \leftarrow Hv.depth + 1$ 
14:       $\Omega.add(Hu)$ 
15:       $Q.enqueue(u)$ 
16:    end if
17:  end for
18: end while
19: return  $\Omega$ 

```

Algorithm 7 BacktrackCosets

Require: Set of cosets Ω with depths of all elements, generating set S_G of group G , generating set S_H of subgroup $H \leq G$, $g \in G$

Ensure: Path from Hg to H in the coset graph

```
1:  $Hg \leftarrow \text{compCoset}(S_G, S_H, g)$ 
2: if  $Hg.\text{depth} == 0$  then
3:   return  $\epsilon$ 
4: else
5:   for  $s \in S$  do
6:      $u \leftarrow g \cdot s^{-1}$ 
7:      $Hu \leftarrow \text{compCoset}(S_G, S_H, u)$ 
8:     if  $Hg.\text{depth} > Hu.\text{depth}$  then
9:       return  $\text{BacktrackCosets}(\Omega, S_G, S_H, u)s$ 
10:    end if
11:  end for
12: end if
```

Algorithm 8 BFSCoset

Require: Generating set S_G of group G , generating set S_H of subgroup $H \leq G$, element $g \in G$

Ensure: Word w over S with $\phi(w) \in Hg$

```
1: return  $\text{BacktrackCos}(\text{GenDepthsCoset}(S_G, S_H), S_G, S_H, g)$ 
```

Lemma 78.

Algorithm 8 solves FPC, and the resulting word is minimal.

Proof. This follows from [26, Theorem 20.5], since Algorithm 8 performs a BFS on the coset graph and because coset graphs are connected. $\square$

Lemma 79.

For a finite group G with generating set S and a subgroup H , the runtime of Algorithm 8 is in $\mathcal{O}([G : H]|S|)$.

Proof. Since the computation of a coset is an elementary action according to Assumption 54, this follows similarly to Lemma 76 from the fact that the coset graph has $[G : H]$ nodes. $\square$

Similarly to Algorithm 5, we have a space requirement linear in $[G : H]$.

It is important to note that the minimality of obtained words is with respect to the number of elements

from S_G . When we apply Algorithm 5 to one of the subgroups, then the elements of S_G are given as words over the original generating set S , and so they can have varying word lengths over S . In order to obtain words that are minimal with respect to S , we either have to choose subgroups generated by a subset of S or we can work on a weighted coset graph, where the weight of an edge is given by its length as a word over S . These weights are nonnegative integers, and since S_G is finite, the weights are also bounded from above by some $B \in \mathbb{N}$. In such a case we can utilize a modified version of BFS called Dial's Algorithm to obtain a runtime in $\mathcal{O}([G : H]B + |E_S|) = \mathcal{O}([G : H](B + |S|))$ [28, Section 1.3.2].

7.2 The Multi-Phase Algorithm

Let $G = G_0 > \dots > G_n = \{e\}$ be a strictly decreasing subgroup chain, where each G_i is given by a generating set S_i , and let all generators be given as words over S_0 . We can now iteratively apply Algorithm 8 in order to obtain a solution to FPPG. A pseudocode can be seen in Algorithm 9.

Algorithm 9 A Multi Phase Algorithm

Require: Subgroup chain $G_0 > \dots > G_n = \{e\}$ with each G_i given by a generating set S_i , $g \in G$

Ensure: Word w of g over S_0

```

1:  $w \leftarrow \epsilon$ 
2: for  $i \leftarrow 1$  to  $n$  do
3:    $w' \leftarrow \text{BFSCoset}(S_{i-1}, S_i, g)$ 
4:    $w \leftarrow w'w$ 
5:    $g \leftarrow \phi(w') \cdot g$ 
6: end for
7: return  $w$ 

```

Theorem 80.

Algorithm 9 solves the Factorization Problem of Permutation Groups 13.

Proof. From Lemma 78 we know that for $g \in G_{i-1}$ Algorithm 8 computes a word w_i with $\phi(w_i) \in G_i g$, which is equivalent to $\phi(w_i) \cdot g^{-1} \in G_i$. By induction over i we obtain for the final word $w = w_n \dots w_1$ that $\phi(w) \cdot g^{-1} \in G_n$ holds, which is equivalent to $\phi(w) = g$. $\square$

Theorem 81.

The runtime of Algorithm 9, measured in elementary actions, is in $\mathcal{O}\left(\sum_{i=1}^n [G_{i-1} : G_i] |S_{i-1}|\right)$.

Proof. This immediately follows from Lemma 79. $\square$

This is a very strong result, as we know from Lagrange's Theorem that $|G| = \prod_{i=0}^{n-1} [G_i : G_{i+1}]$ holds, so the runtime essentially decomposes into the indices of our subgroup chain.

For the Rubik's Cube, an example of such an algorithm is given by the Thistlethwaite Algorithm [29], which was already developed in 1981 and established an upper bound of 52 on the diameter of the Rubik's Cube Group. The 4 phases of Thistlethwaite's Algorithm were later refined by H. Kociemba into a 2-phase algorithm that was utilized in [7] to prove the diameter of the Rubik's Cube Group to be exactly 20, as has been briefly discussed in Section 2.2.

8 The Megaminx Solver

In order to apply Algorithm 9, we need to find a suitable subgroup-chain and construct the *compCoset* function whose existence we assumed in Assumption 54. We will also give some insight on how solutions generated by our solver solve the puzzle from a more intuitive angle, and we will give an example for which our solver cannot generate an optimal solution.

8.1 Subset-Chain

As discussed in Section 7.1, we either have to choose subgroups generated by a subset of the Megaminx Group's standard generating set or we have to modify Algorithm 8 to account for nonnegative integer edge weights. For the sake of simplicity, we go with subgroups generated by subsets of S_M . Since each move has an order of 5, which is a prime, the moves on a face generate each other. Hence, only removing some of a face's moves has no effect on the obtained subgroup and just leaves us with fewer generators. Since that can potentially increase the length of obtained solutions, all moves on a face are always removed at once. We make an ansatz by letting

$$(s_1, \dots, s_{12}) := (U, R, F, L, BR, BL, FR, FL, DR, DL, B, D) \quad (8.1)$$

be an enumeration of the Megaminx's faces and defining generating sets $S_i := \{s_i^n \mid s_i > i \text{ and } n < \text{ord}(s_i)\}$ along with the groups $G_i := \langle S_i \rangle$ for all $i \in [n]_0$. Essentially, we start by removing all moves on the D -face, then all moves on the B -face, and so on, until there are no moves left.

In order to determine if this chain yields a feasible multi-phase algorithm, we want to use Lemma 81, for which we have to compute the respective indices of our subgroup chain. These can be obtained by computing the subgroup's orders in GAP and applying Lagrange's Theorem. The results can be seen in Table 8.1. Unfortunately the index $[G_{10} : G_{11}]$ is too big for us to work with, so we balance it out with the index $[G_{11} : G_{12}]$ by modifying the generating set S_{11} to

$$S'_{11} := \{U, U2, U2', U', RUR', RU2R', RU2'R', RU'R'\}. \quad (8.2)$$

This set generates a group G'_{11} for which we obtain the updated values seen in Table 8.2. These values are now manageable, despite the fact that we have to resort to Dial's Algorithm for the resulting phase. We are left with the subgroup chain

$$G_0 > G_1 > G_2 > G_3 > G_4 > G_5 > G_6 > G_7 > G_8 > G_9 > G_{10} > G'_{11} > G_{12}. \quad (8.3)$$

Table 8.1: Subgroup data

	$ G_i $	$[G_{i-1} : G_i]$
G_0	$1.01 \cdot 10^{68}$	-
G_1	$1.01 \cdot 10^{68}$	1
G_2	$1.68 \cdot 10^{66}$	60
G_3	$8.61 \cdot 10^{60}$	194,880
G_4	$5.38 \cdot 10^{55}$	160,056
G_5	$4.15 \cdot 10^{50}$	129,600
G_6	$1.99 \cdot 10^{42}$	208,099,584
G_7	$2.92 \cdot 10^{37}$	68,400
G_8	$4.54 \cdot 10^{29}$	64,157,184
G_9	$1.75 \cdot 10^{22}$	25,945,920
G_{10}	$8.00 \cdot 10^{12}$	2,189,721,600
G_{11}	5	1,599,935,016,960
G_{12}	1	5

Table 8.2: Updated subgroup data after replacing G_{11} with G'_{11}

	$ G_i $	$[G_{i-1} : G_i]$
G_{10}	$8.00 \cdot 10^{12}$	2,189,721,600
G'_{11}	31,492,800	254,016
G_{12}	1	31,492,800

8.2 Computing Cosets

We are left with the task of working out suitable encodings of all cosets that satisfy Assumption 54. It turns out that, aside from G_{10} and G'_{11} , we can characterize the subgroups in our chain as pointwise stabilizers.

Theorem 82.

For every $i \in [9]_0$, it holds that $\langle S_i \rangle = G_i = (G_M)_{\text{fix}(S_i)}$.

