JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
**Paul Kainberger**

Submitted at
**Institut für Algebra**

Supervisor
**Assoz. Univ.-Prof. Dipl.-Ing. Dr. Erhard Aichinger**

November 2018

# Using Group Theory for solving Rubik's Cube

Bachelor Thesis
to obtain the academic degree of
Bachelor of Science
in the Bachelor's Program
Technische Mathematik

**JOHANNES KEPLER UNIVERSITY LINZ**
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

# 1 Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

...................... ....................................

Date Paul Kainberger

# 2 Acknowledgement

I want to thank my supervisor Assoz. Univ.-Prof. Dipl.-Ing. Dr. Erhard Aichinger for his assistance and support on my bachelor thesis. Furthermore, I am grateful for my family always supporting me during my studies.

# 3 Abstract

We consider Rubik's Cube mathematically using algebraic group theory. We will see that the 6 possible rotations on the cube generate a group. With the help of the computer algebra software GAP we will be able to calculate a guidance to solve the combination puzzle without using any solving techniques made for humans. Finally, we want to compare the quality of GAP's solutions with those of a beginner's algorithm for humans.

# 4 Kurzfassung

Wir betrachten Rubiks Zauberwürfel mathematisch mittels algebraischer Gruppentheorie. Mithilfe des Computeralgebrasystems GAP möchten wir eine Anleitung entwickeln, um einen beliebig verdrehten Zauberwürfel lösen zu können. Dabei verwenden wir keinerlei Lösungsalgorithmen für Menschen. Schließlich vergleichen wir die Qualität der Lösung des Computeralgebrasystems mit jener eines herkömmlichen Anfänger–Algorithmus.

# Contents

# 5 Introduction

Rubik's Cube is a $3 \times 3 \times 3$ cube with 6 sides. Each side is split up into 9 facets. Every single one of these $6 \cdot 9 = 54$ facets is coloured in one of the six colours $\{\text{blue}, \text{green}, \text{orange}, \text{red}, \text{white}, \text{yellow}\}$ such that there are nine facets per colour. One can rotate each side of the cube and therefore change the colour pattern. The goal of the combination puzzle then is to have all facets of the same colour on one side by twisting all six sides in a correct order (see figure 1).
In general, this cannot be done easily. It took the game's inventor Ernõ Rubik over a month to solve the very first Rubik's Cube in 1974. At that time the Hungarian professor tried to help his students understand three–dimensional problems, hence he constructed this cube, which would take the world by storm. [1]
In total, there are approximately $4.3 \cdot 10^{19}$ different patterns possible giving us no chance of simply studying the solutions for all different cubes. [2, p. 12]
Nowadays, there are several algorithms for people to solve a cube. Learning such an algorithm requires patience and practice. The goal of this thesis is to show that Rubik's Cube can also be solved using mathematical group theory ignoring any of these algorithms.

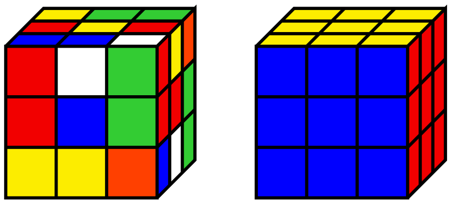


Figure 1: An unsolved Rubik's Cube and its corresponding solved one.

## 5.1 Notation

In order to describe Rubik's Cube, notation is required. Assume that the cube is sitting on a flat surface.

- Let $F'$ denote the front side.
- Let $L'$ denote the left side.
- Let $T'$ denote the top side.
- Let $R'$ denote the right side.
- Let $U'$ denote the underside.
- Let $B'$ denote the back side.

Figure 2 illustrates this definition.

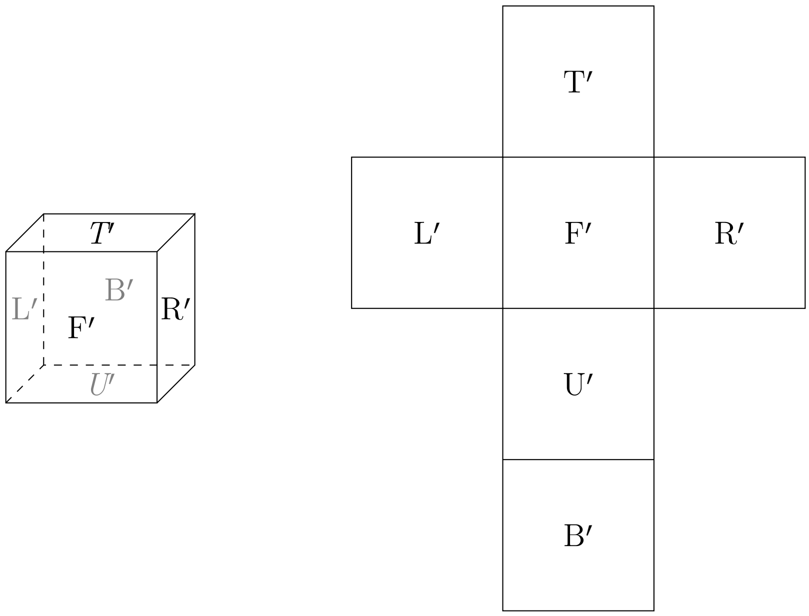
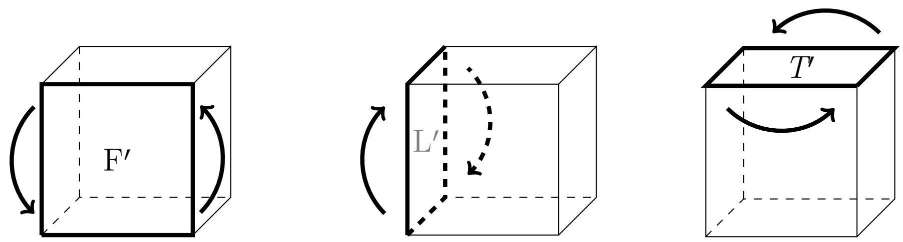
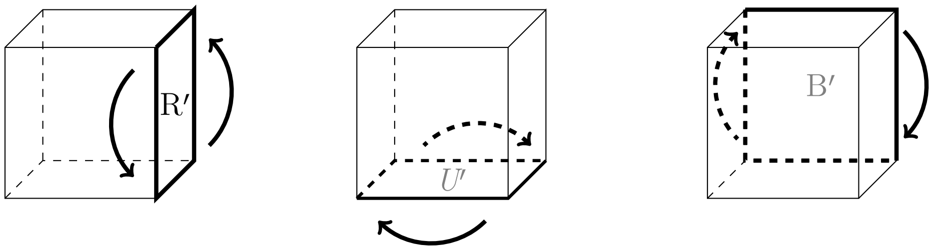



Figure 2: The 6 sides on the (unfolded) cube.

