

ON GOD'S NUMBER(S) AND THE RUBIK'S SLIDE EXTENSION

by James F. Johnston III

May 9, 2016

Montclair State University, Montclair, NJ

Abstract

In a recent article, Jones, Shelton and Weaverdyck and Alm, Gramelspacher, and Rice provided two analyses of the Rubik's Slide game on a board of dimensions 3×3 . This paper extends the work of Jones, Shelton, and Weaverdyck to a board of dimensions 4×4 . Concepts from abstract algebra and graph theory are used to calculate the God's number of many classes of puzzles, which is the least number of moves necessary to reach any end configuration from any starting configuration of game play. It turns out that God's number is equivalent to the diameter of a graph of the group formed by the Rubik's Slide. Group structures are identified along with the Cayley graphs of multiple classes of puzzles and a color-based adjacency matrix is used to explore the overall group structure of the puzzle. A Hamiltonian cycle is identified for the group with supporting graphs that isolate a subgroup and its cosets. A more challenging version (hard mode) of the puzzle that is represented by a group of order 1,625,702,400 is also explored with some interesting preliminary results that will help provide further insight into the structure of this large group. Programming code (Python) used to research this puzzle is also provided, along with GAP group definitions.

Copyright ©2016 by James F. Johnston III. All rights reserved.

Acknowledgements

“A journey of a thousand miles begins with a single step.” - Laozi

When I was first approached by Diana Thomas, who at the time was my professor in an abstract algebra course, little did I realize I was taking my first step of what sometimes felt like a thousand mile journey. Without the assistance, support, and time provided by both of my advisors, Diana and Michael Jones this thesis would never have been completed. A thank you, as well, to Jonathan Cutler, a member of my thesis defense and my first graduate professor at Montclair State University. Most of all a big thank you to my wife Jess, without your support and understanding, I could never have completed this journey. To my newborn son, Xavier Orion, the world is rich with amazing things to discover and all the more beautiful because of the mathematics that supports it; I wish for you the desire to explore, to ask why and to ask how.

Contents

I	Introduction	6
1	Toy Version	8
1.1	The Rules	8
1.2	Notation	8
1.3	Group Properties	10
1.4	Graph Representations	12
1.5	On God's Number	13
II	The 4×4 Version	16
2	4×4 Easy Mode	17
2.1	Group Structure and Naming Convention	18
2.2	Hamiltonian Cycle	21
2.3	On God's Number and the 4×4 Easy Mode	22
2.4	On God's Number and the Adjacency Matrix	23
2.5	Subgames of the 4×4 Easy Mode	30
3	4×4 Hard Mode	36
3.1	Defining the Group	36
3.2	On God's Number and Subgames	36
3.3	Some 4×4 Configurations of Interest	38
III	Summary	40
IV	Works Cited	41
V	Appendix A - Python Code	42
VI	Appendix B - GAP Group Definitions	66

Part I

Introduction



Figure 0.1: The Rubik's Slide game sold by the Rubik's company

The Rubik's Slide (<http://www.rubiks.com/store/product/rubiks-slide>) is a planar game consisting of similar concepts housed in the original Rubik's Cube. The physical differences between the two games is that the original Rubik's Cube is a 3-dimensional object with transformations defined across faces of the cube and restricted by its physical movements while the Rubik's Slide is a 2-dimensional object with three definable transformations across a single face. Logical differences in game play appear as well which generate interesting mathematical structures that we were able to investigate.

The Rubik's Slide available for purchase exists as a two dimensional 3×3 electronic board made up of nine blocks. The object of the game is to move electronically colored blocks about the board by using predefined transformations from an initial starting configuration to a pre-defined final configuration. The addition of requiring one to reach the final configuration in the minimal number of moves generates several interesting mathematical investigations which form the basis of our inquiry.

In the original game, there exists 3 modes of play: easy, medium, and hard. The easy mode allows for a single coloring of up to 4 blocks with transformations right, up, and clockwise 90° , along with their respective inverses left, down, and counterclockwise. The medium mode utilizes the same coloring scheme but changes the rotation from 90° to 45° . Finally, the hard mode allows for up to two colorings for up to 4 blocks and the transformations up, right, and clockwise 45° .

Jones, an advisor to this paper, Shelton and Weaverdyck originally published an analysis of the game on a board of dimensions 3×3 and 2×2 using graph

theory and linear algebra to determine God's number for the Rubik Slide [1]. In addition, Alm, Gramelpacher, and Rice published an analysis [2] using group theory to find God's number. This thesis will be an analysis of a theoretical extension of the game on a 4×4 board of sixteen blocks. The analysis is performed for both easy and hard modes with the easy mode consisting of 16 colorings of up to 16 blocks colored with the transformations right, up, and counterclockwise 90° along with their respective inverses. The hard mode utilizes the same colorings but changes the rotation from 90° to 30° . The results of these two modes are boundaries for any lesser coloring definitions assuming one of the two sets of generating transformations are used.

A complete analysis of the easy mode results in the God's numbers of the game and subgames of the puzzle along with frequency distribution lists. The structure of the easy group is represented algebraically and graphically, a word of 64 characters is identified as a Hamiltonian cycle of the puzzle's graph, and colored adjacency matrices are presented that highlight the visual beauty and structure of groups.

The hard mode of the game is analyzed by looking at one and two color games. Important game moves are identified as well as some God's numbers of the subgames. Limitations and issues related to the size of the hard group ($2^6 \cdot (7!)^2$) are discussed.

1 Toy Version

Before we look at the 4×4 version we will look at a toy version of board size 2×2 , which is useful for demonstrating the rules and developing a set of mathematical tools which extend to analyzing the larger board.

1.1 The Rules

Every game play starts with an initial configuration of the game board, consisting of a set of one or more colored squares of one or more colors. A final configuration consisting of the same number of colored squares in a different configuration from the initial configuration. The objective is to utilize our predefined transformations and navigate from the initial configuration to the final configuration. All versions of the game restrict moves to three primary transformations along with three inverse transformations.

These transformations are defined as:

- right shift, where the color in the right position travels to the left position, in effect as a torus (inverse is a left shift)
- up shift, where the color in the top position travels to the bottom position, as well acting as a torus (inverse is a down shift)
- clockwise shift (inverse is a counterclockwise shift)

The following notation is used to represent our predefined transformations: R, R^{-1}, U, U^{-1}, C and C^{-1} respectively.

Observe in Figure 1.1 for the 2×2 board that the transformations R and R^{-1} result in exactly the same configurations of the board. This is also true for the transformations U and U^{-1} . The transformations of C and C^{-1} produce different configurations where three clockwise transformations result in the inverse transformation, $C^{-1} = C^3$.

In Figure 1.2 a game is presented with an initial configuration and a final configuration. Observe how the transformations are utilized to navigate the board and that there are multiple paths to the desired final configuration. One of the primary focuses is to determine the least number of transformations necessary to move from an initial configuration to any final configuration.