Proof. Since all the elements in G_i have words over S_i , we know that they fix all elements in $\text{fix}(S_i)$, so $G_i \leq (G_M)_{\text{fix}(S_i)}$ holds true. Computing the orders of the groups $(G_M)_{\text{fix}(S_i)}$ in GAP and comparing them with the order of the G_i given in Table 8.1 now yields the desired equality. $\square$

Hence, so far the concept is akin to that of the strong generating sets used for the Schreier Sims Algorithm [21, Section 4.1], with the difference being that we fix multiple points simultaneously.

Corollary 83.

For every $i \in [9]_0$ and $g \in G_i$, it holds that $G_{i+1}g = \{h \in G_i \mid \alpha^g = \alpha^h \text{ for all } \alpha \in \text{fix}(S_i)\}$.

Therefore, we can encode $G_{i+1}g$ by storing the α^g . Determining α^g is an elementary action, as it can be extracted directly from the encoding of g , and we can store each α^g in $\mathcal{O}(\log(n))$. Since $\text{fix}(S_i) \subseteq \Omega$, we have at most n elements to compute and store, which yields the desired properties for this encoding.

To establish a similar result for the remaining subgroups, we need an additional criterion. We can compute the orbits of G_{10} , for example, with GAP, which gives the following result.

Theorem 84.

The set of non-fixed edge stickers in G_{10} decomposes into two orbits.

This allows us to characterize the cosets of G_{10} .

Theorem 85.

Let Δ be one of the non-trivial edge sticker orbits of G_{10} ; then we have $\langle S_{10} \rangle = G_{10} = (G_M)_{\text{fix}(S_{10})} \cap (G_M)_{\{\Delta\}}$.

Proof. The relation $G_{10} \leq (G_M)_{\text{fix}(S_{10})}$ follows with the same argumentation as in the proof of Theorem 82, and we have $G_{10} \leq (G_M)_{\{\Delta\}}$ from Theorem 84. Hence, we have $G_{10} \leq (G_M)_{\text{fix}(S_{10})} \cap (G_M)_{\{\Delta\}}$ and we can compute the order of $(G_M)_{\text{fix}(S_{10})} \cap (G_M)_{\{\Delta\}}$ with GAP to compare to the order of G_{10} given in Table 8.1 to obtain the desired equality. $\square$

While we currently do not know how to efficiently compute group intersections and setwise stabilizers [2, Chapter 4.6], they can still be computed in this particular case in practice, because the group is small enough.

Corollary 86.

In the setting of Theorem 85, we have $G_{10}g = \{h \in G_9 \mid \alpha^g = \alpha^h \text{ for all } \alpha \in \text{fix}(S_{10}) \text{ and } \Delta^h = \Delta^g\}$.

We can conclude that $G_{10}g$ can be encoded by storing α^g for all $\alpha \in \text{fix}(S_{10})$ and storing Δ^g . Since there are at most n elements in Δ , we can compute Δ^g in $\mathcal{O}(n)$ and store Δ^g with space in $\mathcal{O}(n \log(n))$. Since we already established that we can suitably compute and store the α^g , this encoding again satisfies the required properties. With the exact same argumentation we obtain similar results for G'_{11} .

Theorem 87.

The set of non-fixed edge stickers in G_{11} decomposes into two orbits.

Theorem 88.

Let Δ be one of the non-trivial edge sticker orbits of G'_{11} from Theorem 87; then we have $\langle S'_{11} \rangle = G'_{11} = (G_M)_{\text{fix}(S'_{11})} \cap (G_M)_{\{\Delta\}}$.

Corollary 89.

In the setting of Theorem 88, we have $G'_{11}g = \{h \in G_{10} \mid \alpha^g = \alpha^h \text{ for all } \alpha \in \text{fix}(S_{11}) \text{ and } \Delta^h = \Delta^g\}$.

Hence, we found suitable encodings for all cosets that can be computed according to the conditions of Assumption 54, and we therefore obtain the final solving algorithm.

Theorem 90 (The Algorithm).

Algorithm 9 with the subset chain

$$G_0 > G_1 > G_2 > G_3 > G_4 > G_5 > G_6 > G_7 > G_8 > G_9 > G_{10} > G'_{11} > G_{12}$$

and the coset encodings according to Corollary 83, Corollary 86, and Corollary 89 yields a solving algorithm for the Megaminx.

8.3 The Intuitive Interpretation

Now that we have a solving algorithm, we want to gain a more intuitive understanding of how it works. Figure 8.1 depicts snapshots of a solution generated by our solving algorithm, where each snapshots shows a front and back view of the Megaminx. We can see that the algorithm roughly follows a layer-by-layer approach in which the Megaminx is solved from the bottom to the top. This is akin to solving methods utilized by humans for the Megaminx [30] and also the Rubik's Cube [31]. In our case, the puzzle is solved by building and then extending upon a block of solved pieces. These blocks are visible by the increasing number of matching stickers on the Megaminx. This technique is called *blockbuilding*, another example of which would be the Petrus Method [32], which is a solving method for the Rubik's Cube.

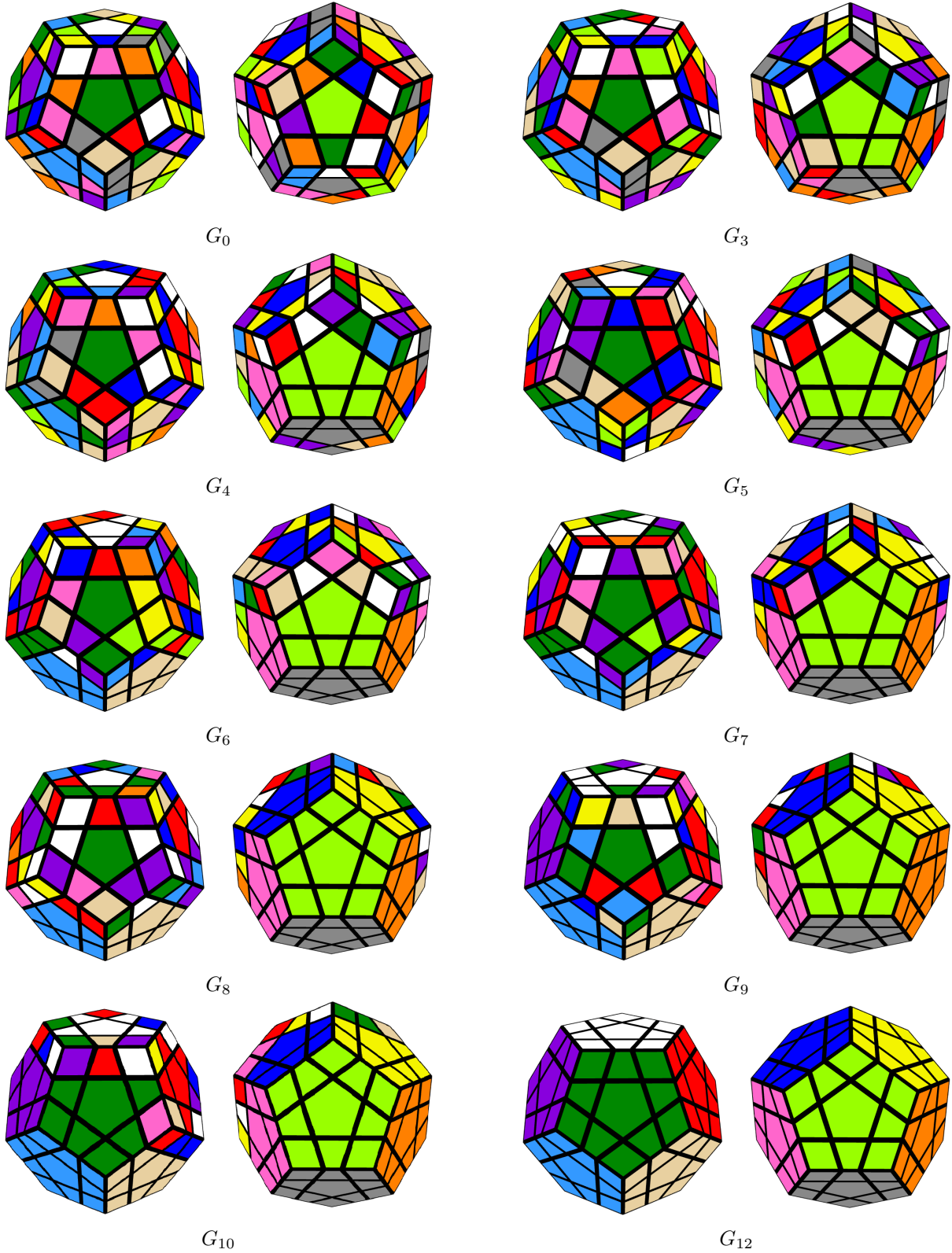


Figure 8.1: Visualization of a solution generated by our solver

An interesting observation we can make is that in G_{10} the remaining edges mostly have the white and red stickers already matching with the center sticker. This is not a coincidence, as we have determined in the previous section that the edge stickers decompose into two orbits in this subgroup. In this case, the orbit that includes all white stickers and the orbit that includes all red stickers are disjoint, so all remaining edges need to be suitably oriented. For example, we can consider Figure 8.2, which depicts a state with two flipped edges. As can be verified with GAP, the corresponding permutation is not an element in G_{10} , because the red and white stickers on those edges are in the wrong orbits.