Each of the 6 possible rotations will be considered as one quarter turn (90 degrees) counter–clockwise. The turns are done as if the solver is looking at that particular face, and then turns the face in the counter–clockwise direction. For $M \in \{F, L, T, R, U, B\}$ we defined $M'$ as one side on the cube. Now the rotations on these sides are denoted as $M$. Each twist's inverse is then given by the 90 degree rotation of the face clockwise and denoted $M^{-1}$. [3]
Figure 3 shows how the rotations are considered.




(a) Rotating the front side. (b) Rotating the left side. (c) Rotating the top side.




(d) Rotating the right side. (e) Rotating the underside. (f) Rotating the back side.

Figure 3: Illustration of the six rotations.

# 6 Mathematical approach on Rubik's Cube

The cube consists of 26 pieces: There are 8 corner pieces with 3 different coloured facets, 12 edge pieces, with 2 facets and 2 colours each, and 6 middle pieces, each having just 1 coloured facet. Taking a closer look at Rubik's Cube, we realise that each piece appears on the cube exactly once and is therefore unique. Since all pieces are unique and the colours occur at most once per piece, their facets are unique as well. This motivates us to number each facet. Without loss of generality we will number the unfolded cube as in figure 4.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 19 | 20 | 21 | | | |
| | | | 22 | 23 | 24 | | | |
| | | | 25 | 26 | 27 | | | |
| 10 | 11 | 12 | 1 | 2 | 3 | 28 | 29 | 30 |
| 13 | 14 | 15 | 4 | 5 | 6 | 31 | 32 | 33 |
| 16 | 17 | 18 | 7 | 8 | 9 | 34 | 35 | 36 |
| | | | 37 | 38 | 39 | | | |
| | | | 40 | 41 | 42 | | | |
| | | | 43 | 44 | 45 | | | |
| | | | 46 | 47 | 48 | | | |
| | | | 49 | 50 | 51 | | | |
| | | | 52 | 53 | 54 | | | |

Figure 4: The numbered, unfolded cube.

Now, that each facet is numbered from 1 to 54, we can consider one rotation of a side as a permutation on the 54 facets. We receive 6 permutations representing the twists of the cube's 6 sides. Namely, the permutations are

$$
\begin{aligned}
F &= (1,7,9,3)\ (2,4,8,6)\ (12,37,34,27)\ (15,38,31,26)\ (18,39,28,25),\\
L &= (1,19,46,37)\ (4,22,49,40)\ (7,25,52,43)\ (10,16,18,12)\ (11,13,17,15),\\
T &= (1,28,54,10)\ (2,29,53,11)\ (3,30,52,12)\ (19,25,27,21)\ (20,22,26,24),\\
R &= (3,39,48,21)\ (6,42,51,24)\ (9,45,54,27)\ (28,34,36,30)\ (29,31,35,33),\\
U &= (7,16,48,34)\ (8,17,47,35)\ (9,18,46,36)\ (37,43,45,39)\ (38,40,44,42),\\
B &= (10,21,36,43)\ (13,20,33,44)\ (16,19,30,45)\ (46,52,54,48)\ (47,49,53,51).
\end{aligned}
$$

Furthermore, several twists equal a composition of these 6 permutations which results in another permutation. Hence, the rotations form an algebraic permutation group $G$ with generators $F, L, T, R, U$ and $B$. All properties of a group follow directly from the properties of permutations. Also, the group is not abelian, as permuations do not commute in general. Apparently $G \subseteq S_{54}$, however we have $G \subsetneq S_{54}$, since we cannot permute all facets. For instance, by twisting the cube's sides, the six middle facets

cannot be moved at all and a facet sitting in a corner will always remain in one of the corners.
One can easily find out the permutation which should be applied on the unsolved cube in order to receive the solved one. But in general it is impossible to split this single permutation into a composition of our six permutations $F, L, T, R, U, B$. This is where today's computer algebra systems come in handy, especially GAP [4] with an emphasis on computational group theory.

# 7 A GAP–program for solving Rubik's Cube

In section 11 one can find the GAP–code for the program described here. Its functions are characterised and tested in section 9.

The GAP–program for solving any Rubik's Cube using group theory consists of two main functions, `Solve` and `SolveGuide`, which call several auxiliary functions.
Both main functions take an unsolved Rubik's Cube represented by a list of 54 colours as input. The order of these colours is meant to be as shown in figure 4. Since the user needs to type the colours of 54 facets sitting on a three–dimensional object, the input is first checked for typos.
Next, the pattern of the solved cube is being calculated according to the input of an unsolved cube. This can vary depending on which way the user holds the cube.
As mentioned in section 6, all facets are unique, thus the main functions are able to determine the single permutation between the unsolved cube and its corresponding solved one.
Then, this permutation is split up into the group generators, i.e. the six rotations of the sides, using a GAP–internal function.
`Solve` simply returns a word consisting of the six sides' twists which solve the input cube in that exact order.
On the other hand, the function `SolveGuide` not only gives a solution for an unsolved cube, it also presents a guidance to apply it on a physical cube. We are going to see in section 8 that most solutions for our program are quite large. Also, it is easy to read it wrong. Since the group is not abelian, forgetting or misinterpreting one single move will result in not being able to solve the cube. And we will only realise our mistake at the very end, when the solution is nowhere near. Therefore we want to check every now and then whether we are still on the right path. If we then realise a mistake, we can either restart the program with our current position or even undo the mistake.
Due to that, `SolveGuide` takes two arguments: The unsolved cube and a non–negative integer $n$. The program will then display every $n$–th twist a cube showing what it should look like at that stage in order to make comparison possible.