1.2 Notation

Instead of using colors it will be more convenient to number the board. This allows the configurations and transformations on the board to be represented by algebraic permutations. Cycle notation is used to describe every possible configuration and how transformations affect each configuration. An identity configuration is represented by choosing the top row from left to right as 1 and 2, respectively, then the bottom row is represented by 3 and 4. In Figure 1.3 colors are replaced by numbers and permutation notation is used to

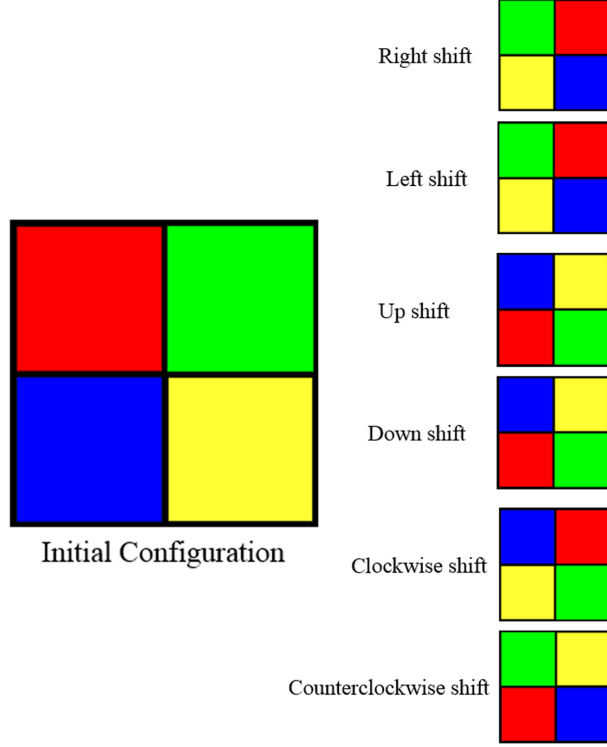


Figure 1.1: The toy version predefined transformations in action.

describe each predefined transformation. Utilizing cycle notation these transformations are defined as $R = (1\ 2)(3\ 4)$, $U = (1\ 3)(2\ 4)$, and $C = (1\ 2\ 4\ 3)$.

Let us take a side-by-side comparison of playing the game and performing permutation operations to understand more completely how to interpret our results. The identity configuration will be the initial configuration, from now referred to as $I = ()$, and the final configuration is set as $(2\ 3)$. The final configuration can be reached by the combination CR as shown in Figure 1.4. Note that the permutation operations are performed left to right for the ease of reading the game transformations. Algebraically, it is seen that $ICR = ()(1\ 2\ 4\ 3)(1\ 2)(3\ 4) = (2\ 3) = CR$. Therefore, the configuration $CR = (2\ 3)$ implies that the value of top row position one and bottom row position four do not change but the second and third positions swap.

To perform a sequence of transformations, for example $C(CR)$, it can be performed in two specific orders. Either directly as C, C, R , which is $(CC)R$ (i.e. C^2R) or as the transformation of CR on C . Using cycle notation the

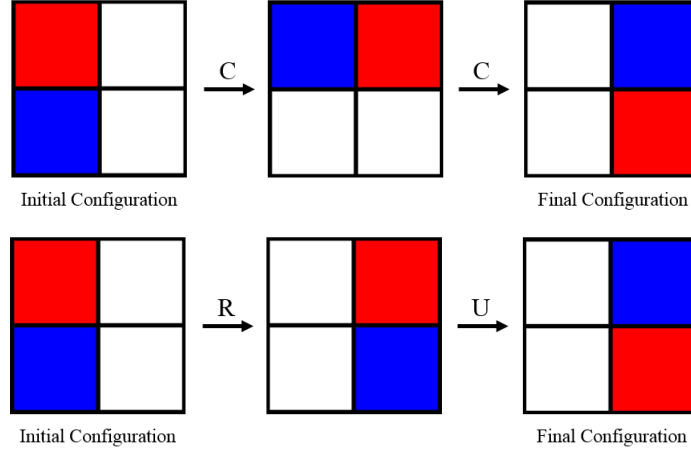


Figure 1.2: A game with two solutions displayed.

transformations become easy to calculate.

$$CR = \begin{pmatrix} 2 & 3 \end{pmatrix}$$

$$\therefore C(CR) = \begin{pmatrix} 1 & 2 & 4 & 3 \end{pmatrix} \begin{pmatrix} 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 3 \end{pmatrix} \begin{pmatrix} 2 & 4 \end{pmatrix}$$

To calculate $(CC)R$ first calculate C^2 and then apply transformation R to this result.

$$C^2 = \begin{pmatrix} 1 & 2 & 4 & 3 \end{pmatrix} \begin{pmatrix} 1 & 2 & 4 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 4 \end{pmatrix} \begin{pmatrix} 2 & 3 \end{pmatrix}$$

$$(CC)R = \begin{pmatrix} 1 & 4 \end{pmatrix} \begin{pmatrix} 2 & 3 \end{pmatrix} \begin{pmatrix} 1 & 2 \end{pmatrix} \begin{pmatrix} 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 3 \end{pmatrix} \begin{pmatrix} 2 & 4 \end{pmatrix} = C^2R$$

The configurations with permutation composition are associative but they are not abelian, which is easily seen by the following comparison.

$$(CR)C = \begin{pmatrix} 2 & 3 \end{pmatrix} \begin{pmatrix} 1 & 2 & 4 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 2 \end{pmatrix} \begin{pmatrix} 3 & 4 \end{pmatrix} \neq C(CR)$$

There are multiple ways to describe specific configurations of the board therefore it is important to develop a consistent denotation. By starting with the identity configuration, I , it is possible to generate all other configurations by only applying R and C transformations to all resulting configurations until you exhaust all possible configurations. It is unnecessary to include the U transformation because it is actually a composition of R and C , specifically $U = C^2R$. The toy version only has 8 possible configurations and we denote them as $\{I, R, C, CR, C^2, C^2R, C^3, C^3R\}$.

1.3 Group Properties

Every version of the Rubik's Slide paired with the permutation operation is an algebraic group because it has the four following properties:

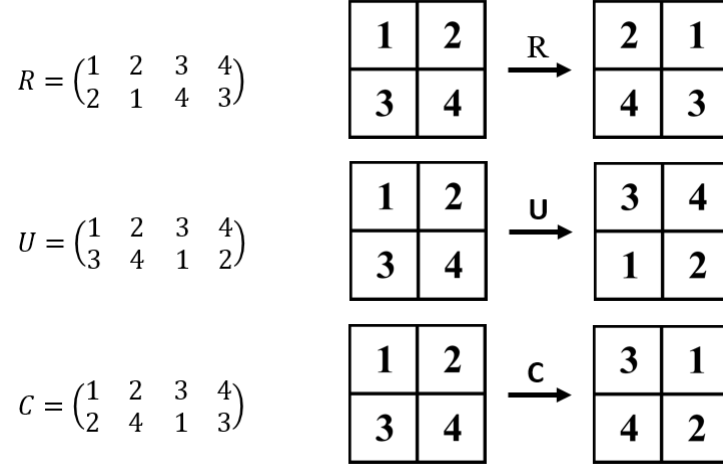


Figure 1.3: A permutation perspective of the toy version.

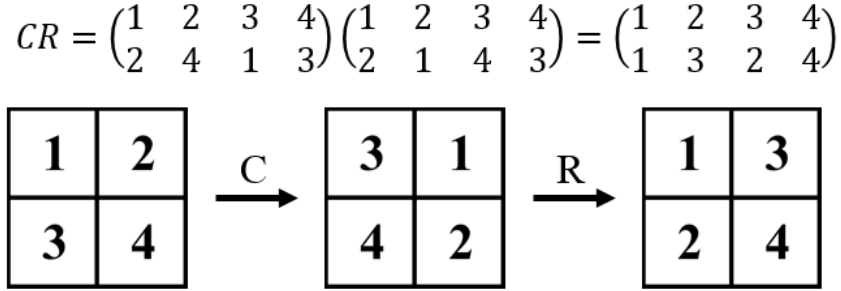


Figure 1.4: Permutation operations compared to actual board transformations.

1. An identity configuration, $I = ()$, exists.
2. The transformations on the configurations are associative.
3. For each final configuration that can be reached from some starting configuration by a sequence of transformations there exists an inverse sequence of transformations that will return the board to the starting configuration . (The list of inverses for the toy version is provided below.)
4. Closure exists (i.e. the set of all possible configurations achieved is closed).