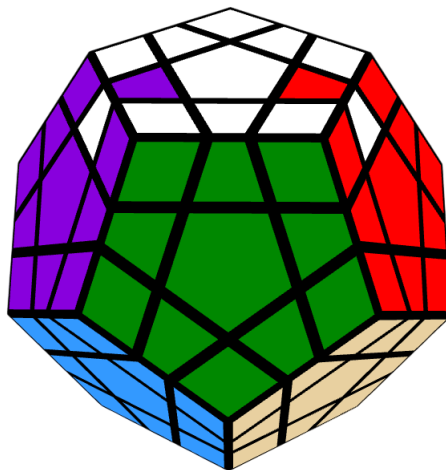


Figure 8.2: Two flipped edges

8.4 Suboptimality of Multi-Phase Solutions

We have mentioned before that solutions generated by this algorithm are not necessarily optimal, and we can now give an example of a Megaminx state for which our algorithm generates a suboptimal solution.

Example 91 (Suboptimality of the Multiphase).

Suppose we apply the move sequence $F R' F' R2$ to a solved Megaminx, which leaves us with the state depicted in Figure 8.3. The corresponding permutation of this state is an element in G_9 , since the move sequence is a word over S_9 . Algorithm 9 now finds an optimal move sequence that transforms this permutation into a permutation of G_{10} . By exhaustively generating all optimal solutions to this phase with our solving algorithm, we can show that $F R' F'$ is the only optimal solution. The resulting state is depicted in Figure 8.4 and cannot be solved in a single move; hence, the algorithm cannot find an optimal solution to the original state.

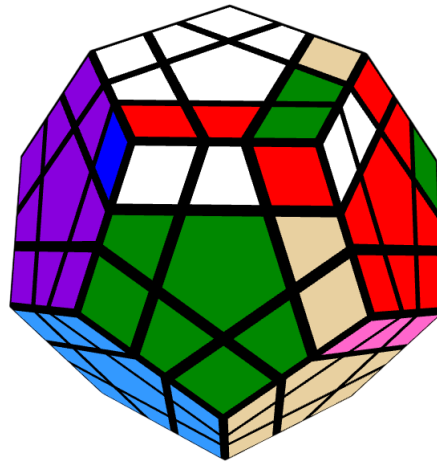


Figure 8.3: Solved state after applying $F R' F' R^2$

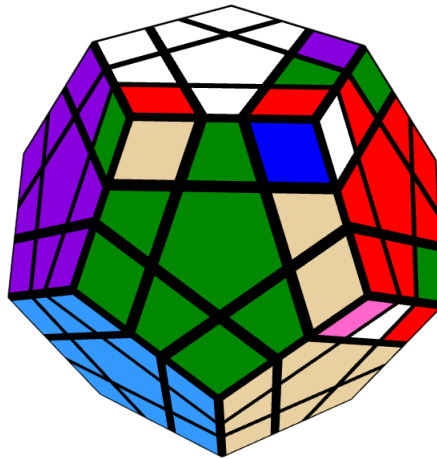


Figure 8.4: Solved state after applying $F R' F' R^2 F R' F'$

We can conclude that it is possible to find better solutions by allowing suboptimal solutions of the phases. This technique is used in Kociemba's Algorithm [7] for the Rubik's Cube and allows it to always find an optimal solution. However, we will not go more into detail with this particular optimization idea, as that would go beyond the scope of this thesis.

9 Implementation

In this chapter we discuss the implementation of our Megaminx solver, explaining details that are not clear from the theoretical part. We start with a more detailed discussion of the encoding of the Megaminx Group G_M and the respective cosets of our subgroup chain. Secondly, we explain how the runtime of our solver can be substantially reduced in practice by precomputing the depths computed in Algorithm 6. For practical reasons, we reduce the total number of phases by omitting the subgroups G_1 and G_2 from the subgroup chain. The resulting index $[G_0 : G_3] = 11,692,800$ is still relatively small, so this change does not cause any issues. This solver has been implemented in the programming language Go, the repository can be found in [33]. Since we will not be able to go into detail with all aspects of the implementation, we refer the interested reader to the repository.

9.1 Encoding the Megaminx

In Corollary 65 we found that the Megaminx Group is isomorphic to a subgroup of $(\mathbb{Z}_2 \wr_{[30]} A_{30}) \times (\mathbb{Z}_3 \wr_{[20]} A_{20})$, which yields a more natural type of encoding. In this encoding, the elements of the Megaminx Group are represented by the permutation of edges and corners, as well as the orientation change of each such piece. To properly identify the underlying isomorphism, we need to find a well-defined notion of piece orientations. For this purpose we can consider the solved state of the Megaminx and select one sticker of each piece to serve as a reference point. An example of such a selection is visualized in Figure 9.1, with two different views of the puzzle.

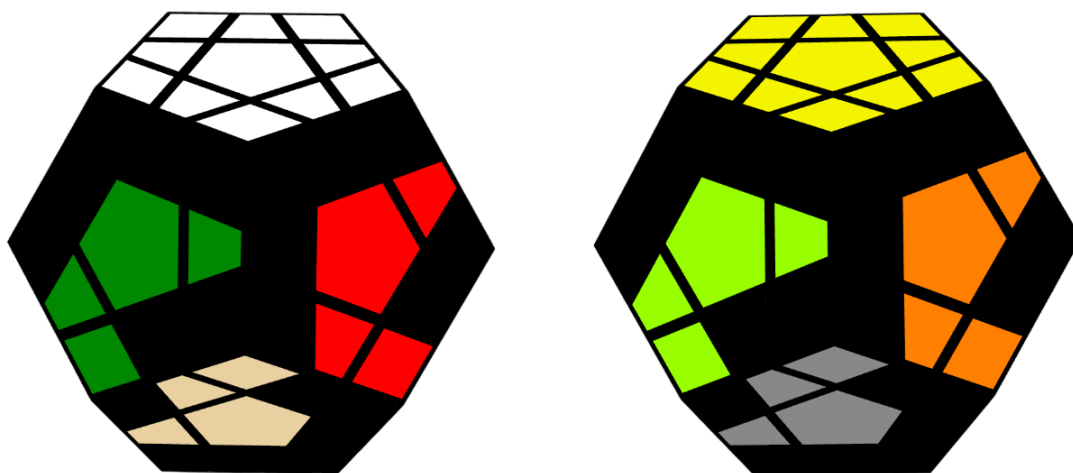


Figure 9.1: A selection of reference stickers serving as a choice of orientation for each piece

The orientation of a piece in an arbitrary position can now be defined by the position of its reference sticker relative to where the reference sticker of the piece in that position would be in the solved state. To better understand this, we can consider the example given in Figure 9.2, which depicts the reference stickers in the solved state and after applying an R to the solved state.

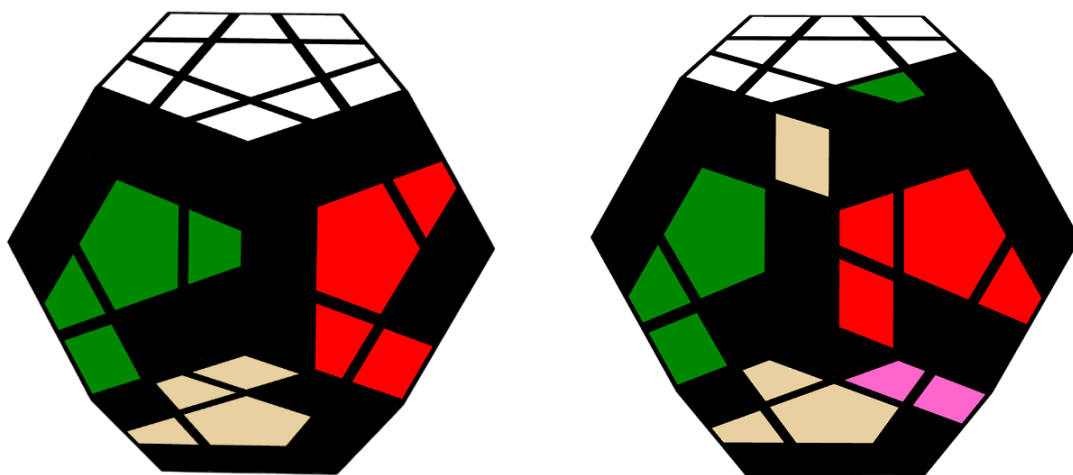


Figure 9.2: Reference stickers in the solved state and after applying an R to the solved state

The solved state represents the default positions of all reference stickers, so in that state all pieces have an orientation of 0. Now we can take a look at the state on the right, where we consider the edge on the front, between the green and red centers, as an example. In the solved state the reference sticker of that position points to the left, but now it points to the right, so that piece has an orientation of 1. To give another example,

we can consider the corner on the front of the U face, with the beige reference sticker. That sticker points to the left, but in the solved state the reference sticker of that position points upwards. Since we can make the beige sticker point upward by twisting that piece in place clockwise, we define its orientation as 2. If the beige sticker was on the right, we would require a counterclockwise twist, which we define as an orientation of 1. Note that this could also be defined the other way around.