# 8 Quality of the program

As we now have a program to compute a guidance to solve Rubik's Cube, we are interested in how well it actually works. In particular, we want to know how many rotations the program suggests in order to solve an unsolved cube. The provided solution should consist of few enough twists such that we can actually apply it on a physical cube.
In 2014, Tomas Rokicki and Morley Davidson proved that every possible Rubik's cube can be solved within 26 quarter–moves and that there exist indeed cubes for which 26 quarter–twists are necessary. In this context, the number 26 is referred to as "God's Number". [5] If we consider $M^2$ for $M \in \{F, L, T, R, U, B\}$ as one move instead of two, God's number was found as well and equals 20. [6]
We want to compare God's Number in the quarter–turn metric with the GAP–program. Hence, we let it solve 1 million randomly generated cubes and measure the number of moves the program suggests for each solution. This can be done easily with the function `DistrNumberOfMoves(` $n$ `)`, where in our case $n = 10^6$. The results are shown in table 1.

| # moves | # cubes | # moves | # cubes | # moves | # cubes | # moves | # cubes |
|---|---|---|---|---|---|---|---|
| 31 | 2 | 62 | 1507 | 90 | 26245 | 118 | 8479 |
| 33 | 2 | 63 | 1741 | 91 | 26977 | 119 | 7491 |
| 34 | 1 | 64 | 1975 | 92 | 27952 | 120 | 6699 |
| 37 | 4 | 65 | 2376 | 93 | 28628 | 121 | 5844 |
| 38 | 2 | 66 | 2621 | 94 | 28992 | 122 | 5083 |
| 39 | 4 | 67 | 3163 | 95 | 29974 | 123 | 4330 |
| 40 | 2 | 68 | 3600 | 96 | 29966 | 124 | 3650 |
| 41 | 13 | 69 | 4052 | 97 | 30003 | 125 | 3100 |
| 42 | 9 | 70 | 4465 | 98 | 29872 | 126 | 2538 |
| 43 | 28 | 71 | 5172 | 99 | 29600 | 127 | 2011 |
| 44 | 26 | 72 | 5935 | 100 | 29211 | 128 | 1541 |
| 45 | 30 | 73 | 6681 | 101 | 28762 | 129 | 1137 |
| 46 | 46 | 74 | 7639 | 102 | 27755 | 130 | 841 |
| 47 | 51 | 75 | 8531 | 103 | 26714 | 131 | 670 |
| 48 | 95 | 76 | 9221 | 104 | 25802 | 132 | 485 |
| 49 | 101 | 77 | 10433 | 105 | 24345 | 133 | 307 |
| 50 | 119 | 78 | 11534 | 106 | 23218 | 134 | 222 |
| 51 | 143 | 79 | 12698 | 107 | 21653 | 135 | 163 |
| 52 | 213 | 80 | 13783 | 108 | 19910 | 136 | 95 |
| 53 | 267 | 81 | 15118 | 109 | 18479 | 137 | 74 |
| 54 | 320 | 82 | 16479 | 110 | 17422 | 138 | 34 |
| 55 | 396 | 83 | 17520 | 111 | 15967 | 139 | 14 |
| 56 | 502 | 84 | 19040 | 112 | 14617 | 140 | 14 |
| 57 | 548 | 85 | 20338 | 113 | 13379 | 141 | 12 |
| 58 | 687 | 86 | 21560 | 114 | 12394 | 142 | 7 |
| 59 | 886 | 87 | 22917 | 115 | 11150 | 143 | 1 |
| 60 | 1016 | 88 | 24378 | 116 | 10392 | 144 | 1 |
| 61 | 1242 | 89 | 25114 | 117 | 9431 | 145 | 1 |

Table 1: One million random cubes grouped by the number of moves to solve them.

We can see that most Rubik's Cubes are solved within 60 to 130 twists. This might sound a lot considering every possible cube requires theoratically at most 26 quarter–

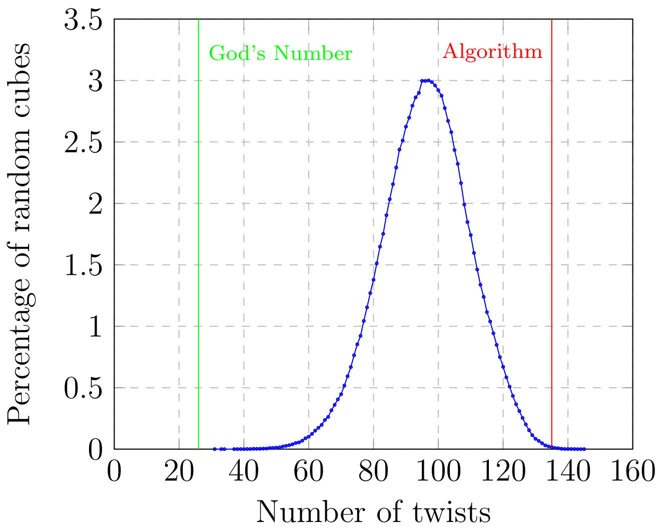



Figure 5: The distribution of 1 million random cubes regarding the number of moves.

rotations. But following algorithms, people need a lot more in general. Daniel Duberg and Jakob Tideström studied the algorithm featured on the official Rubik's Cube website and concluded that the average number of moves for this beginner's algorithm is approximately 135. [7]

Figure 5 visualises the distribution of table 1 together with God's Number and the average number of moves for the beginner's algorithm.

Still, there are other more advanced techniques requiring very few moves. Some so called speedcubing–algorithms can be found in [8].

# 9 Functions

The following GAP–code is defined globally in order to let several funtions as well as the solver use it:

**GAP**

```
gap > F := (1,7,9,3)(2,4,8,6)(12,37,34,27)(15,38,31,26)
(18,39,28,25);;
gap > L := (1,19,46,37)(4,22,49,40)(7,25,52,43)(10,16,18,12)
(11,13,17,15);;
gap > T := (1,28,54,10)(2,29,53,11)(3,30,52,12)(19,25,27,21)
(20,22,26,24);;
gap > R := (3,39,48,21)(6,42,51,24)(9,45,54,27)(28,34,36,30)
(29,31,35,33);;
gap > U := (7,16,48,34)(8,17,47,35)(9,18,46,36)(37,43,45,39)
(38,40,44,42);;
gap > B := (10,21,36,43)(13,20,33,44)(16,19,30,45)(46,52,54,48)
(47,49,53,51);;
gap > cube := Group( F, L, T, R, U, B );;
gap > f := FreeGroup( "F", "L", "T", "R", "U", "B" );;
gap > hom := GroupHomomorphismByImages( f, cube,
GeneratorsOfGroup( f ), GeneratorsOfGroup( cube ) );;
gap > b := "blue";;
gap > g := "green";;
gap > o := "orange";;
gap > r := "red";;
gap > w := "white";;
gap > y := "yellow";;
```

First, there are the 6 rotations $F, L, T, R, U, B$ defined, which are used as group generators for the group cube.
Next we define a free group $f$ with $"F", "L", "T", "R", "U", "B"$ as generators. Note that these generators are strings and not the above variables.
Then we form a homomorphism between these groups. It maps the generators from the free group to the related permutation.
Lastly, we define the six colours as variables. Since the input cube is a list of 54 colours, the user surely appreciates that he can type the variables rather than the whole strings.