As found [1], the group associated with the toy version is the dihedral group of order 8, D_4 . It is generated by R and C and can be written as,

$$\langle R, C | C^4 = R^2 = I, RCR^{-1} = C^{-1} \rangle$$

Each configuration of the puzzle is also considered to be an element of the group so “configuration” and “element” are interchangeable. The inverses for each element of the group are:

$$I^{-1} = I, R^{-1} = R, C^{-1} = C^3, (CR)^{-1} = CR,$$

$$(C^2)^{-1} = C^2, (C^2R)^{-1} = C^2R, (C^3)^{-1} = C, (C^3R)^{-1} = C^3R$$

1.4 Graph Representations

In order to gain insight, we can represent aspects of the game using graphs. Each configuration also can be viewed as a vertex of a graph where the transformations that connect each vertex are viewed as edges. A useful mathematical tool is to construct the group as a graph with directed edges. A graph representation in this manner is called a Cayley graph (see Fraleigh [4] pg. 70 for a more explicit definition). Figure 1.5 displays two Cayley graphs of the toy version, one only containing R and C transformations and the other containing R, U and C transformations. Since R and C generate the group the number of vertices remain the same in both Cayley graphs but the number of edges differs between the two graphs. Consider the U transformation as a set of shortcuts that allows one to travel from vertex to vertex with less transformations. The Cayley graph without the U transformation contains 8 vertices and 12 edges while the Cayley graph with the U transformation has 8 vertices and 16 edges.

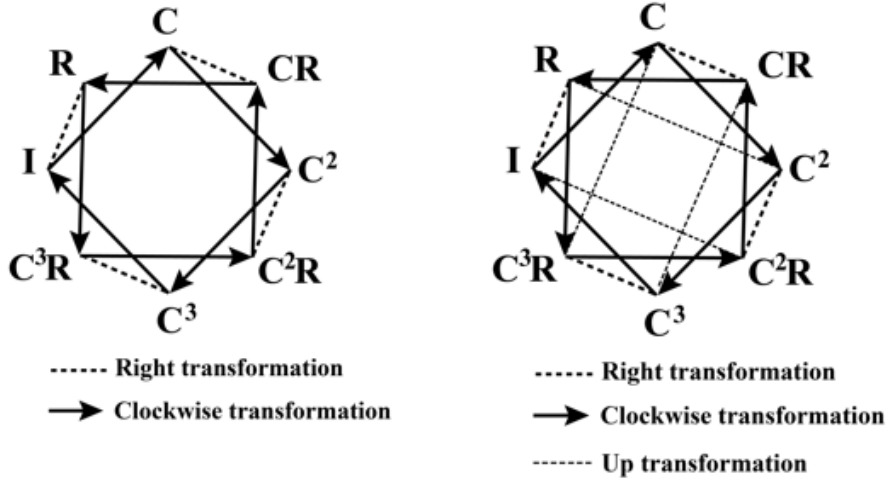


Figure 1.5: The toy version displayed as two Cayley graphs.

A Hamiltonian path is a sequence of transformations that walks one through every vertex exactly once. If a path is identified such that the walk begins and

ends on the same vertex then this path is called a Hamiltonian cycle. Finding a cycle is useful because that means a sequence of transformations or a word, in graph theory terminology, has been identified that guarantees a method of solving every game given any starting position. Think about what this means, given a random starting configuration we have a sequence of transformations that visits each vertex therefore no matter what final configuration is chosen the walk will eventually reach the final configuration.

Hamiltonian paths can be difficult to find as the number of vertices increase. It is a significant problem of interest to mathematicians and computer scientists alike and is an NP-complete problem, meaning there exists no efficient algorithm to find such a path within polynomial time.

In Figure 1.6 a Hamiltonian cycle is displayed and is represented by the word $CCRCRCCR$. Algebraically it can be shown that this word is Hamiltonian cycle by way of calculating $CCRCRCCR = C^3RC^3R = (C^3R)(C^3R)^{-1} = I$. Using the Cayley graphs in Figure 1.5 it is also possible to visually walk through each vertex using the word.

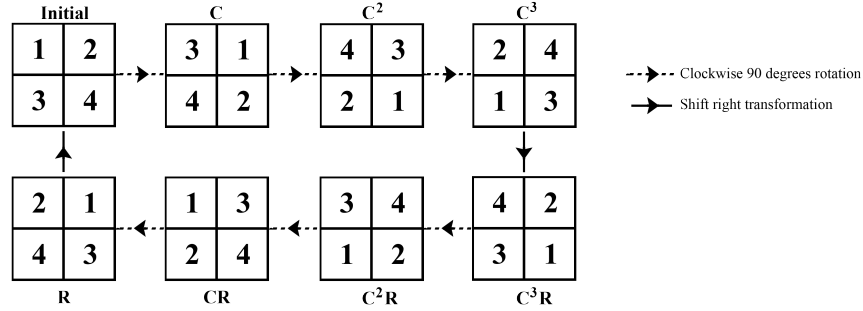


Figure 1.6: A Hamiltonian cycle through the toy version.

1.5 On God's Number

If an omniscient being were to play a game, it would be played by utilizing the most efficient set of transformations to reach any final configuration from any initial configuration, the least number of steps required for this omniscient being to succeed is called God's number. In graph theory terminology, God's number is the maximum eccentricity of any two vertices in a graph, in which the eccentricity of a vertex is the maximum distance between itself any other vertex in the graph. This is also referred to as the diameter of the graph.

Finding God's number in a small graph can be determined by inspecting the Cayley graph in which all three transformations R , U and C are present. Observe that vertex I is adjacent to four vertices and non-adjacent to three vertices, specifically CR , C^2 , and C^3R . Performing one transformation it is possible to navigate to the remaining non-adjacent vertices. Therefore the diameter of the

graph is found to be 2, which is the God's number of the toy version.

Another method of finding the diameter of the graph is by recognizing a few properties this graph holds. The toy version is a strongly 4-regular graph (see Bollobas [3] pg.4 for more information on k -regular graphs). It is 4-regular because every vertex is of degree four and from the viewpoint of any of the vertices it would be impossible to identify which vertex one is located at. It is strongly regular because every two adjacent vertices share the same number of common vertices, 1, and every two non-adjacent vertices share the same number of common vertices, 4. It has been proven that every strongly regular graph that has two non-adjacent vertices and that shares at least one common vertex has diameter of 2.

Additionally, the toy version can be represented as a bipartite graph as displayed in Figure 1.7. A bipartite graph is a graph that consists of two disjoint sets of vertices where each set contains only non-adjacent vertices. Visually, this is the easiest way to observe the diameter of this graph. Similar to the explanation used for reading the Cayley graph to determine God's number, you can reach any non-adjacent vertex by way of two simple transformations to any adjacent vertex in the opposing set and then back to the non-adjacent vertex.

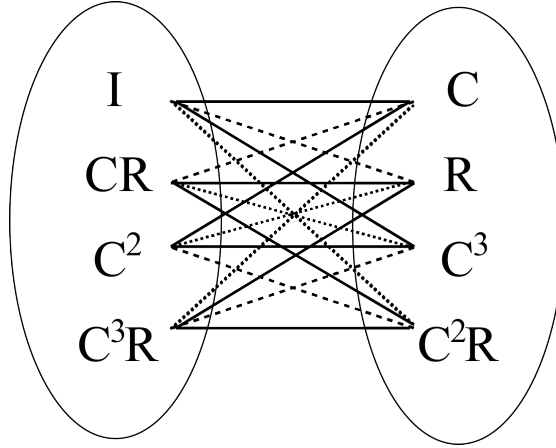


Figure 1.7: The toy version represented as a bipartite graph.

The final tool to be utilized for finding the diameter of the graph will be through the power of an adjacency matrix (see Babat Ch.3 for more theorems on adjacency matrices). To build an adjacency matrix A , of the graph G , each row and column represent a vertex $v_i \in V(G)$. Therefore the toy version adjacency matrix is an 8×8 matrix. For each entry of our matrix A , if the vertices, v_i, v_j share an edge then $A_{i,j} = 1$ otherwise the entry is 0. This implies that within one transformation these vertices are reachable from the given vertex. Since the graph is 4-regular every row and column will contain exactly four 1's in our initial adjacency matrix.