The reference stickers that we choose have the practical advantage that moves on the U and D faces cause no orientation changes, and all clockwise moves of the remaining faces by 72° cause the same type of orientation changes, which makes for a more elegant computational model.

9.2 Coset Encodings

Having encoded the Megaminx via wreath products, we now need to understand how this affects the encoding of cosets that we have worked out in Section 8.2. Originally, we found that cosets can mostly be encoded by images of fixed points of the respective subgroups. Suppose $H \leq G_M$ is a subgroup of the Megaminx Group; then we call a piece a *fixed piece* of H if all of its stickers are fixed points of H . Note here that one sticker of a piece is fixed if and only if all stickers on that piece are fixed. Hence, the image of fixed stickers can equivalently be described by the orientation changes and images of all fixed pieces.

What remains is to work out how to encode the images of the respective non-trivial edge sticker orbits of G_{10} and G'_{11} , which we do exemplarily with G_{10} . One of the two non-trivial edge sticker orbits of G_{10} , denoted by Δ , is visualized in Figure 9.3, where we additionally colored the centers to make the puzzle's orientation more easily recognizable. We want Δ to be included in the set of reference stickers, so we redefine the selected reference stickers according to Figure 9.4. We note that this only changes the reference stickers of two edges on the red face.

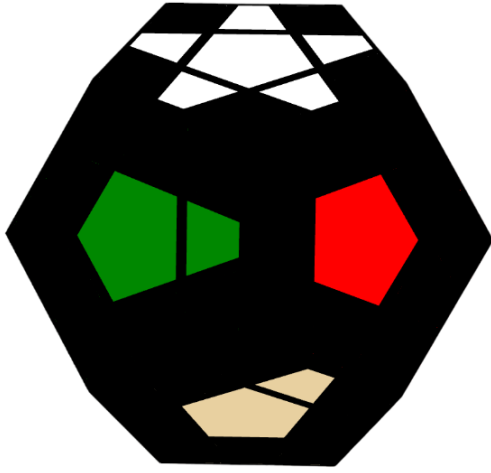


Figure 9.3: An edge sticker orbit Δ of G_{10}

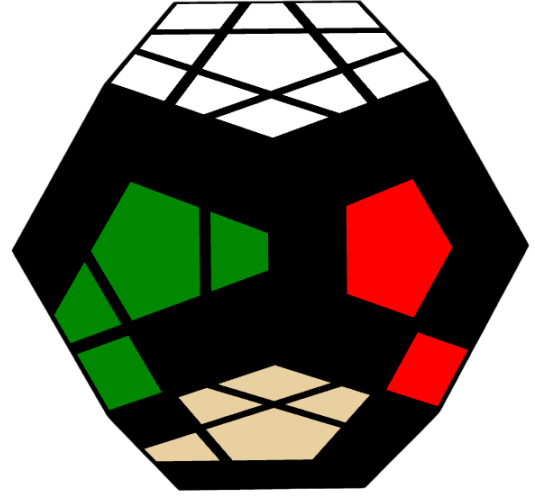


Figure 9.4: Reference sticker selection including Δ

Suppose $g \in G_9$ and consider the encoding of the coset $G_{10}g$. As explained before, this encoding already includes the images of all fixed edges, hence the remaining images are the images of non-fixed edges which are the positions in which non-fixed edges land under g . Since each non-fixed edge has exactly one sticker of Δ attached to it, we also know there is exactly one element of Δ^g in each position where such an edge lands. Which element that is, is determined by the orientation change of the edge in that position. Hence, we can describe Δ^g by the orientation changes of the (non-fixed) edges in these positions.

We have determined that the cosets can be encoded by images and orientation changes of a suitable selection of pieces, depending on the respective phase. The image of a fixed corner is given by an element in $[19]_0$; however, when $i \in [20]$ corners are already fixed, then there are only $20 - i$ possible images that the next corner can attain. Hence, the image of the i -th corner can also be described by an element in $[19 - i]_0$, and since the permutation group of corners is an even group, the image of the last two corners can be ignored. Each corner orientation change is described by an element in $[2]_0$, and if the orientation change to all but one corner is determined, then this determines the orientation change of the last corner, as seen in Section 4.1, so the orientation change of the last corner can be ignored. A similar argument can be made for edges, so we find that these coset encodings can be bijectively mapped to a Cartesian product of sets of the form $[n]_0$ with $n \in \mathbb{N}$. These elements can be naturally enumerated, which provides us with an encoding of cosets as elements in $[m - 1]_0$, where $m \in \mathbb{N}$ is the number of cosets of the respective phase. Such an encoding allows us to store the generated depths in an array, which is highly efficient.

9.3 Precomputations

Every time we generate a solution, we generate all depths for each coset graph. Since those depths do not depend on the input state, it suffices to compute and store them once. The coset encoding that we worked out in the previous chapter allows us to store the depths of all cosets in an array, which removes the factor of $n \log(n)$ in the space requirement of Algorithm 6. These arrays are usually called *lookup tables* and allow for very fast generation of solutions, since the running time is then dominated by the backtracking algorithm.

Lemma 92.

For a coset of depth $l \in \mathbb{N}_0$ and with the data of Algorithm 6 already precomputed, Algorithm 8 has a runtime in $\Theta(l|S|)$.

Proof. This immediately follows from the fact that the backtracking algorithm has a runtime in $\Theta(l|S|)$, as seen in [26, Section 20.2]. $\square$

A summarized version of each lookup table with distribution of depths, average depth, and number of cosets is given in Table 9.1. The depths of the last table $G'_{11} \rightarrow G_{12}$ are given with respect to the generating set S_M of the Megaminx Group and not with respect to S'_{11} . In particular, we immediately obtain an upper bound on the diameter of the Megaminx Group by adding up the maximum depths of each table.

Theorem 93.

The diameter of the Megaminx Group is at most 136.

Storing depths as bytes, these lookup tables require a total of 2.35 GB of storage, which can easily be kept in memory for very fast generation of solutions. Since depths could be stored in less than a byte, it would be possible to further reduce the required storage. However, since the current requirement is sufficiently low, we do not optimize this aspect in the scope of this thesis.

Table 9.1: Distribution of coset depths, total number of cosets, and average depth for each phase

Depth	$G_0 \rightarrow G_3$	$G_3 \rightarrow G_4$	$G_4 \rightarrow G_5$	$G_5 \rightarrow G_6$	$G_6 \rightarrow G_7$	$G_7 \rightarrow G_8$
0	1	1	1	1	1	1
1	12	4	4	4	4	4
2	168	24	24	24	24	24
3	2,214	278	262	282	246	254
4	26,466	2,346	2,112	2,811	1,784	2,180
5	256,292	14,944	12,995	26,054	9,956	18,229
6	1,668,044	49,632	41,955	217,512	28,721	137,208
7	5,125,524	66,712	52,747	1,643,810	25,133	920,459
8	4,249,463	24,960	18,659	9,851,214	2,531	4,936,030
9	364,457	1,155	841	38,231,916		16,847,646
10	159			77,848,865		26,679,561
11				65,554,807		13,470,216
12				14,362,567		1,141,888
13				359,505		3,484
14				212		
Total cosets	11,692,800	160,056	129,600	208,099,584	68,400	64,157,184
Average depth	7.23	6.62	6.57	10.15	6.23	9.78

Depth	$G_8 \rightarrow G_9$	$G_9 \rightarrow G_{10}$	$G_{10} \rightarrow G'_{11}$
0	1	1	1
1	4	4	4
2	24	32	12
3	226	244	48
4	1,687	1,628	142
5	13,045	11,464	528
6	89,562	78,738	1,840
7	545,812	525,693	6,015
8	2,657,519	3,367,902	16,382
9	8,141,704	19,845,430	40,516
10	10,580,277	101,072,749	74,594
11	3,749,592	387,376,731	80,878
12	166,350	834,011,052	30,580
13	117	678,742,974	2,362
14		155,114,098	114
15		9,332,294	
16		240,563	
17		3	
Total cosets	25,945,920	2,189,721,600	254,016
Average depth	9.56	12.16	10.18

Depth	$G'_{11} \rightarrow G_{12}$
0	1
1	4
3	4
4	32
5	64
7	64
8	508
9	978
11	978
12	7,582
13	14,665
15	14,639
16	112,897
17	216,489
19	214,318
20	1,592,735
21	2,808,647
23	2,519,641
24	12,644,566
25	8,504,103
27	2,111,473
28	728,287
29	125
Total cosets	31,492,800
Average depth	23.89

10 Experiments

Ideally, Algorithm 7 tests the elements of S in a random order, which makes our solver nondeterministic and introduces a certain level of variance. The goal of this chapter is to test this variance by repeatedly solving the same state of the Megaminx and to use these results to find a suitable test setup that allows us to compare different modifications of our solver.

To obtain accurate and consistent results, all tests in this chapter were performed with the Lichtenberg high-performance cluster (HPC) of the TU Darmstadt, utilizing 16 cores and 80 GB of memory. Furthermore, the Lichtenberg HPC was used to compute lookup tables bigger than 10 GB.