## 9.1 IsInputCorrect

This function checks whether the input cube, represented by a list of 54 colors, has obvious typos. Possible inputerrors would be for instance forgetting or double-counting one facet. However the function does not check if the list of colours does indeed represent an existing cube. There might be for instance the unlikely case of accidentally swapping two colours, which are either both middle pieces or both not middle pieces, such that solving the cube becomes impossible. Then IsInputCorrect would return `true`, even though the cube is not solvable. This problem is fixed in the main functions `Solve` and `SolveGuide`, which will return an error if an input cube does not actually exist.

### Test 1

For our first test we use the solved cube itself and rotate the front side once. The code

**GAP**

```
gap > c := [w,w,w,w,w,w,w,w,w,o,o,b,o,o,b,o,o,b,b,b,b,b,b,r,
r,r,g,r,r,g,r,r,g,r,r,o,o,o,g,g,g,g,g,g,y,y,y,y,y,y,y,y,y];;
gap > IsInputCorrect[ c ];
true
```

returns `true` as desired.

### Test 2

Now we want to see how input errors are handled. We exchange two middle pieces, change the first colour from "white" to "red" and omit the second colour.

**GAP**

```
gap > c := [r,w,w,o,w,w,w,w,o,o,b,o,w,b,o,o,b,b,b,b,b,b,r,r,
r,g,r,r,g,r,r,g,r,r,o,o,o,g,g,g,g,g,g,y,y,y,y,y,y,y,y,y];;
gap > IsInputCorrect[ c ];
Typo.  Cube must have 54 colours instead of 53.
Typo.  Middle facets are not duplicate free.
Typo.  10 facets coloured in red instead of 9.
Typo.  7 facets coloured in white instead of 9.
false
```

### Test 3

For the third test we want to input a non–existing cube which is not treated as wrong input. If we consider a solved cube and just swap one edge, the cube is not solvable any more. However, we will consider the same example later on and see that the functions `Solve` and `SolveGuide` will take care of this kind of input.

**GAP**

```
gap > (2,26) in cube;
false
gap > doesNotExist := [w,o,w,w,w,w,w,w,w,g,g,g,g,g,g,g,g,g,
o,o,o,o,o,o,o,w,o,b,b,b,b,b,b,b,b,b,
r,r,r,r,r,r,r,r,r,y,y,y,y,y,y,y,y,y];;
gap > IsInputCorrect( doesNotExist );
true
```

## 9.2 DisplayCube

A list of 54 colours is quite difficult to read. Therefore, the function `DisplayCube(l)` displays a list `l` as an unfolded cube. First of all, we can compare the input with the actual cube which helps us prevent typos and correct them. Furthermore, we can check while solving the cube whether we made a mistake. We can then try to undo the mistake or restart the program using the current position rather than realising at the very end that we made a mistake several twists ago. We want to test `DisplayCube` together with the function `SolvedCube`.

## 9.3 SolvedCube

`SolvedCube` calculates the order of colours of the solved cube depending on the input cube.
We could also fix the solved cube. Then the user would have to rotate the cube as a whole in order to match the middle facets with the fixed solved cube which would decrease user friendliness. As writing the input for our program already takes a long time compared to solving it afterwards, we should be as quick as possible to get the program started.

### Test 1

We test two unsolved cubes with two different solved cubes. In order to be able to read them properly we will make use of the function `DisplayCube`.

**GAP**

```
gap > unsolved := [w,o,r,w,w,b,o,g,r,g,g,o,w,g,o,b,r,y,r,b,w,o,o,
b,b,y,b,y,r,o,y,b,y,y,o,y,g,y,g,w,r,b,r,r,o,w,g,b,g,y,r,w,w,g];;
gap > DisplayCube( unsolved );

          r  b  w
          o  o  b
          b  y  b
 g  g  o  w  o  r  y  r  o
 w  g  o  w  w  b  y  b  y
 b  r  y  o  g  r  y  o  y
          g  y  g
          w  r  b
          r  r  o
          w  g  b
          g  y  r
          w  w  g

gap > DisplayCube( SolvedCube( unsolved ) );

          o  o  o
          o  o  o
          o  o  o
 g  g  g  w  w  w  b  b  b
 g  g  g  w  w  w  b  b  b
 g  g  g  w  w  w  b  b  b
          r  r  r
          r  r  r
          r  r  r
          y  y  y
          y  y  y
          y  y  y
```

We can see clearly that the middle facets match.

**Test 2**

**GAP**

```
gap > unsolved := [w,o,r,w,y,b,o,g,r,g,g,o,w,b,o,b,r,y,r,b,w,o,o,
b,b,y,b,y,r,o,y,g,y,y,o,y,g,y,g,w,w,b,r,r,o,w,g,b,g,o,r,w,w,g];;
gap > DisplayCube( unsolved );

          r  b  w
          o  o  b
          b  y  b
 g  g  o  w  o  r  y  r  o
 w  g  b  w  w  r  y  g  y
 b  r  y  o  g  r  y  o  y
          g  y  g
          w  w  b
          r  r  o
          w  g  b
          g  o  r
          w  w  g

gap > DisplayCube( SolvedCube( unsolved ) );

          o  o  o
          o  o  o
          o  o  o
 g  g  g  w  w  w  b  b  b
 g  g  g  w  w  w  b  b  b
 g  g  g  w  w  w  b  b  b
          r  r  r
          r  r  r
          r  r  r
          y  y  y
          y  y  y
          y  y  y
```

Again, the middle facets match perfectly.

## 9.4 FindPosCornerCInCornersC

As mentioned in section 6, each corner is unique.
The function `FindPosCornerCInCornersC` takes as argument one corner piece represented by its three different colours and determines the position of that piece on the solved cube together with the permutation it needs in order to fit on the solved cube. In other words, it finds out where that specific corner piece should lie on the solved cube. This is needed for the function `ColoursToNumbers`.

**Test 1**

Let $A = [[\mathtt{w},\mathtt{b},\mathtt{o}],[\mathtt{r},\mathtt{b},\mathtt{w}],[\mathtt{r},\mathtt{w},\mathtt{g}]]$ be a list of corners. We want to find the position and permutation $p$ of the corner $a = [\mathtt{w},\mathtt{g},\mathtt{r}]$ in $A$, such that $a \circ p \in A$.

**GAP**

```
gap > A := [[w,b,o],[r,b,g],[r,w,g]];;
gap > a := [w,g,r];;
gap > FindPosCornerCInCornersC( a, A );
[ 3, (1,2,3) ]
```

This means that $a \circ (1,2,3)$ is located at the third position in $A$, which is true.

**Test 2**

Let $A = [[\mathtt{w},\mathtt{b},\mathtt{o}],[\mathtt{r},\mathtt{b},\mathtt{w}],[\mathtt{r},\mathtt{w},\mathtt{g}]]$ be the same list of corners as in Test 1. Let $a = [\mathtt{w},\mathtt{r},\mathtt{o}]$, which is not in the list $A$.