The value of the entry A_{ij}^n where $n \geq 1$ is the number of different $v_i - v_j$ walks

of length n in G . To discover the least amount of transformations required to visit each vertex from any other vertex you must identify the minimum degree of the polynomial $I + A + A^2 + \cdots + A^n$ such that each entry of the resulting matrix has no 0 entries, the number n is the diameter/God's number of the graph. The matrix represented by $I + A + A^2 + \cdots + A^n$ is also known as an n – *walk*. Another way of stating God's number n , is as the smallest n – *walk* such that there exists no 0 entries. Figure 1.8 displays the adjacency matrices representing both the 1 – *walk* and 2 – *walk* of the toy version, hence God's number is 2.

$$\begin{array}{c}
 \begin{array}{c} I \\ R \\ C \\ CR \\ C^2 \\ C^2R \\ C^3 \\ C^3R \end{array} \begin{bmatrix} I & R & C & CR & C^2 & C^2R & C^3 & C^3R \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \\
 I+A
 \end{array}
 \qquad
 \begin{array}{c}
 \begin{array}{c} I \\ R \\ C \\ CR \\ C^2 \\ C^2R \\ C^3 \\ C^3R \end{array} \begin{bmatrix} I & R & C & CR & C^2 & C^2R & C^3 & C^3R \\ 5 & 1 & 1 & 4 & 4 & 1 & 1 & 4 \\ 1 & 5 & 4 & 1 & 1 & 4 & 4 & 1 \\ 1 & 4 & 5 & 1 & 1 & 4 & 4 & 1 \\ 4 & 1 & 1 & 5 & 4 & 1 & 1 & 4 \\ 4 & 1 & 1 & 4 & 5 & 1 & 1 & 4 \\ 1 & 4 & 4 & 1 & 1 & 5 & 4 & 1 \\ 1 & 4 & 4 & 1 & 1 & 4 & 5 & 1 \\ 4 & 1 & 1 & 4 & 4 & 1 & 1 & 5 \end{bmatrix} \\
 I+A+A^2
 \end{array}$$

Figure 1.8: The toy version 1 – *walk* and 2 – *walk* adjacency matrices.

Part II

The 4×4 Version

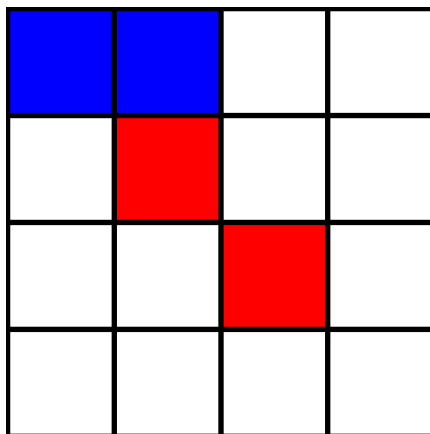


Figure 1.9: A sample 4×4 version initial configuration.

Using the tools developed and explored on the toy version, we approach the 4×4 version of the Rubik's Slide. This game is not currently manufactured by the Rubik's company. A digital version is available for download at (<http://www.jimmyjohnston.org/research/rubikslide/>). Because this game is a theoretical version of a smaller game, I have adjusted the rules using the original game rules as a guide.

3×3 Original Game Rules

1. The **easy mode** consists of 1-color board pieces with a total of 4 colored blocks per configuration. Three transformations, right, up and clockwise 90° rotation, along with their respective inverses are allowed.
2. The **medium mode** consists of the same coloring configuration and transformations except for the rotation changes to a clockwise 45° rotation and its respective inverse.
3. The **hard mode** consists of 2-colored board pieces with a total of 5 colored blocks per configuration. Three transformations right, up and clockwise 45° rotation, along with their respective inverses are allowed.

4×4 Adjusted Game Rules The adjusted rules provide a bigger picture of the group structure. There exists only two rules for the 4×4 version, easy and hard.

1. The **easy mode** defines up to 16 unique colors and up to 16 colored blocks using the transformations right, up, and clockwise 90° rotation along with their respective inverses.
2. The **hard mode** consists of the same coloring configuration and transformations except for the rotation changes to a clockwise 30° rotation and its respective inverse.

2 4×4 Easy Mode

As with the toy version, the R , U , and C are similar transformations. In Figure 2.1 the effect of operating on a larger game board requires updated permutations.

$$\begin{aligned}
 R &= (1\ 2\ 3\ 4)(5\ 6\ 7\ 8)(9\ 10\ 11\ 12)(13\ 14\ 15\ 16) \\
 U &= (1\ 13\ 9\ 5)(2\ 14\ 10\ 6)(3\ 15\ 11\ 7)(4\ 16\ 12\ 8) \\
 C &= (1\ 4\ 16\ 13)(2\ 8\ 15\ 9)(3\ 12\ 14\ 5)(6\ 7\ 11\ 10)
 \end{aligned}$$

Recall from the toy version the transformations R and U were inverses of themselves and $C^{-1} = C^3$. In the 4×4 easy mode the inverses of the generating transformations are $R^{-1} = R^3$, $U^{-1} = U^3$, $C^{-1} = C^3$.

The objective of the game remains the same, given an initial configuration and final configuration utilize the predefined transformations to navigate from the initial configuration to your destination the final configuration. Figure 2.2 demonstrates a game with two possible solutions.

The group generated by the configurations and their transformations is non-abelian as well, observe that $CR \neq RC$, which are two unique elements in our group.

$$\begin{aligned}
 CR &= (1\ 4\ 16\ 13)(2\ 8\ 15\ 9)(3\ 12\ 14\ 5)(6\ 7\ 11\ 10) \\
 &\quad (1\ 2\ 3\ 4)(5\ 6\ 7\ 8)(9\ 10\ 11\ 12)(13\ 14\ 15\ 16) \\
 &= (2\ 5\ 4\ 13)(3\ 9)(6\ 8\ 16\ 14)(7\ 12\ 15\ 10)
 \end{aligned}$$

$$\begin{aligned}
 RC &= (1\ 2\ 3\ 4)(5\ 6\ 7\ 8)(9\ 10\ 11\ 12)(13\ 14\ 15\ 16) \\
 &\quad (1\ 4\ 16\ 13)(2\ 8\ 15\ 9)(3\ 12\ 14\ 5)(6\ 7\ 11\ 10) \\
 &= (1\ 8\ 3\ 16)(2\ 12)(5\ 7\ 15\ 13)(6\ 11\ 14\ 9)
 \end{aligned}$$

$$\therefore CR \neq RC$$

Figure 2.3 presents CR as configurations of the board with a subgame embedded in it. Notice that even though only four boxes are colored all boxes

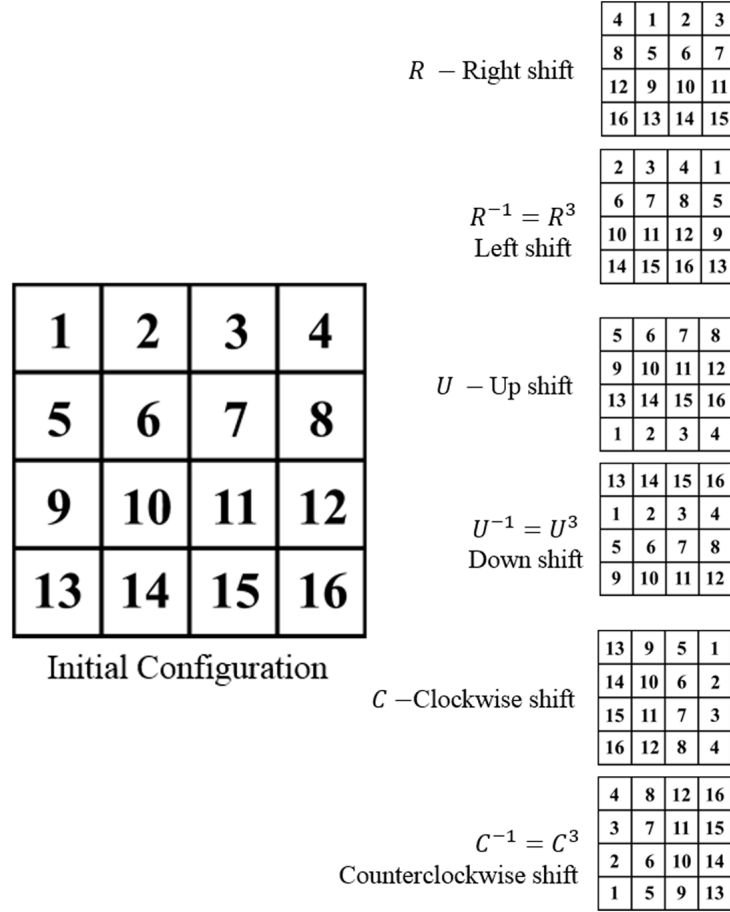


Figure 2.1: The 4×4 easy mode transformations.

do transform. From the perspective of considering a specific game it is possible to discuss such a subgame while discussing the complete group and its permutations.