10.1 Developing a Test Setup

In Section 2.2 we have seen the Superflip-Configuration of the Rubik's Cube, in which all edges are twisted in place. Such a state also exists on the Megaminx and is depicted in Figure 10.1. Since the superflip of the Rubik's Cube has the highest possible depth of 20, it can be considered difficult to solve and makes for an interesting test subject of solving algorithms. This prompts us to use the Megaminx's superflip as our test subject. Figure 10.2 depicts the results of running our algorithm on the superflip 100 times, where the algorithm achieved an average solution length of 93.86 and an average runtime of 0.196 ms.

Figure 10.1: Superflip on the Megaminx

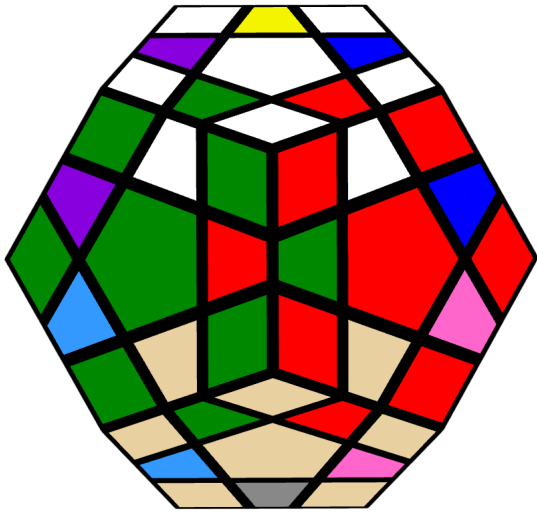
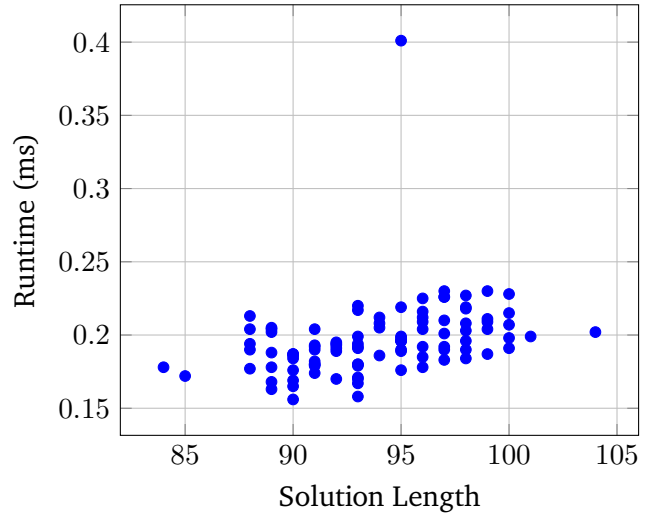


Figure 10.2: 100 solutions to the superflip



We can observe that the runtimes seem to linearly scale with the solution lengths, which is in accordance with Lemma 92. It also becomes apparent that the solution lengths vary a decent bit; hence, it can make sense to generate multiple solutions and then pick the shortest one. For this purpose we introduce a parameter $s \in \mathbb{N}$, which controls how many solutions are generated to pick from. The results of running the solver 100 times on the Superflip for different values of s can be seen in Figure 10.3.

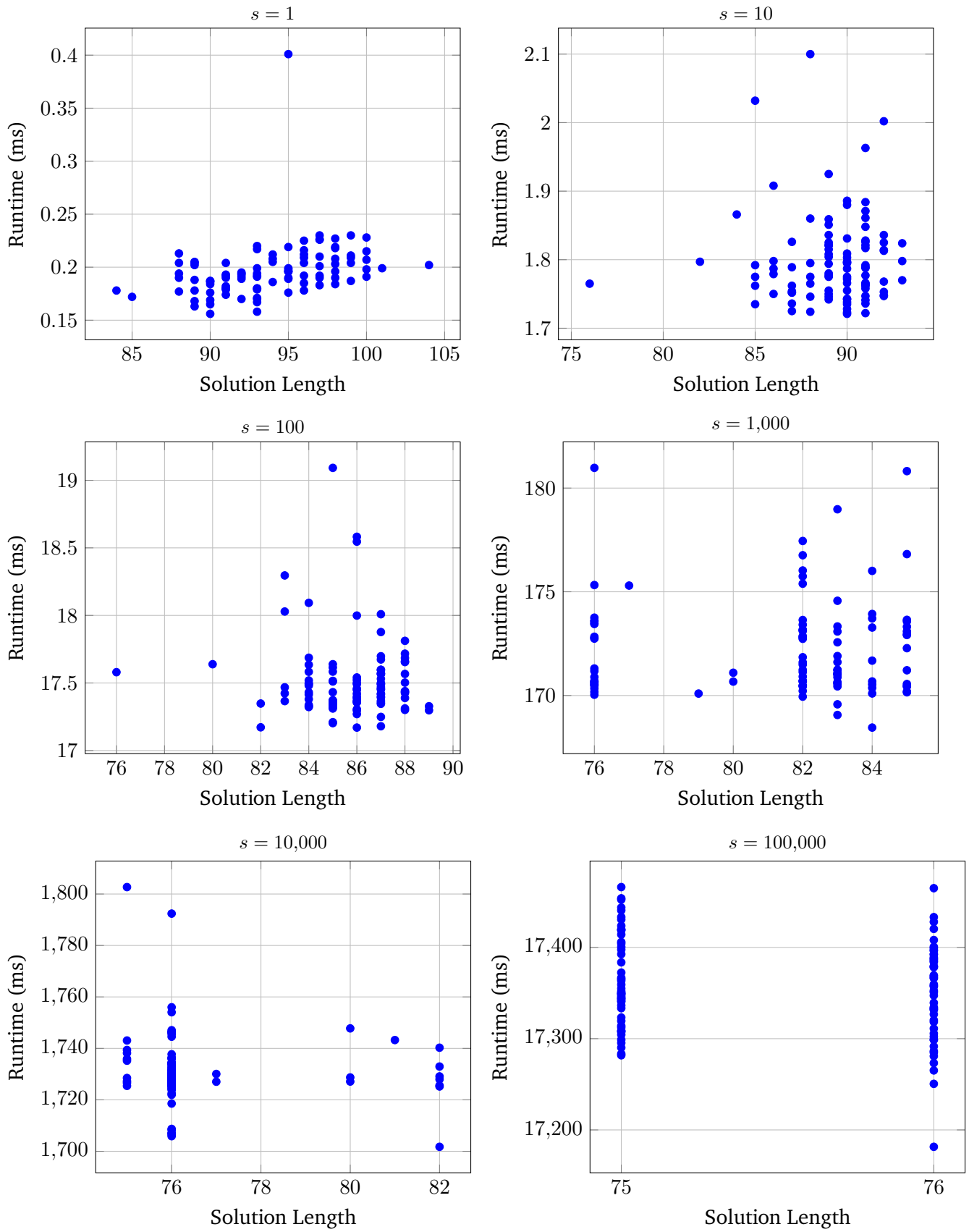


Figure 10.3: Solving superflip for multiple choices of s

First of all, we can observe that the average runtime is proportional to s , which makes sense as the algorithm is executed s times. We also note that for higher values of s , the solution length seems to have little to no effect on the runtime. This observation is also plausible, since generating more solutions statistically improves the shortest solution length but has no effect on the average solution length. Furthermore, we notice that the solution lengths improve quite substantially up until $s = 100,000$, for which they stagnate between 75 and 76. Interestingly, only one solution of 75 moves has been found, so it is very well possible that only one such solution can be generated with our algorithm, in which case the results for $s = 100,000$ could barely be improved any further. While the 75 move solution does not compete with the shortest currently known solution of 58 moves [34], it is still a decent result for a general solving algorithm that is not directed at the Megaminx's Superflip.

Instead of generating entire solutions multiple times, we can also generate multiple solutions for each phase and then choose one that is most promising. In this case, we choose the solution for which the resulting state has the lowest depth in the next phase, as that information is readily available in the respective lookup table. Again, we introduce a parameter $p \in \mathbb{N}$ that controls how many solutions we generate for each phase to choose from. The results of running the solver 100 times on the Superflip, for different choices of p , can be seen in Figure 10.4.