**GAP**

```
gap > A := [[w,b,o],[r,b,g],[r,w,g]];;
gap > a := [w,r,o];;
gap > FindPosCornerCInCornersC( a, A );
Error, Corner [white,red,orange] not found on the cube.
called from <function "FindPosCornerCInCornersC">( <arguments> )
```

Since the corner cannot be found, an error is returned.

## 9.5 FindPosEdgeCInEdgesC

This function is the equivalence to the function `FindPosCornerCInCornersC`, but this time for edges on Rubik's cube instead of corners, meaning that the lists consist of two colours rather than three.

## 9.6 ColoursToNumbers

As mentioned in section 6, all facets on Rubik's Cube are unique and we can therefore number them. It makes sense to first number the solved cube. Then, we can deduce the numbering on the unsolved cube from it. This is exactly what the function `ColoursToNumbers` does. It takes the list of 54 shuffled colours, derives the solved cube from it, numbers the solved cube and then calculates the unique numbering of the unsolved cube. This function is crucial in order to find out the desired permutation in our Rubik's Cube group.

## 9.7 Solve

Finally, we are able to solve a Rubik's cube. The input of the function `Solve` is a list of 54 colours representing an arbitrary cube. The output is desired to be a word consisting of the six group generators $F, L, T, R, U, B$. Applying these permutations (i.e. twisting the cube's sides) in this very order should then solve the input cube.

### Test 1

In order to test the function we generate a random group element on the cube. This is achieved by the GAP–internal function `Random`. Since all group elements are just permutations, we first apply it on a fixed solved cube.
Note that the function `Solve` returns a "word", which is why we cannot use it as permutation on the unsolved cube. Therefore we use the GAP–internal function `Image` which finds the image of the word on our globally defined mapping `hom`.

**GAP**

```
gap > solved := [w,w,w,w,w,w,w,w,w,g,g,g,g,g,g,g,g,g,o,o,o,o,o,
o,o,o,o,b,b,b,b,b,b,b,b,b,r,r,r,r,r,r,r,r,r,y,y,y,y,y,y,y,y,y];;
gap > unsolved := Permuted( solved, Random( cube ) );;
gap > DisplayCube( unsolved );

          r  o  w
          y  o  r
          r  b  o
 b  r  b  w  w  g  w  g  g
 g  g  g  o  w  r  w  b  g
 r  b  y  b  r  o  b  o  y
          o  b  w
          y  r  y
          y  o  g
          g  w  o
          y  y  w
          y  b  r

gap> sol := Solve( unsolved );
L*B^-1*R^-1*F*T*R^-1*F*U^-1*R^2*U*B^-1*U^-1*B*R^-1*T*R*L^-1*F*
T^-1*L*F^-1*(T*L)^2*T^-1*L^-1*T^-1*U*L^-1*(U^-1*F)^2*(U*L)^2*
U^-1*F^-1*L^-1*T*F*T^-1*L*T^-1*L^-1*T*L^-1*F^-1*L*F*T*F*R*F^-1*
R^-1*T^-1*F^-1*T*R*F^-1*R^-1*T^-1*L^-1*F*L
```

```
gap > DisplayCube( Permuted( unsolved, Image( hom, sol ) ) );

          o  o  o
          o  o  o
          o  o  o
 g  g  g  w  w  w  b  b  b
 g  g  g  w  w  w  b  b  b
 g  g  g  w  w  w  b  b  b
          r  r  r
          r  r  r
          r  r  r
          y  y  y
          y  y  y
          y  y  y
```

The combination of group generators solves the cube.

#### Test 2

We consider again the non–existing Rubik's cube from Test 3 of function `IsInputCorrect`. The function `IsInputCorrect` only checks for obvious mistakes but is not able to determine whether the cube actually exists. This problem is now handled in the functions `Solve` and `SolveGuide`.

GAP

```
gap > doesNotExist := [w,o,w,w,w,w,w,w,w,g,g,g,g,g,g,g,g,g,
o,o,o,o,o,o,o,w,o,b,b,b,b,b,b,b,b,b,
r,r,r,r,r,r,r,r,r,y,y,y,y,y,y,y,y,y];;
gap> Solve( doesNotExist );
Error, Cube does not exist.  Check input.
called from <function "Solve">( <arguments> )
```

## 9.8 SolveGuide

As mentioned in section 7, `SolveGuide` does not only take the list of 54 colours as input, but also a non–negative integer $n$ as a second argument and displays a guidance for solving the cube. It splits the whole solution into smaller steps of length $n$ and displays the Rubik's Cube in between these steps to allow comparing it with the physical cube.

#### Test 1

Let us first test a solved Rubik's Cube, which was only twisted a little bit in order to receive a short solution. We take $n = 3$, meaning that the cube is being displayed every third move.

**GAP**

```
gap > unsolved := [b,o,g,b,w,g,o,o,w,o,g,y,y,g,w,y,g,g,
b,o,r,y,o,r,r,b,y,o,w,w,r,b,b,b,y,o,
w,w,r,w,r,o,g,y,w,r,b,b,r,y,r,y,g,g];;
gap> SolveGuide( unsolved, 3 );
Your initial cube:


          b  o  r
          y  o  r
          r  b  y
 o  g  y  b  o  g  o  w  w
 y  g  w  b  w  g  r  b  b
 y  g  g  o  o  w  b  y  o
          w  w  r
          w  r  o
          g  y  w
          r  b  b
          r  y  r
          y  g  g

Step 1:
F^-1*L*R^-1

          o  o  b
          o  o  o
          w  w  g
 w  w  r  b  b  o  y  b  r
 g  g  g  w  w  o  y  b  w
 o  y  y  g  g  w  o  b  w
          r  r  b
          r  r  r
          y  y  g
          b  b  r
          y  y  r
          g  g  y
```

```
Step 2:
R^-1*T*U

         o  o  w
         o  o  w
         o  o  w
g  g  g  w  w  r  b  b  b
g  g  g  w  w  r  b  b  b
g  g  g  w  w  r  b  b  b
         r  r  y
         r  r  y
         r  r  y
         y  y  o
         y  y  o
         y  y  o

Step 3:
R
Your cube is now solved.
```

### Test 2

Now we consider a random cube. Let $n = 25$ to keep the output small.