2.1 Group Structure and Naming Convention

The easy mode of the 4×4 game has a group that contains 64 elements; as a graph there are 64 vertices and 192 edges, a summary of different board sizes and their vertex/edge counts is presented in Figure 2.4. Using GAP software, $\langle C, R, U | R^4 = U^4 = C^4 = I, RCU = C \rangle$ is identified as a series of semidirect products of the cyclic group C_2 with the direct product of C_4 and C_2 , $((((C_4 \times C_2) \rtimes C_2) \rtimes C_2) \rtimes C_2)$. The naming convention chosen here to represent every vertex of our graph is $C^i R^j U^k$ such that $i, j, k \in \{0, 1, 2, 3\}$, this

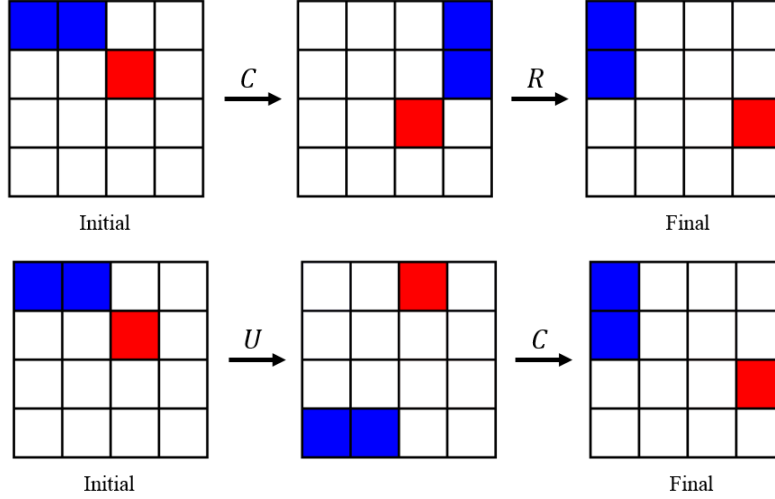


Figure 2.2: Game demonstration with two possible solutions.

$$CR = (2\ 5\ 4\ 13)(3\ 9)(6\ 8\ 16\ 14)(7\ 12\ 15\ 10)$$

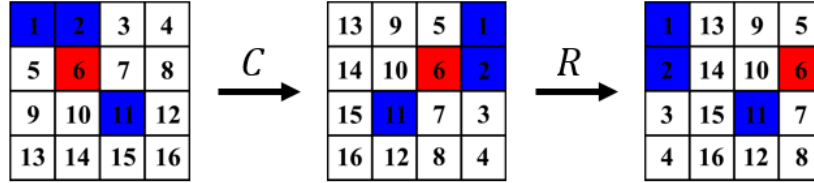


Figure 2.3: CR with a subgame embedded.

convention is not unique (i.e. $CR = UC$) since there are many ways to name the vertex and this will allow us to avoid confusion when communicating movements through the graph.

A subgroup of the 4×4 easy mode is

$$H = \langle R, U | R^4 = U^4 = I, RU = UR \rangle,$$

which is an abelian subgroup that provides a rich structure for moving about the graph of its parent group. The left cosets of this subgroup, H, cH, c^2H, c^3H , are displayed in Figure 2.5. One can travel from $I \rightarrow C \rightarrow C^2 \rightarrow C^3 \rightarrow I$ by clockwise transformations but for the remaining vertices, for example $R \rightarrow CR$, they are not directly connected by a clockwise transformation because the parent group is not abelian. R is connected to CU in cH by way of a clockwise transformation, $RC = CU$.

Configuration Size	Vertices	Edges
2 x 2	8	16
3 x 3	36	108
4 x 4	64	192

Figure 2.4: Summary of graph information for easy modes.

Lemma 1.

$$UR = \begin{pmatrix} 1 & 14 & 11 & 8 \end{pmatrix} \begin{pmatrix} 2 & 15 & 12 & 5 \end{pmatrix} \begin{pmatrix} 3 & 16 & 9 & 6 \end{pmatrix} \begin{pmatrix} 4 & 13 & 10 & 7 \end{pmatrix}$$

Proof.

$$\begin{aligned}
UR &= \begin{pmatrix} 1 & 13 & 9 & 5 \end{pmatrix} \begin{pmatrix} 2 & 14 & 10 & 6 \end{pmatrix} \begin{pmatrix} 3 & 15 & 11 & 7 \end{pmatrix} \begin{pmatrix} 4 & 16 & 12 & 8 \end{pmatrix} \\
&\quad \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 & 7 & 8 \end{pmatrix} \begin{pmatrix} 9 & 10 & 11 & 12 \end{pmatrix} \begin{pmatrix} 13 & 14 & 15 & 16 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 14 & 11 & 8 \end{pmatrix} \begin{pmatrix} 2 & 15 & 12 & 5 \end{pmatrix} \begin{pmatrix} 3 & 16 & 9 & 6 \end{pmatrix} \begin{pmatrix} 4 & 13 & 10 & 7 \end{pmatrix}
\end{aligned}$$

Theorem 1. *The subgroup $H = \langle R, U \rangle$ generated by R and U is abelian and of order 16.*

Proof.

$$\begin{aligned}
RU &= \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 & 7 & 8 \end{pmatrix} \begin{pmatrix} 9 & 10 & 11 & 12 \end{pmatrix} \begin{pmatrix} 13 & 14 & 15 & 16 \end{pmatrix} \\
&\quad \begin{pmatrix} 1 & 13 & 9 & 5 \end{pmatrix} \begin{pmatrix} 2 & 14 & 10 & 6 \end{pmatrix} \begin{pmatrix} 3 & 15 & 11 & 7 \end{pmatrix} \begin{pmatrix} 4 & 16 & 12 & 8 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 14 & 11 & 8 \end{pmatrix} \begin{pmatrix} 2 & 15 & 12 & 5 \end{pmatrix} \begin{pmatrix} 3 & 16 & 9 & 6 \end{pmatrix} \begin{pmatrix} 4 & 13 & 10 & 7 \end{pmatrix} = UR
\end{aligned}$$

Since $R^4 = U^4 = I$, $\langle R, U \rangle = \{R^i U^j : i, j = 0, 1, 2, 3\}$. All $R^i U^j$ are found to be distinct by inspecting the structure.