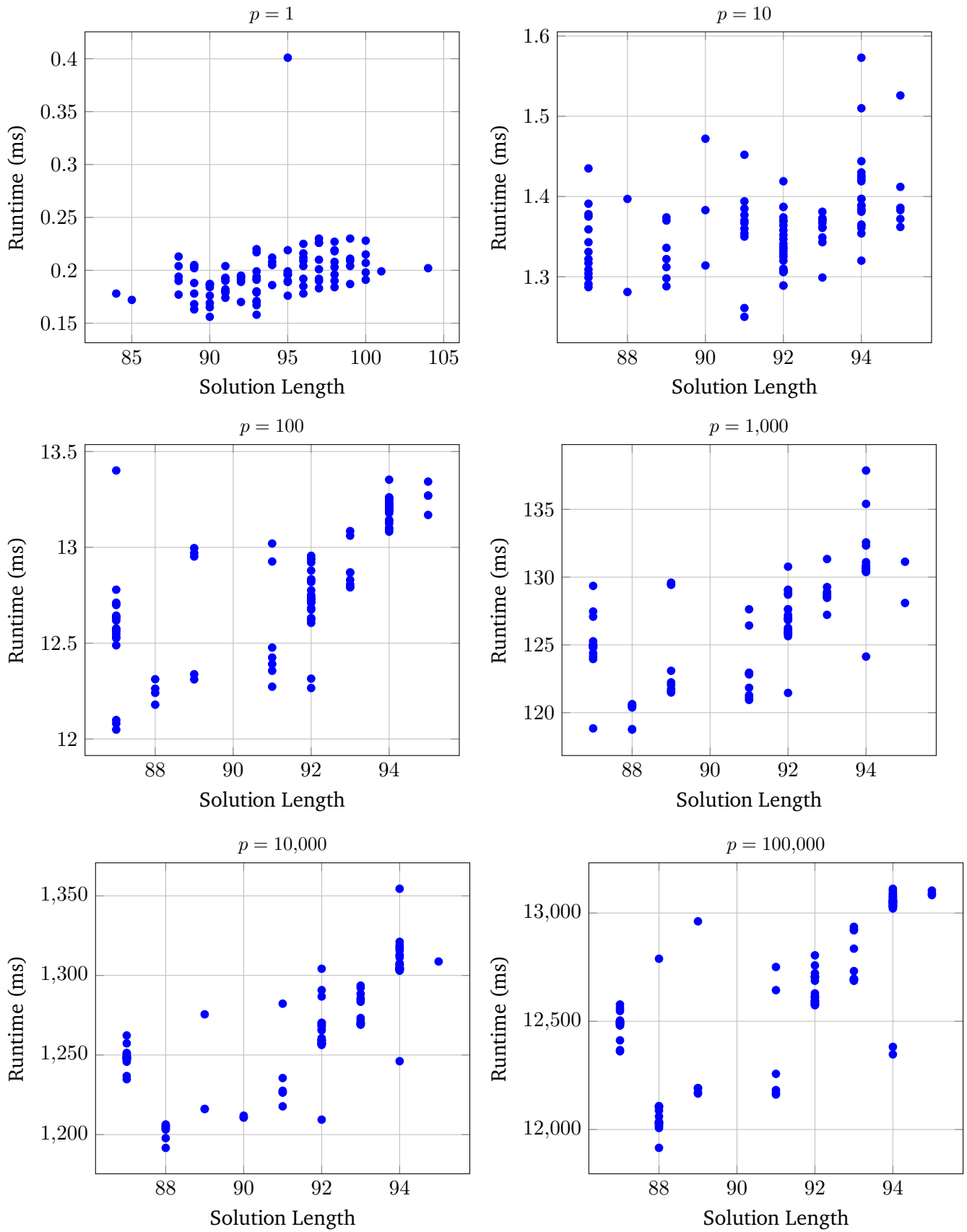


Figure 10.4: Solving superflip for multiple choices of p

This time, the runtime scales a bit better than linearly, most likely because this approach optimizes the average solution lengths. We also note that even for high values of p , there seems to be a correlation between solution length and runtime, for which we can assume the same reason. However, the solution lengths do not really improve beyond $p = 10$, so the better runtime scaling has less of an effect at high values of p .

Now we want to adjust both p and s at the same time. Again, we run the solver 100 times on the Superflip, this time with different choices of s and p , which yields the average solution lengths seen in Table 10.1 and the average runtimes in Table 10.2. Due to the runtime scaling, these tests are limited to $s \cdot p \leq 100,000$.

Table 10.1: Average solution length for superflip

	$s = 1$	$s = 10$	$s = 100$	$s = 1,000$	$s = 10,000$	$s = 100,000$
$p = 1$	93.86	89.20	85.72	81.35	76.51	75.50
$p = 10$	91.53	87.35	86.38	83.89	80.42	
$p = 100$	91.34	87.20	86.85	85.90		
$p = 1,000$	91.20	87.37	86.93			
$p = 10,000$	91.28	87.16				
$p = 100,000$	91.23					

Table 10.2: Average runtime (ms) for superflip

	$s = 1$	$s = 10$	$s = 100$	$s = 1,000$	$s = 10,000$	$s = 100,000$
$p = 1$	0.196	1.797	17.520	172.243	1,731.043	17,356.823
$p = 10$	1.360	13.435	134.927	1,344.257	13,457.261	
$p = 100$	12.821	129.212	1,282.424	12,818.853		
$p = 1,000$	126.724	1,270.527	12,678.229			
$p = 10,000$	1,265.540	12,636.757				
$p = 100,000$	12,639.751					

We can confirm that higher values of p seem to have little to no effect on the resulting solution lengths and even worsen them for higher values of s . So while the runtimes scale better with p , this seems of little use to us, as there is seemingly no real benefit from increasing p .

Lastly, we want to see how the solver performs on a variety of different inputs. So instead of running our solver 100 times on the same state, we generate 100 uniformly independent states of the megaminx at random and run the solver once on each of them, for every choice of s and p . We obtain the average lengths in Table 10.3 and the average runtimes in Table 10.4.

Table 10.3: Average solution length for random states

	$s = 1$	$s = 10$	$s = 100$	$s = 1,000$	$s = 10,000$	$s = 100,000$
$p = 1$	102.40	97.13	92.98	89.80	87.09	85.04
$p = 10$	98.61	93.26	90.35	88.44	86.80	
$p = 100$	97.47	93.46	91.77	90.52		
$p = 1,000$	97.07	93.94	91.69			
$p = 10,000$	97.23	93.46				
$p = 100,000$	97.29					

Table 10.4: Average runtime (ms) for random states

	$s = 1$	$s = 10$	$s = 100$	$s = 1,000$	$s = 10,000$	$s = 100,000$
$p = 1$	0.233	2.206	21.029	205.075	2,028.364	20,154.641
$p = 10$	1.670	16.058	157.274	1,570.218	15,636.108	
$p = 100$	14.904	147.770	1,478.303	14,753.571		
$p = 1,000$	145.834	1,460.920	14,560.178			
$p = 10,000$	1,452.730	14,551.098				
$p = 100,000$	14,562.590					

These results confirm what we already found for the Superflip, so investing more runtime into higher values of s seems to be the most promising.

Since this test setup is generally more reflective of the solver's general performance, we want to use it for all remaining sections of this chapter. For a fair comparison of results, we keep using the same 100 states that we generated here. Also, since we found the parameter p to be of little use for lowering solution lengths, we will stick with $p = 1$ and only vary s from here on out. We refer to this test as the *standard test*.

10.2 Bigger Tables

In Example 91 we have seen that solutions generated with multiple phases are not always optimal, and hence it might be beneficial to reduce the number of phases by omitting subgroups from the subgroup chain. For example, we can omit the subgroup G_4 , leaving us with a lookup table of $[G_3 : G_5] = 20,743,257,600$ GB. A summarized version of this lookup table can be seen in Table 10.5.

Table 10.5: Summary of the new lookup table for the reduced chain

Depth	$G_3 \rightarrow G_5$
0	1
1	8
2	64
3	748
4	7,796
5	73,044
6	637,227
7	5,924,812
8	55,870,729
9	440,301,846
10	2,386,288,360
11	6,923,587,953
12	8,068,077,097
13	2,710,933,209
14	151,186,048
15	368,658
Total cosets	20,743,257,600
Average depth	11.51

We immediately obtain an improved upper bound on the Megaminx Group's diameter.

Theorem 94.

The diameter of the Megaminx Group is at most 133.

We now want to know how this reduced subgroup chain competes with our original subgroup chain. The default test yields the results in Table 10.6.

Table 10.6: Default test for the reduced subgroup chain

	$s = 1$	$s = 10$	$s = 100$	$s = 1,000$	$s = 10,000$	$s = 100,000$
Average Solution Length	100.94	95.29	91.45	88.79	85.95	83.92
Average Runtime (ms)	0.271	2.387	22.056	212.271	2,054.715	20,507.056

As anticipated, the solution lengths have improved a bit. However, even with shorter solutions, the runtimes have increased, perhaps because of the higher memory usage. Interestingly, generating 1,000,000 solutions to

the Superflip yields a best solution length of only 76, so we cannot expect to always find better solutions with a reduced subgroup chain.

10.3 Alternative Subset-Chain

Another idea we can experiment with is to test a different subgroup chain, for which we can make a slight adjustment to our original subgroup chain. We currently have

- $G_5 = \langle R, U, F, L, BR, BL, FR \rangle,$
- $G_6 = \langle R, U, F, L, BR, BL \rangle,$
- $G_7 = \langle R, U, F, L, BR \rangle,$

where all powers of generators are also moves. Instead of first removing moves on FR and then moves on BL , we can also do it the other way around. So let $G'_6 := \langle R, U, F, L, BR, FR \rangle$ and consider the subgroup chain

$$G_0 > G_3 > G_4 > G_5 > G'_6 > G_7 > G_8 > G_9 > G_{10} > G'_{11} > G_{12}, \quad (10.1)$$

which we call the *alternative subgroup chain*. Similarly to what we did in the previous chapter, we can now omit the subgroup G_5 and instead go from G_4 directly to G'_6 . Summaries of all new lookup tables can be seen in Table 10.7.