**GAP**

```
gap > solved := [w,w,w,w,w,w,w,w,w,g,g,g,g,g,g,g,g,g,o,o,o,o,o,
o,o,o,o,b,b,b,b,b,b,b,b,b,r,r,r,r,r,r,r,r,r,y,y,y,y,y,y,y,y,y];;
gap > unsolved := Permuted( solved, Random( cube ) );;
gap > SolveGuide( unsolved, 25 );
Your initial cube:

         r  b  o
         o  o  w
         y  b  w
y  w  g  o  o  r  b  g  b
g  g  y  b  w  r  w  b  y
r  g  o  w  b  r  g  g  w
         g  r  w
         r  r  o
         b  y  b
         y  r  o
         y  y  o
         g  w  y
```

```
Step 1:
L^-1*F^-1*L*F*L*T^-1*L^-1*T^-2*F*T^-1*F^-1*
T^-1*L*T*L^-1*F^-1*L*F*T*F^-1*T^-1*L^-1*F*T^-1

          b  o  r
          b  o  b
          y  w  o
 r  y  g  o  g  b  y  w  g
 g  g  b  o  w  r  b  b  y
 r  g  g  w  r  w  g  g  w
          r  w  o
          r  r  o
          b  y  b
          y  r  o
          y  y  o
          w  w  y

Step 2:
L*T*L^-1*F^-1*L^-1*F*L*F^-2*L^-1*F*L*T^-1*L*
T*L^-1*F^-2*T*(F^-1*T^-1*L^-1)^2

          g  y  y
          r  o  b
          r  w  o
 o  g  g  y  g  b  y  w  b
 r  g  o  b  w  o  w  b  y
 g  y  w  r  r  b  r  g  w
          g  w  w
          b  r  o
          y  y  b
          o  r  o
          b  y  o
          w  g  r

Step 3:
L^-1*T*L^2*F^-1*L^-1*F*L^-2*F^-2*T*L*T*L^-1*
T^-1*F^-1*(F^-1*T^-1)^2*F*B^-1*U*B
```

```
          y b b
          g o o
          g o o
g y o y b b y w w
r g b w w w g b y
b w w g g r g b w
          o o w
          r r y
          y y b
          r r o
          g y o
          r r r


Step 4:
F^-2*T^-1*U^-1*R^-1*U*F*R^-2*F^-1*B^-1*R^-1*U^-1
Your cube is now solved.
```

We can verify the result by applying the twists' inverses on a solved cube. Note that since the permutation group is not abelian, for two moves $A, B \in \{F, L, T, R, U, B\}$ we have $(AB)^{-1} = B^{-1}A^{-1}$ [9, p. 83].

## 9.9 DistrNumberOfMoves

This function is used for testing purposes (see section 8).
For $n \in \mathbb{N}$, `DistrNumberOfMoves(` $n$ `)` calculates how many moves the program suggests to solve $n$ randomly chosen Rubik's cubes. It then groups all cubes with the same amount of twists in order to get a distribution. The function returns a list consisting of lists with 2 elements. The first entry of each inner list contains the number of moves while the second one features the number of cubes requiring this number of moves to be solved by the GAP–program.
`DistrNumberOfMoves(` $n$ `)` terminates for $n \leq 10^6$ within reasonable time (a few seconds).

### Test

We test the function for $n = 20$.

**GAP**

```
gap > DistrNumberOfMoves( 20 );
[ [ 85, 1 ], [ 87, 2 ], [ 89, 1 ], [ 91, 1 ], [ 92, 1 ],
[ 94, 1 ], [ 97, 2 ], [ 98, 1 ], [ 101, 1 ], [ 103, 1 ],
[ 107, 1 ], [ 108, 2 ], [ 109, 3 ], [ 113, 1 ], [ 114, 1 ] ]
```

This means that for 20 randomly generated Rubik's Cubes, there is one having 85 moves to be solved, there are two cubes which need 87 twists and so on. Once more, note that all these cubes are still theoratically solvable within 26 twists (see section 8).

# 10 Outlook

Ernõ Rubik's original cube is $3 \times 3 \times 3$, but other versions are produced in different shapes and sizes. Since those cubes' operations also generate permutation groups, they can be dealt with GAP as well.
Christoph Bandelow and also David Joyner not only considered a $2 \times 2 \times 2$–cube, but likewise more abstract ones including a pyramid or a dodecahedron. Again, all sides are rotatable and permute the facets. The number of sides, which equals the number of different colours, impacts the size of the generated permutation group. For instance, the Magic Dodecahedron with twelve faces allows approximately $10^{68}$ different patterns. [10, 11]
Although the solutions for larger groups will not be as short as those for the original Rubik's Cube ($\sim 10^{19}$ possibilities), one can still find a guidance for every pattern using group theory without any solving algorithm for humans.