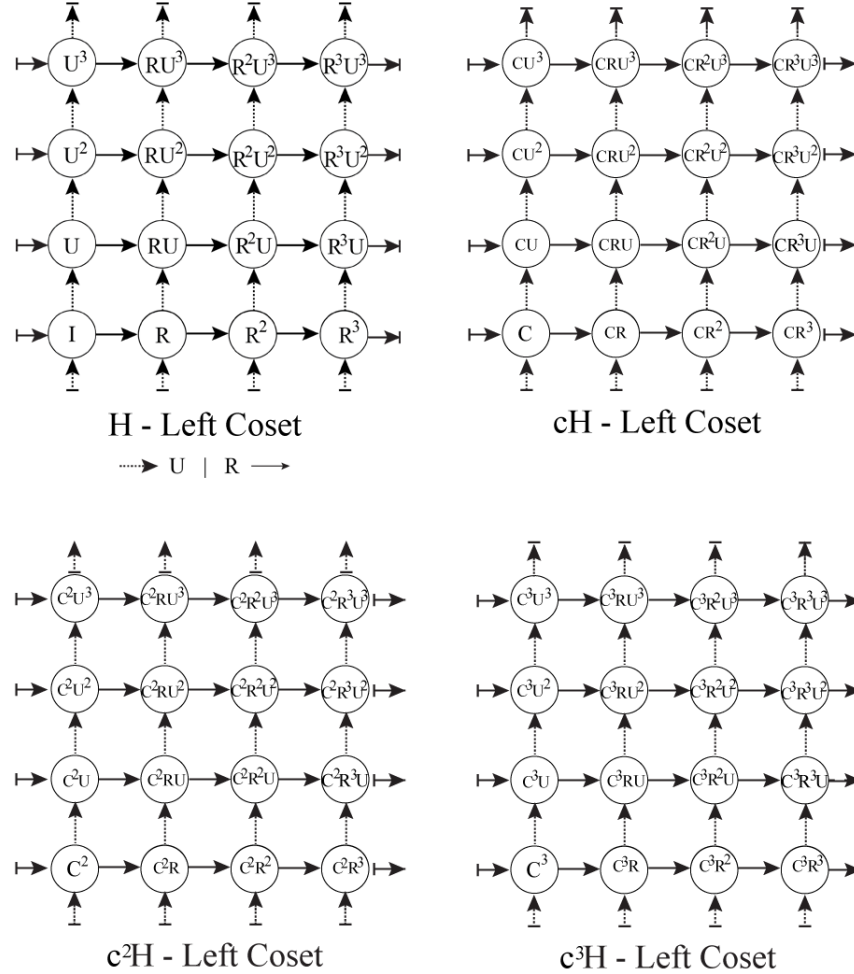


Figure 2.5: The cosets of subgroup $H = \langle R, U \rangle$.

2.2 Hamiltonian Cycle

The benefit of having the subgroup $H = \langle R, U | R^4 = U^4 = I, RU = UR \rangle$ structure is that it provides an easy method for identifying a Hamiltonian cycle. This is achieved by walking through each vertex in each coset and moving from coset to coset. Start at element $I \in \text{coset } H$ and move directly to coset cH by way of a clockwise transformation on I . Walk through all vertices in cH using the word $RRRURRRURRRURRR$ and then exit the coset by a clockwise transformation into c^2H . This pattern is repeated for each coset until returning to H where a final iteration of the word returns one to I . The word for the complete Hamiltonian cycle is $(CRRRURRRURRRURRR)^4$.

Lemma 2.

$$U^3 = \begin{pmatrix} 1 & 5 & 9 & 13 \end{pmatrix} \begin{pmatrix} 2 & 6 & 10 & 14 \end{pmatrix} \begin{pmatrix} 3 & 7 & 11 & 15 \end{pmatrix} \begin{pmatrix} 4 & 8 & 12 & 16 \end{pmatrix}$$

and

$$CU^3 = \begin{pmatrix} 1 & 8 & 3 & 16 \end{pmatrix} \begin{pmatrix} 2 & 12 \end{pmatrix} \begin{pmatrix} 5 & 7 & 15 & 13 \end{pmatrix} \begin{pmatrix} 6 & 11 & 14 & 9 \end{pmatrix}$$

Proof.

$$U^3 = [(1 \ 13 \ 9 \ 5) (2 \ 14 \ 10 \ 6) (3 \ 15 \ 11 \ 7) (4 \ 16 \ 12 \ 8)]^3 \\ = (1 \ 5 \ 9 \ 13) (2 \ 6 \ 10 \ 14) (3 \ 7 \ 11 \ 15) (4 \ 8 \ 12 \ 16)$$

$$CU^3 = \begin{pmatrix} 1 & 4 & 16 & 13 \end{pmatrix} \begin{pmatrix} 2 & 8 & 15 & 9 \end{pmatrix} \begin{pmatrix} 3 & 12 & 14 & 5 \end{pmatrix} \begin{pmatrix} 6 & 7 & 11 & 10 \end{pmatrix} \\ \begin{pmatrix} 1 & 5 & 9 & 13 \end{pmatrix} \begin{pmatrix} 2 & 6 & 10 & 14 \end{pmatrix} \begin{pmatrix} 3 & 7 & 11 & 15 \end{pmatrix} \begin{pmatrix} 4 & 8 & 12 & 16 \end{pmatrix} \\ = \begin{pmatrix} 1 & 8 & 3 & 16 \end{pmatrix} \begin{pmatrix} 2 & 12 \end{pmatrix} \begin{pmatrix} 5 & 7 & 15 & 13 \end{pmatrix} \begin{pmatrix} 6 & 11 & 14 & 9 \end{pmatrix}$$

Theorem 2. *The word $(CRRRURRRURRRURRR)^4$ is a Hamiltonian cycle of the group generated by $\langle C, R, U \rangle$.*

Proof. $(CRRRURRRURRRURRR)^4 = (CR^3UR^3UR^3UR^3)^4 = (CR^{12}U^3)^4$ because the cosets of subgroup $H = \langle R, U \rangle$ are abelian and $(CR^{12}U^3)^4 = (CU^3)^4 = [(1\ 8\ 3\ 16)(2\ 12)(5\ 7\ 15\ 13)(6\ 11\ 14\ 9)]^4 = I$, this shows us that we will end on our starting configuration. Also observe in Figure 2.5 that the initial C rotation places us into one of the cosets of H and the word $RRRURRRURRRURRR$ traverses us across each element of the coset before invoking another C rotation moving us to the next coset. Therefore we travel through every element of the group and end where we initially started, hence a Hamiltonian cycle has been found.

2.3 On God's Number and the 4×4 Easy Mode

The God's number or the diameter of the graph can easily be determined by analyzing the coset structure of subgroup $H = \langle R, U \rangle$. Choose vertex I as the initial configuration and locate the furthest vertex within coset H from I . Through quick inspection of the coset graph in Figure 2.5 the furthest vertex from I in coset H is R^2U^2 . It can also be observed that the vertices coset H are neighbors with vertices within cosets cH and c^3H by either a clockwise or counterclockwise rotation. The only coset not directly connected to H is c^2H , which is exactly two clockwise or two counterclockwise rotations away. It can

be deduced that the vertices that are most distant from elements in H are those in c^2H . From any coset you can reach any elements within that coset by at most 4 transformations. Therefore the diameter of the graph or God's number is $4 + 2 = 6$.

Lemma 4. *The maximum distance of any vertex y in the subgroup H generated by $\langle R, U \rangle, y \in V(H)$ from I is the maximum distance from any other vertex in $H, x \in V(H)$ to y*

$$\max_{x,y} d(x, y) = \max_y d(I, y), \forall x, y \in V(H)$$

Proof. Through inspection of H this is easily observed.

Lemma 5. *The distance, $d(I, R^2U^2) = 4$ and the diameter of subgroup $H = 4$*

$$\text{diam}H = \max_{x,y} d(x, y) = \max_y d(I, y) = d(I, R^2U^2) = 4, \forall x, y \in V(H)$$

Proof. Through inspection it is found that $d(I, y) < d(I, R^2U^2) = 4, \forall y \in V(H)$. $\text{diam}H = 4$ by Lemma 4 and the definition of diameter of a graph.

Lemma 6. *The maximum distance between the cosets of subgroup $H = \langle R, U \rangle$ is 2.*

Proof. The left cosets of $H = \langle R, U \rangle$ are H, cH, c^2H, c^3H , through inspection the maximum distance is 2, the distance between cosets H and c^2H and also cosets cH and c^3H .

Theorem 2. *The diameter of the graph, G of the 4×4 easy version Rubik's Slide is 6.*

Proof. The distance between cosets of subgroup H is at most 2. Each coset have the same diameter of 4 and Lemma 4 implies that the distance between any two vertices, $v_i v_j \in V(G) \leq 6$.