Table 10.7: Lookup tables of the alternative subgroup chain

Depth	$G_5 \rightarrow G'_6$	$G'_6 \rightarrow G_7$
0	1	1
1	4	4
2	24	24
3	246	282
4	1,878	2,665
5	11,064	23,854
6	34,423	192,654
7	39,925	1,388,680
8	14,628	7,980,302
9	1,027	29,554,259
10	4	54,889,685
11		37,889,454
12		5,909,388
13		63,148
Total cosets	103,224	137,894,400
Average depth	6.55	9.99

Depth	$G_4 \rightarrow G'_6$
0	1
1	8
2	64
3	716
4	7,200
5	66,194
6	572,377
7	5,221,461
8	47,634,189
9	360,683,484
10	1,860,777,907
11	4,960,872,102
12	4,845,900,880
13	1,240,550,667
14	55,378,996
15	164,153
16	1
Total cosets	13,377,830,400
Average depth	11.35

Unfortunately, we do not obtain a better upper bound on the Megaminx Group's diameter. Applying the default test, we obtain the results in Table 10.8 for the alternative subgroup chain and in Table 10.8 for its shortened variant without G_5 .

Table 10.8: Default test for the alternative subgroup chain

	$s = 1$	$s = 10$	$s = 100$	$s = 1,000$	$s = 10,000$	$s = 100,000$
Average Solution Length	102.50	97.85	93.49	90.35	88.00	85.17
Average Runtime (ms)	0.241	2.235	21.348	207.733	2,055.980	20,359.276

Table 10.9: Default test for the shortened alternative subgroup chain

	$s = 1$	$s = 10$	$s = 100$	$s = 1,000$	$s = 10,000$	$s = 100,000$
Average Solution Length	101.12	96.22	91.67	89.08	86.47	84.03
Average Runtime (ms)	0.257	2.370	21.939	212.273	2,063.318	20,681.385

While we can barely notice a difference in the performances of the original and alternate chains, the shortened variant of the original chain performs a bit better than the shortened variant of the alternate chain. On the other hand, for the shortened variants, we only have 14.5 GB worth of lookup tables for the alternate chain, which is considerably less than the 20.3 GB required for the lookup tables of the original chain.

10.4 Meet-In-The-Middle

When omitting subgroups, the resulting lookup tables very quickly become impractical, which sets a limit on how few phases we can realistically have. One way to work around this issue is by utilizing a meet-in-the-middle (MITM) approach, for which we only compute a partial lookup table. This means we only compute the depths of states up to some previously specified $d \in \mathbb{N}_0$ and then solve an arbitrary state m by starting a BFS from m until a state with depth at most d is found. Going from m to this state and then to the solved state yields an optimal solution.

We apply this approach in order to omit the subgroup G'_{11} , which would otherwise require a lookup table of $[G_{10} : G_{12}] \approx 8$ TB. Firstly, we fix $s = 1$ and see how the solver performs with different values of d , which yields the results in Table 10.10.

Table 10.10: MITM approach with various values of d

	$d = 10$	$d = 11$	$d = 12$	$d = 13$	$d = 14$
Average Solution Length	89.63	89.57	90.05	89.40	89.61
Average Runtime (ms)	16,891.412	4,389.687	989.222	272.542	61.252

For values below 10 and above 14, we run out of memory for one of the respective BFS. As we would have expected, the resulting solution lengths remain consistent throughout different choices of d . The runtime seems to scale as 4^{-d} , which is because we only have two faces left, and to find new states, each move has to be followed by one of the four moves on the other face.

Sticking with $d = 14$, we now reintroduce the parameter s , but only for phases up to the last one. That means that we generate s solutions up to the last phase, pick one that is of lowest length, and then finish its last phase optimally with the MITM approach. The results can be seen in Table 10.11.

Table 10.11: MITM approach with $d = 14$ and various values of s

	$s = 1$	$s = 10$	$s = 100$	$s = 1,000$	$s = 10,000$	$s = 100,000$
Average Solution Length	89.23	86.26	84.02	82.25	81.30	80.70
Average Runtime (ms)	54.439	56.859	89.101	237.264	1,826.690	18,065.493

We can see that for increasing values of s , the runtime is also increasingly dominated by the computation of all the solutions, as opposed to the MITM computations. Most importantly, the resulting solution lengths have improved substantially. We can conclude that this approach allows us to obtain considerably better solutions and that the significance of the resulting runtime increase depends on how big s is.

The distribution of depths for this phase was generated in September of 2024 by Ben Whitmore [35] but has not been verified in the scope of this thesis. Since the maximum depth is conjectured to be 26 here, this could be combined with the subgroup chain from Section 10.2 to bring down the upper bound on the Megaminx Group's Diameter to 116.

11 Random State Scrambles

In this chapter, we want to discuss a possible application of our solver for speedcubing competitions. To ensure fairness, puzzles are scrambled according to computer-generated move sequences, called *scrambles*. A rather straightforward approach to generate scrambles is to generate sequences of fixed length $l \in \mathbb{N}$ of uniformly independent random moves. Such scrambling algorithms are called *random move scramblers* and allow for very fast generation of scrambles. In the case of the Rubik's Cube, an argument was made in [8] to show that the resulting distribution of a random move scrambler converges toward the uniform distribution for $l \rightarrow \infty$, and the same argument can be repeated for all twisty puzzles. For practical reasons, we want l to be as small as possible, so we essentially want to find a minimal l for which the resulting distribution can be deemed sufficiently uniform. The issue is that it can be very challenging to determine such an l , and in practice we often have to go with an educated guess.

Until 2007, all scramblers used by the World Cube Association were random move scramblers [36]. While we now know that the random move scrambler with $l = 25$ used for the Rubik's Cube back then was definitely insufficient to obtain a properly scrambled puzzle [8], that scrambler was already replaced in 2008 by a so-called *random state scrambler* developed by H. Kociemba [37]. A random state scrambler generates scrambles by generating solutions to uniformly independent random states of a puzzle. While such scramblers yield a uniform distribution of states, they require the existence of a sufficiently efficient solver for the respective twisty puzzle in question. To this day, the World Cube Association has introduced random state scramblers for most of the competition-relevant puzzles but has imposed lower bounds on the depths of generated states [38].

The Megaminx is one of the puzzles for which the World Cube Association has not yet introduced a random state scrambler. The scrambling algorithm used for the Megaminx is based on an idea from S. Pochmann [39] and is a variant of a random move scrambler. Scrambles generated with this algorithm consist of 77 moves, 70 of which are random, and there are only two choices for each random move. Hence, there are $2^{70} \approx 1.18 \cdot 10^{21}$ such scrambles, so only a tiny fraction of the approximately 10^{68} existing states can actually be generated with these scrambles. Furthermore, it has been suggested in [40] that states obtained with these scrambles are noticeably easier to solve than states generated from a uniform distribution. We can conclude that it would be beneficial to reevaluate the existing Megaminx scrambler and find a better alternative, for example, a random

state scrambler based on the solver that we have developed.

However, the currently used scrambles are a lot easier to apply to a Megaminx than scrambles generated with our solver and also require fewer moves. While we have seen in the last chapter that the solution lengths can be optimized, this improvement came with the cost of higher runtimes and memory usage, which is also not ideal. Hence, further optimization might be required until our solver can be considered a viable alternative for the generation of scrambles.

12 Conclusion

We have seen that twisty puzzles, such as the Rubik's Cube and the Megaminx, can be modeled as permutation groups and that the problem of finding optimal solutions reduces to a known group theoretic problem, for which there are currently no known efficient algorithms. We showed that this problem can be solved with a BFS algorithm but with a runtime and space requirement that is linear in the order of the group. Since this algorithm is only feasible for twisty puzzles for which the number of possible states is relatively low, we utilized a divide-and-conquer technique to refine the algorithm into multiple phases that can be suitably configured, depending on the twisty puzzle at hand and the computational resources available. We then designed and implemented such an algorithm for the Megaminx and determined that it is very efficient in practice. While solving algorithms designed with multiple phases cannot be guaranteed to produce optimal solutions, we have found that our solver can still reliably produce relatively short solutions. We additionally introduced several modification techniques that can further improve the obtained solution lengths of such solving algorithms. However, applying these techniques often leads to increased runtimes. We have seen that algorithms developed with our approach can be used to obtain an upper bound on the God's Number of a twisty puzzle and that a lower bound can be determined with an enumerative argument on the number of move sequences. In the case of the Megaminx, we used these findings to determine that its God's Number must be between 47 and 133. We also explained how such solvers are of importance for the speedcubing community, where they are used to generate scramble sequences that yield a uniform distribution of states.

This thesis serves as a foundation for the group theoretic and computational analysis of twisty puzzles and also pushes forward on such research for the Megaminx. With respect to our research of the Megaminx, we anticipate a verification of the conjectured upper bound of 116 on its God's Number, as well as a substantial expansion on the modification ideas that we developed. In particular, we expect further reductions of the proposed subgroup chains, the development of entirely new subgroup chains, and the application of the meet-in-the-middle approach onto other phases. For twisty puzzles in general, we can expect our findings to be used to support the development and optimization of efficient solving algorithms and to establish and strengthen the bounds of their God's Number. These developments will help the speedcubing community to increase fairness at speedcubing competitions and to further improve and develop solving methods used by humans.