# 11 Appendix

## 11.1 GAP–Code

```
#global definitions

F := (1,7,9,3)(2,4,8,6)(12,37,34,27)(15,38,31,26)(18,39,28,25);
L := (1,19,46,37)(4,22,49,40)(7,25,52,43)(10,16,18,12)(11,13,17,15);
T := (1,28,54,10)(2,29,53,11)(3,30,52,12)(19,25,27,21)(20,22,26,24);
R := (3,39,48,21)(6,42,51,24)(9,45,54,27)(28,34,36,30)(29,31,35,33);
U := (7,16,48,34)(8,17,47,35)(9,18,46,36)(37,43,45,39)(38,40,44,42);
B := (10,21,36,43)(13,20,33,44)(16,19,30,45)(46,52,54,48)(47,49,53,51);

cube := Group( F, L, T, R, U, B );
f := FreeGroup( "F", "L", "T", "R", "U", "B" );
hom := GroupHomomorphismByImages( f, cube, GeneratorsOfGroup( f ), GeneratorsOfGroup( cube ) );

b := "blue";
g := "green";
o := "orange";
r := "red";
w := "white";
y := "yellow";

########################################

IsInputCorrect := function( c )
	#######
	#Input:			c, a cube represented by a list of colours.
	#Precondition:		IsList( c )
	#Output:		true if there are no obvious typos in c, otherwise false.
	#Postcondition:		c does not have obvious typos.
	#######

	local flag, col, i;

	flag := true;
	if not( Size( c ) = 54 ) then
		Print( "Typo. Cube must have 54 colours instead of ", Size( c ), ".\n");
		flag := false;
	fi;
	if not( IsDenseList( c ) ) then
		Print( "Typo. List of colours is not dense.\n");
		flag := false;
```

```
        fi;
        if not( IsDuplicateFree( [c[5],c[14],c[23],c[32],c[41],c[50]] ) ) then
            Print( "Typo. Middlepieces are not duplicate free.\n" );
            flag := false;
        fi;
        col := Collected( c );
        if not( Size( col ) = 6 ) then
            Print( "Typo. Cube must have 6 different colours instead of ", Size( col ), ".\n");
            flag := false;
        fi;
        i := 1;
        while i <= Size( col ) do
            if col[i][1] in [b,g,o,r,w,y] then
                if not( col[i][2] = 9 ) then
                    Print( "Typo. ", col[i][2], " pieces coloured in ", col[i][1], " instead of 9.\n");
                    flag := false;
                fi;
            else
                Print( "Typo. Colour ", col[i][1], " does not exist.\n");
                flag := false;
            fi;
            i := i + 1;
        od;
        return flag;
    end;

    ######################################

    SolvedCube := function( u )
        #######
        #Input:          u, an unsolved cube represented by a list of 54 colours
        #Precondition:   IsInputCorrect( u )
        #Output:         solved cube, represented by a list of 54 colours
        #Postcondition:  middlepiecesC( u ) = middlepiecesC( solved ) and
        #                solved represents a solved cube
        #######

        local middlePiecesC, solved, i, j;

        middlePiecesC := [u[5],u[14],u[23],u[32],u[41],u[50]];
        solved := [];
        i := 1;
        while i <= Size( middlePiecesC ) do
            j := 1;
```

```
            while j <= 9 do
                Add( solved, middlePiecesC[i] );
                j := j + 1;
            od;
            i := i + 1;
        od;
        return solved;
end;

#####################################

DisplayCube := function( cube )
        #######
        #Input:             cube, represented by a list of 54 colours.
        #Precondition:      IsInputCorrect( cube )
        #Output:            none, function prints the unfolded cube.
        #Postcondition:     none
        #######

        local c, i;

        #only display first character of strings
        c := ShallowCopy( cube );
        i := 1;
        while i <= 54 do
            c[i] := c[i]{[1]};
            i := i + 1;
        od;

        Print( "      ", c[19], " ", c[20], " ", c[21], "\n" );
        Print( "      ", c[22], " ", c[23], " ", c[24], "\n" );
        Print( "      ", c[25], " ", c[26], " ", c[27], "\n" );
        Print( c[10], " ", c[11], " ", c[12], " ", c[1], " ", c[2], " ", c[3], " ", c[28], " ", c[29], " ", c[30], "\n" );
        Print( c[13], " ", c[14], " ", c[15], " ", c[4], " ", c[5], " ", c[6], " ", c[31], " ", c[32], " ", c[33], "\n" );
        Print( c[16], " ", c[17], " ", c[18], " ", c[7], " ", c[8], " ", c[9], " ", c[34], " ", c[35], " ", c[36], "\n" );
        Print( "      ", c[37], " ", c[38], " ", c[39], "\n" );
        Print( "      ", c[40], " ", c[41], " ", c[42], "\n" );
        Print( "      ", c[43], " ", c[44], " ", c[45], "\n" );
        Print( "      ", c[46], " ", c[47], " ", c[48], "\n" );
        Print( "      ", c[49], " ", c[50], " ", c[51], "\n" );
        Print( "      ", c[52], " ", c[53], " ", c[54] );
end;
```

```
########################################

FindPosCornerCInCornersC := function( cC, csC )
	#######
	#Input:			cC, a list of 3 colours and
	#			csC, a list of lists of 3 colors
	#Precondition:		IsList( cC ) and
	#			IsList( csC )
	#Output:		i, an index which marks the position of cC in csC and
	#			the permutation p s.t. cC*p in csC
	#Postcondition:		cC = csC[i] and
	#			IsPerm( p ) and
	#			cC*p in csC
	#######

	local s3, i;

	s3 := [(),(1,2),(1,3),(2,3),(1,2,3),(1,3,2)];
	i := 1;
	while i <= Size( s3 ) do
		if Permuted( cC, s3[i] ) in csC then
			return [Position( csC, Permuted( cC, s3[i] ) ),s3[i]];
		fi;
		i := i + 1;
	od;
	ErrorNoReturn( "Corner [", cC[1], ",", cC[2], ",", cC[3], "] not found on the cube." );
end;

########################################

FindPosEdgeCInEdgesC := function( eC, esC )
	#######
	#Input:			eC, a list of 2 colours and
	#			esC, a list of lists of 2 colors
	#Precondition:		IsList( eC ) and
	#			IsList( esC )
	#Output:		i, an index which marks the position of eC in esC and
	#			the permutation p s.t. eC*p in esC
	#Postcondition:		eC = esC[i] and
	#			IsPerm( p ) and
	#			eC*p in esC
	#######

	local s2, i;
```

```

    s2 := [(),(1,2)];
    i := 1;
    while i <= Size( s2 ) do
        if Permuted( eC, s2[i] ) in esC then
            return [Position( esC, Permuted( eC, s2[i] ) ),s2[i]];
        fi;
        i := i + 1;
    od;
    ErrorNoReturn( "Edge [", eC[1], ",", eC[2], "] not found on the cube." );
end;

####################################

ColoursToNumbers := function( unsolved, solved )
    #######
    #Input:          unsolved, a cube represented by a list of 54 colours and
    #                solved, a cube represented by a list of 54 colours
    #Precondition:   IsList( unsolved ) and
    #                IsList( solved ) and
    #                IsInputCorrect( unsolved ) and
    #                solved = SolvedCube( unsolved )
    #Output:         l, a list containing the numbers 1..54
    #Postcondition:  IsList( l ) and
    #                Permuted( l, MappingPermListList( solvedNr, unsolvedNr ) ) = [1..54]
    #######

    local l, cornersC, cornersNr, edgesC, edgesNr, i, j, cornerC, edgeC, posPerm, n, c, e;

    l := [,,,,5,,,,,,,,,14,,,,,,,,,23,,,,,,,,,32,,,,,,,,,41,,,,,,,,,50,,,,];

    #Calculate corners of solved cube as colour triples
    cornersNr := [[1,12,25],[3,27,28],[7,18,37],[9,34,39],[10,19,52],[16,43,46],[21,30,54],[36,45,48]];
    cornersC := [];
    i := 1;
    while i <= Size( cornersNr ) do
        j := 1;
        c := [];
        while j <= Size( cornersNr[1] ) do
            Add( c, solved[cornersNr[i][j]] );
            j := j + 1;
        od;
        Add( cornersC, c );
```

```
			i := i + 1;
		od;

		#Calculate edges of solved cube as colour doubles
		edgesNr := [[2,26],[4,15],[6,31],[8,38],[11,22],[13,49],[17,40],[20,53],[24,29],[33,51],[35,42],[44,47]];
		edgesC := [];
		i := 1;
		while i <= Size( edgesNr ) do
			j := 1;
			e := [];
			while j <= Size( edgesNr[1] ) do
				Add( e, solved[edgesNr[i][j]] );
				j := j + 1;
			od;
			Add( edgesC, e );
			i := i + 1;
		od;

		#Locate corners of unsolved cube in solved cube
		i := 1;
		while i <= Size( cornersNr ) do
			cornerC := [unsolved[cornersNr[i][1]],unsolved[cornersNr[i][2]],unsolved[cornersNr[i][3]]];
			posPerm := FindPosCornerCInCornersC( cornerC, cornersC );
			n := Permuted( cornersNr[posPerm[1]], posPerm[2]^-1 );
			j := 1;
			while j <= 3 do
				l[cornersNr[i][j]] := n[j];
				j := j + 1;
			od;
			i := i + 1;
		od;

		#Locate edges of unsolved cube in solved cube
		i := 1;
		while i <= Size( edgesNr ) do
			edgeC := [unsolved[edgesNr[i][1]],unsolved[edgesNr[i][2]]];
			posPerm := FindPosEdgeCInEdgesC( edgeC, edgesC );
			n := Permuted( edgesNr[posPerm[1]], posPerm[2] );
			j := 1;
			while j <= 2 do
				l[edgesNr[i][j]] := n[j];
				j := j + 1;
			od;
			i := i + 1;
```

```
        od;
        return l;
end;

###################################

Solve := function( unsolved )
        #######
        #Input:             unsolved, a cube represented by a list of 54 colours
        #Precondition:      IsList( unsolved )
        #Output:            solution, a word representing a composition of group generators
        #Postcondition:     IsWord( solution) and
        #                   solution applied to unsolved results in SolvedCube( unsolved )
        #######

        local solved, solvedNr, unsolvedNr, p, solution;

        if not( IsInputCorrect( unsolved ) ) then
                return;
        fi;

        solved := SolvedCube( unsolved );
        solvedNr := [1..54];
        unsolvedNr := ColoursToNumbers( solved, unsolved );

        #Evaluate single permutation to solve cube
        p := MappingPermListList( unsolvedNr, solvedNr );

        #Handle non-existing cubes
        if not( p in cube ) then
                ErrorNoReturn( "Cube does not exist. Check input." );
        fi;

        #Decompose single permutation into group generators
        solution := PreImagesRepresentative( hom, p );

        return solution;
end;

###################################

SolveGuide := function( unsolved, n )
        #######
        #Input:             unsolved, a cube represented by a list of 54 colours and
        #                   n, a non-negativ integer
```

```
	#Precondition:		IsList( unsolved ) and
	#			IsInt( n ) and
	#			n >= 0
	#Output:		nothing is returned, solution gets printed
	#Postcondition:		none
	#######

	local solved, solvedNr, unsolvedNr, p, solution, i, j, partlysolved, middlePiecesC, c, s;

	Print( "Your initial cube:\n" );
	DisplayCube( unsolved );
	Print( "\n\n" );
	if not( IsInputCorrect( unsolved ) ) then
		return;
	fi;

	solved := SolvedCube( unsolved );
	solvedNr := [1..54];
	unsolvedNr := ColoursToNumbers( unsolved, solved );

	#Evaluate single permutation to solve cube
	p := MappingPermListList( solvedNr, unsolvedNr );

	#Handle non-existing cubes
	if not( p in cube ) then
		ErrorNoReturn( "Cube does not exist. Check input." );
	fi;

	#Decompose single permutation into group generators
	solution := PreImagesRepresentative( hom, p );

	if n = 0 then
		Print( Length( solution ), " turns required:\n", solution, "\nYour cube is then solved.\
	n_______________" );
		return;
	fi;
	partlysolved := unsolved;
	i := 1;
	j := 1;
	while i <= Length( solution ) - n do
		s := Subword( solution, i, i + n - 1 );
		Print( "Step ", j, ":\n" );
		Print( s, "\n" );
		partlysolved := Permuted( partlysolved, Image( hom, s ) );
		DisplayCube( partlysolved );
```

```
            Print( "\n\n" );
            i := i + n;
            j := j + 1;
        od;
        Print( "Step ", j, ": \n" );
        Print( Subword( solution, i, Length( solution ) ) );
        Print( "\nYour cube is now solved.\n_______________" );
        return;
end;

########################################

DistrNumberOfMoves := function( n )
        #######
        #Input:              n, an integer
        #Precondition:       IsList( unsolved ) and
        #                    IsInt( n )
        #Output:             a, a list
        #Postcondition:      IsList( a )
        #######

        local c, i, a;

        hom := GroupHomomorphismByImages( f, cube, GeneratorsOfGroup( f ), GeneratorsOfGroup( cube ) );
        a := [];
        i := 1;
        while i <= n do
            c := PreImagesRepresentative( hom, Random( cube ) );
            Add( a, Length( c ) );
            i := i + 1;
        od;
        a := Collected( a );
        return a;
end;
```