2.4 On God's Number and the Adjacency Matrix

It is also possible to find the diameter of the graph by forming an adjacency matrix as demonstrated in the toy version. The adjacency matrix for the 4×4 easy mode of the Rubik's Slide is a 64×64 matrix with each vertex representing the rows and columns. Figures 2.6-10 display some of the 4×4 easy mode adjacency matrices where $A_n = I + A + A^2 + \dots + A^n$ for $n \in \{1, 2, 3, 4, 6\}$ this will be referred to as n -walks, Table 1 provides the key describing each vertex represented in the adjacency colored matrices. The images represent the matrix A_n if an entry is the color white that implies a 0-entry in the matrix, a color represents a non-0-entry. Colors were chosen since the colored image is

more graphically pleasing and we are solely interested in the existence of paths between two vertices.

The color black represents the vertices along the diagonal which is $A_0 = I$ these vertices are immediately reached without any transformations because there are 0 edges necessary between a vertex and itself. In Figure 2.6 the 1 – *walk* includes the color blue where blue represents all vertices that share an edge. The color teal in Figure 2.7 represents all vertices that are exactly two hops (a hop is the number of edges traveled from some vertex C to another vertex D) away from themselves. For the remaining figures, the color green represents 3 hops, yellow 4 hops, pink 5 hops, and red 6 hops.

In Figure 2.10, the 6 – *walk*, the entries colored red, require the maximum number of hops to reach each other, which in this puzzle is 6. For example, the path between vertex 1 – I and vertex 43 – $C^2R^2U^2$ are colored red, therefore a minimum of 6 transformations must be used to reach the other vertex. One such path from vertex I to vertex $C^2R^2U^2$ is $I \rightarrow C \rightarrow C^2 \rightarrow C^2R \rightarrow C^2R^2 \rightarrow C^2R^2U \rightarrow C^2R^2U^2$ which you can trace in Figure 2.5. These vertices are reached by the sequence of transformations C, C, R, R, U, U . Some additional observations about the adjacency matrix images are that they are constructed so that the vertices are grouped by cosets of the $H = \langle R, U \rangle$ subgroup. This ordering shows that by the 4 – *walk* the nodes contained within the specific cosets have all been reached and that by the 6 – *walk* all entries are non-zero entries implying that every vertex has a sequence of transformations of length 6 or smaller that connects them to every other vertex in the graph, 6 is God's number.

Coset	H		cH		c^2H		c^3H
1.	I	17.	C	33.	C^2	49.	C^3
2.	U	18.	CU	34.	C^2U	50.	C^3U
3.	U^2	19.	CU^2	35.	C^2U^2	51.	C^3U^2
4.	U^3	20.	CU^3	36.	C^2U^3	52.	C^3U^3
5.	R	21.	CR	37.	C^2R	53.	C^3R
6.	RU	22.	CRU	38.	C^2RU	54.	C^3RU
7.	RU^2	23.	CRU^2	39.	C^2RU^2	55.	C^3RU^2
8.	RU^3	24.	CRU^3	40.	C^2RU^3	56.	C^3RU^3
9.	R^2	25.	CR^2	41.	C^2R^2	57.	C^3R^2
10.	R^2U	26.	CR^2U	42.	C^2R^2U	58.	C^3R^2U
11.	R^2U^2	27.	CR^2U^2	43.	$C^2R^2U^2$	59.	$C^3R^2U^2$
12.	R^2U^3	28.	CR^2U^3	44.	$C^2R^2U^3$	60.	$C^3R^2U^3$
13.	R^3	29.	CR^3	45.	C^2R^3	61.	C^3R^3
14.	R^3U	30.	CR^3U	46.	C^2R^3U	62.	C^3R^3U
15.	R^3U^2	31.	CR^3U^2	47.	$C^2R^3U^2$	63.	$C^3R^3U^2$
16.	R^3U^3	32.	CR^3U^3	48.	$C^2R^3U^3$	64.	$C^3R^3U^3$

Table 1: Key for adjacency matrices in Figures 2.6-10

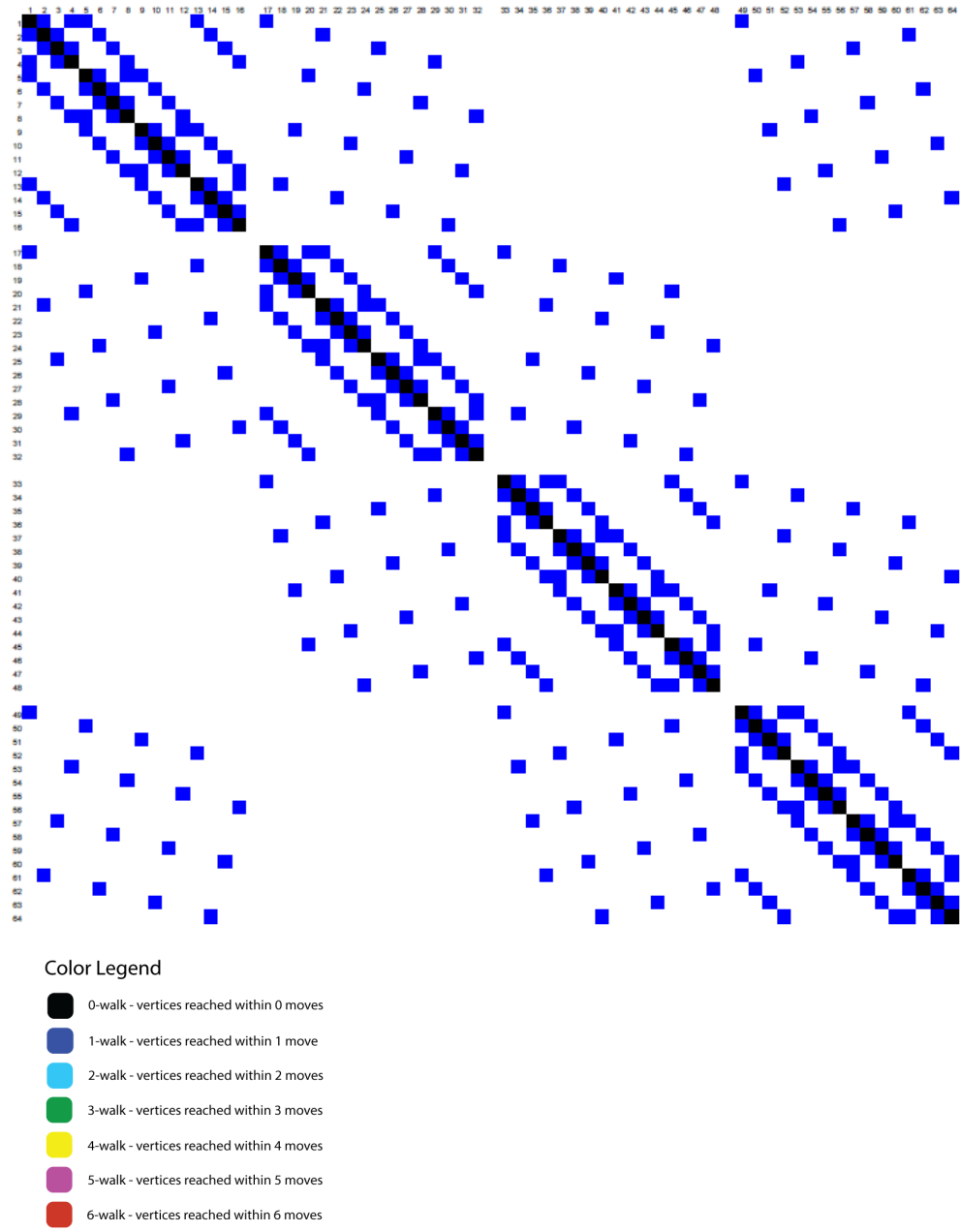


Figure 2.6: $1 - walk$ adjacency matrix organized by cosets of the subgroup $H = \langle R, U \rangle$