13 Acknowledgments

I want to take this opportunity to express my sincerest gratitude to my advisor Prof. Dr. Pascal Schweitzer for his continuous advice, feedback and motivation. Without him, I would not have been able to work on such an interesting topic and he has greatly assisted my progress with his expertise. I also want to give a special thank you to Jonas Balsfulland and Melda Eksi for all their advice on the programming part of this thesis.

I want to thank everyone who helped me by either proofreading the thesis or discussing its contents with me. Furthermore, I want to thank my friends and family for giving me moral support and to the entire Speedcubing Community for inspiring this topic.

Lastly, I want to thank the TU Darmstadt for letting me utilize the Lichtenberg high performance cluster to perform several relevant computations.

Bibliography

- [1] A. Krotov, *How i built a robot that solves the rubik's cube faster than anyone else in the world*. Accessed: Oct. 24, 2025. [Online]. Available: <https://www.guinnessworldrecords.com/world-records/fastest-robot-to-solve-a-rubiks-cube>
- [2] D. F. Holt, B. Eick, and E. A. O'Brien, *Handbook of Computational Group Theory*. Taylor & Francis Group, 2020, p. 532, ISBN: 9780367659448.
- [3] R. G. Bryan, *God's algorithm for the 2x2x2 pocket cube*. Accessed: Oct. 25, 2025. [Online]. Available: https://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/Jerry_Bryan__God's_Algorithm_for_the_2x2x2_Pocket_Cube.html
- [4] J. Scherphuis, *Mini cube, the 2x2x2 rubik's cube*. Accessed: Oct. 25, 2025. [Online]. Available: <https://www.jaapsch.net/puzzles/cube2.htm>
- [5] J. Scherphuis, *Jaap's puzzle page*. Accessed: Oct. 24, 2025. [Online]. Available: <https://www.jaapsch.net/puzzles/index.htm>
- [6] *Speedsolving puzzles community*. Accessed: Oct. 24, 2025. [Online]. Available: <https://www.speedsolving.com>
- [7] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge, "The diameter of the Rubik's cube group is twenty", *SIAM J. Discrete Math.*, vol. 27, no. 2, pp. 1082–1105, 2013, ISSN: 0895-4801,1095-7146. DOI: 10.1137/120867366
- [8] Y. Qu, T. Rokicki, and H. Yang, "Rubik's cube scrambling requires at least 26 random moves", *arxiv.org*, Oct. 2024. DOI: 10.48550/ARXIV.2410.20630
- [9] T. Rokicki, "Towards God's number for Rubik's cube in the quarter-turn metric", *College Math. J.*, vol. 45, no. 4, pp. 242–253, 2014, ISSN: 0746-8342,1931-1346. DOI: 10.4169/college.math.j.45.4.242
- [10] E. D. Demaine, M. L. Demaine, S. Eisenstat, A. Lubiw, and A. Winslow, "Algorithms for solving Rubik's cubes", in *Algorithms—ESA 2011*, ser. Lecture Notes in Comput. Sci. Vol. 6942, Springer, Heidelberg, 2011, pp. 689–700, ISBN: 978-3-642-23719-5; 978-3-642-23718-8. DOI: 10.1007/978-3-642-23719-5_58
- [11] *50 years of rubik's cube*. Accessed: Oct. 24, 2025. [Online]. Available: <https://www.rubiks.com/history>

-
- [12] D. Singmaster, *The first world championship*. Accessed: Oct. 24, 2025. [Online]. Available: <https://www.jaapsch.net/puzzles/cubic3.htm#p4>
- [13] C. Hardwick, *Rubik's games world championships 2003*. Accessed: Oct. 24, 2025. [Online]. Available: <https://www.speedcubing.com/chris/wc2003.html>
- [14] *What's next for the wca?* Accessed: Oct. 24, 2025. [Online]. Available: <https://www.worldcubeassociation.org/about>
- [15] *History of the speedcubing community & wca*. Accessed: Oct. 24, 2025. [Online]. Available: <https://www.worldcubeassociation.org/speedcubing-history>
- [16] M. Gonzalez, *Rubiks revenge scrambled*, CC BY-SA 3.0. Accessed: Oct. 24, 2025. [Online]. Available: https://commons.wikimedia.org/wiki/File:Rubiks_revenge_scrambled.jpg
- [17] M. Gonzalez, *Pocket cube scrambled*, CC BY-SA 3.0. Accessed: Oct. 24, 2025. [Online]. Available: https://commons.wikimedia.org/wiki/File:Pocket_cube_scrambled.jpg
- [18] L. Garron, *Twizzle editor*. Accessed: Oct. 24, 2025. [Online]. Available: <https://alpha.twizzle.net/edit/?puzzle=megaminx>
- [19] T. Minkwitz, "An algorithm for solving the factorization problem in permutation groups", *Journal of Symbolic Computation*, vol. 26, no. 1, pp. 89–95, 1998, ISSN: 0747-7171. DOI: 10.1006/jsco.1998.0202
- [20] J. D. Dixon and B. Mortimer, *Permutation groups* (Graduate texts in mathematics 163). New York: Springer, 1996, 346 pp., ISBN: 9781461268857.
- [21] Á. Seress, *Permutation group algorithms* (Cambridge tracts in mathematics 152), 2. ed. Cambridge: Cambridge Univ. Press, 2012, 264 pp., ISBN: 9780521661034.
- [22] *Gap – groups, algorithms, and programming, version 4.14.0*. Accessed: Oct. 24, 2025. [Online]. Available: <https://www.gap-system.org>
- [23] A. Botz, *Gap megaminx*. Accessed: Oct. 24, 2025. [Online]. Available: <https://git.rwth-aachen.de/alexander.botz/megaminx-gap>
- [24] C. H. Liang, *Kilominx god's number bounds*. Accessed: Oct. 24, 2025. [Online]. Available: <https://www.speedsolving.com/threads/kilominx-gods-number-bounds.67278/post-1279209>
- [25] H. Kociemba, *Lower bound for megaminx in htm and qtm*. Accessed: Oct. 24, 2025. [Online]. Available: <https://www.speedsolving.com/threads/lower-bound-for-megaminx-in-htm-and-qtm.35724/post-1190421>
- [26] T. H. Cormen, *Introduction to algorithms*, Fourth edition, C. E. Leiserson, R. L. Rivest, and C. Stein, Eds. Cambridge, Massachusetts: The MIT Press, 2022, 1291 pp., ISBN: 9780262367509.
- [27] S. Russell, *Artificial intelligence, A modern approach*, Third edition., P. Norvig, Ed. Boston: Pearson, 2016, 1152 pp., ISBN: 9781292153971.
- [28] D. P. Bertsekas, *Linear network optimization, Algorithms and codes*, 2. print. Cambridge, Massachusetts: MIT Press, 1992, 359 pp., ISBN: 0262023342.

-
- [29] J. Scherphuis, *Thistlethwaite's 52-move algorithm*. Accessed: Oct. 24, 2025. [Online]. Available: <https://www.jaapsch.net/puzzles/thistle.htm>
- [30] D. Ferenc, *The megaminx - easy solution*. Accessed: Oct. 24, 2025. [Online]. Available: <https://ruwix.com/twisty-puzzles/megaminx-gigaminx-teraminx-petaminx/>
- [31] *Solvethecube*. Accessed: Oct. 24, 2025. [Online]. Available: <https://solvethecube.com>
- [32] L. Petrus, *Solving rubik's cube for speed*. Accessed: Oct. 24, 2025. [Online]. Available: <https://lar5.com/cube/>
- [33] A. Botz, *Computational megaminxer*. Accessed: Oct. 24, 2025. [Online]. Available: <https://git.rwth-aachen.de/alexander.botz/megaminx-solver-v2.0>
- [34] W. Randelshofer, *Pretty patterns*. Accessed: Oct. 24, 2025. [Online]. Available: <https://www.randelshofer.ch/rubik/megaminx/U300.02.html>
- [35] B. Whitmore, *Gods algorithm distribution*. Accessed: Oct. 24, 2025. [Online]. Available: https://benwh1.github.io/web/twisty_puzzles/megaminx/megaminx_RU/index.html
- [36] *Competition regulations (2007)*. Accessed: Oct. 24, 2025. [Online]. Available: <https://regulations.worldcubeassociation.org/history/files/regulations2007.html#scrambling>
- [37] *Competition regulations (2008)*. Accessed: Oct. 24, 2025. [Online]. Available: <https://regulations.worldcubeassociation.org/history/files/regulations2008.html#scrambling>
- [38] *Wca regulations*. Accessed: Oct. 24, 2025. [Online]. Available: <https://www.worldcubeassociation.org/regulations/#article-4-scrambling>
- [39] S. Pochmann, *Megaminx scrambler*. Accessed: Oct. 24, 2025. [Online]. Available: https://www.stefan-pochmann.info/spocc/other_stuff/tools/scramble_megaminx
- [40] C. H. Liang, *Proposal to improve megaminx scrambles*. Accessed: Oct. 24, 2025. [Online]. Available: www.speedsolving.com/threads/proposal-to-improve-megaminx-scrambles.95596