# References


[1] *Rubik's official website*. 1999. URL: https://eu.rubiks.com/about (visited on Oct. 1, 2018).

[2] David Singmaster. *Notes on Rubik's Magic Cube*. 1st ed. Enslow Publishers, 1981.

[3] Lindsey Daniels. *Group Theory and the Rubik's Cube*. 2014. URL: http://math.fon.rs/files/DanielsProject58.pdf (visited on Oct. 1, 2018).

[4] *GAP – Groups, Algorithms, and Programming, Version 4.8.10*. The GAP Group. 2018.

[5] Tomas Rokicki and Morley Davidson. *God's Number is 26 in the Quarter–Turn Metric*. 2014. URL: http://www.cube20.org/qtm (visited on Oct. 1, 2018).

[6] Tomas Rokicki et al. "The Diameter of the Rubik's Cube Group Is Twenty". In: *SIAM Journal on Discrete Mathematics* 27.2 (2013), pp. 1082–1105.

[7] Daniel Duberg and Jakob Tideström. *Comparison of Rubik's Cube Solving Methods Made for Humans*. Bachelor's Thesis. 2015. URL: http://www.diva-portal.org/smash/get/diva2:812006/FULLTEXT01.pdf (visited on Oct. 1, 2018).

[8] Dan Harris. *Speed Solving the Cube*. Sterling Publishing, 2008.

[9] Alexander H. Frey, Jr. and David Singmaster. *Handbook of Cubik Math*. Enslow Publishers, 1982.

[10] Christoph Bandelow. *Inside Rubik's Cube and Beyond*. Trans. from German by Jeannette Zehnder and Lucy Moser. Birkhäuser, 1982.

[11] David Joyner. *Adventures in Group Theory: Rubik's Cube, Merlin's Machine, and Other Mathematical Toys*. 2nd ed. Johns Hopkins University Press, 2008.