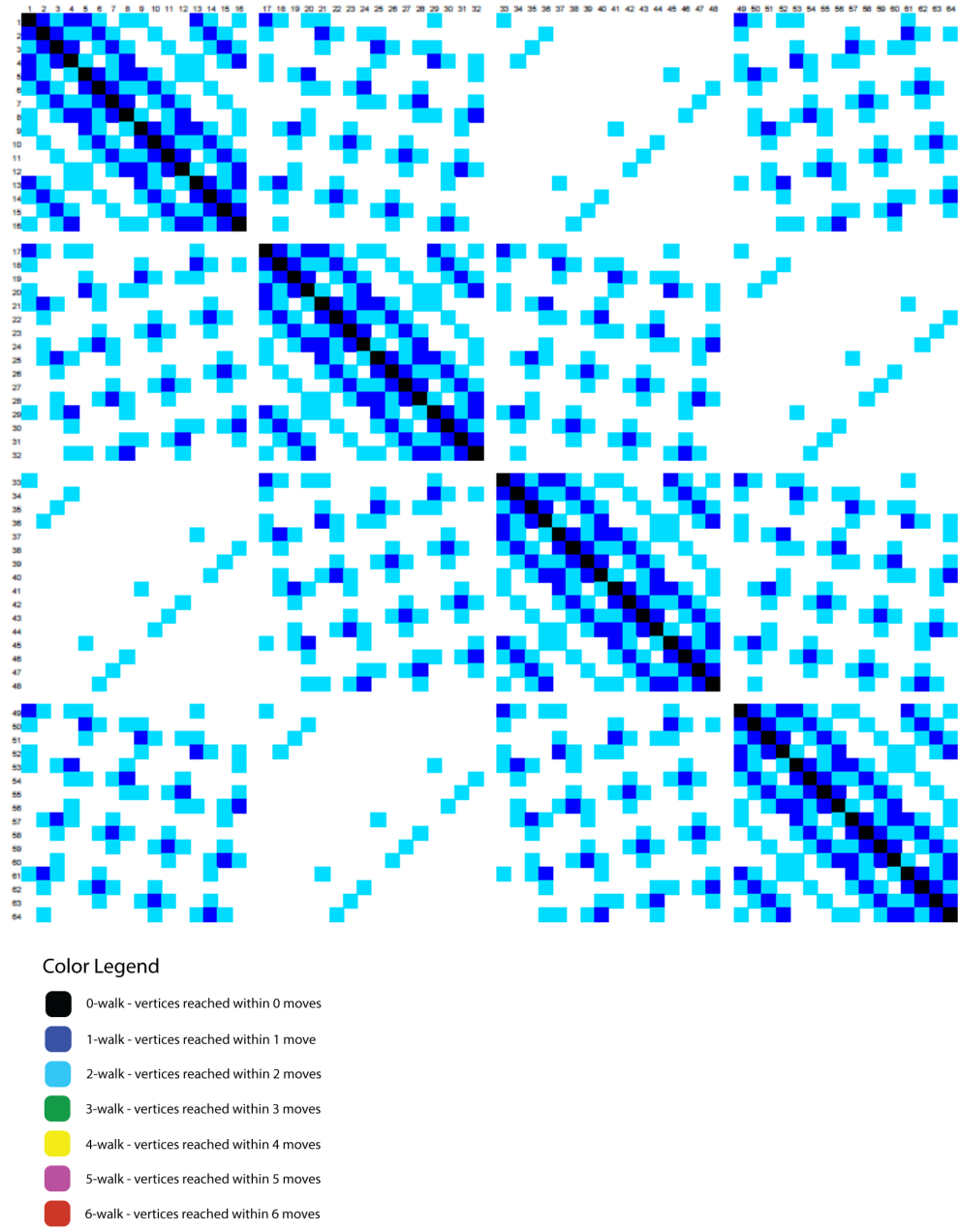


Figure 2.7: 2-walk adjacency matrix organized by cosets of the subgroup $H = \langle R, U \rangle$

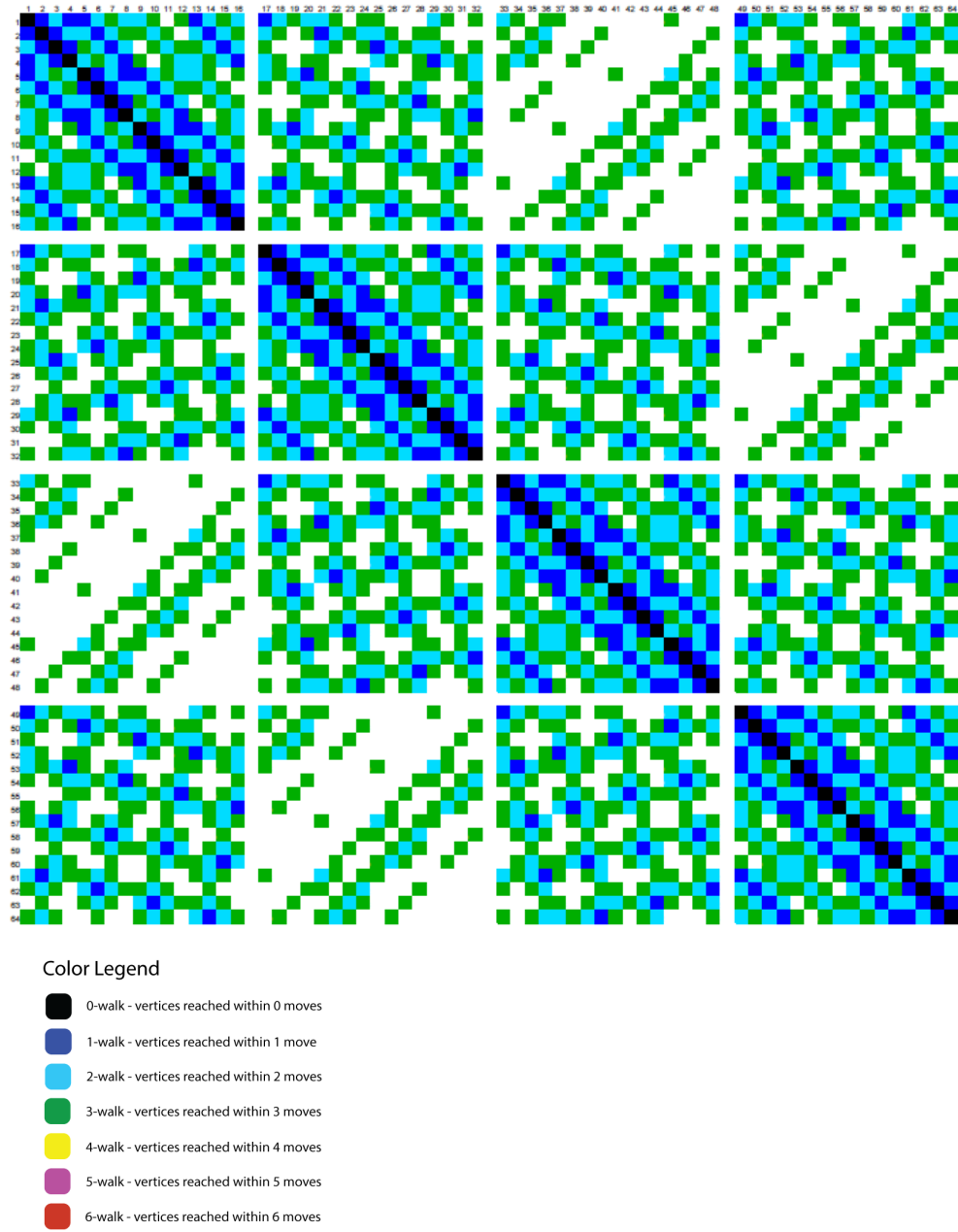


Figure 2.8: 3 - walk adjacency matrix organized by cosets of the subgroup $H = \langle R, U \rangle$

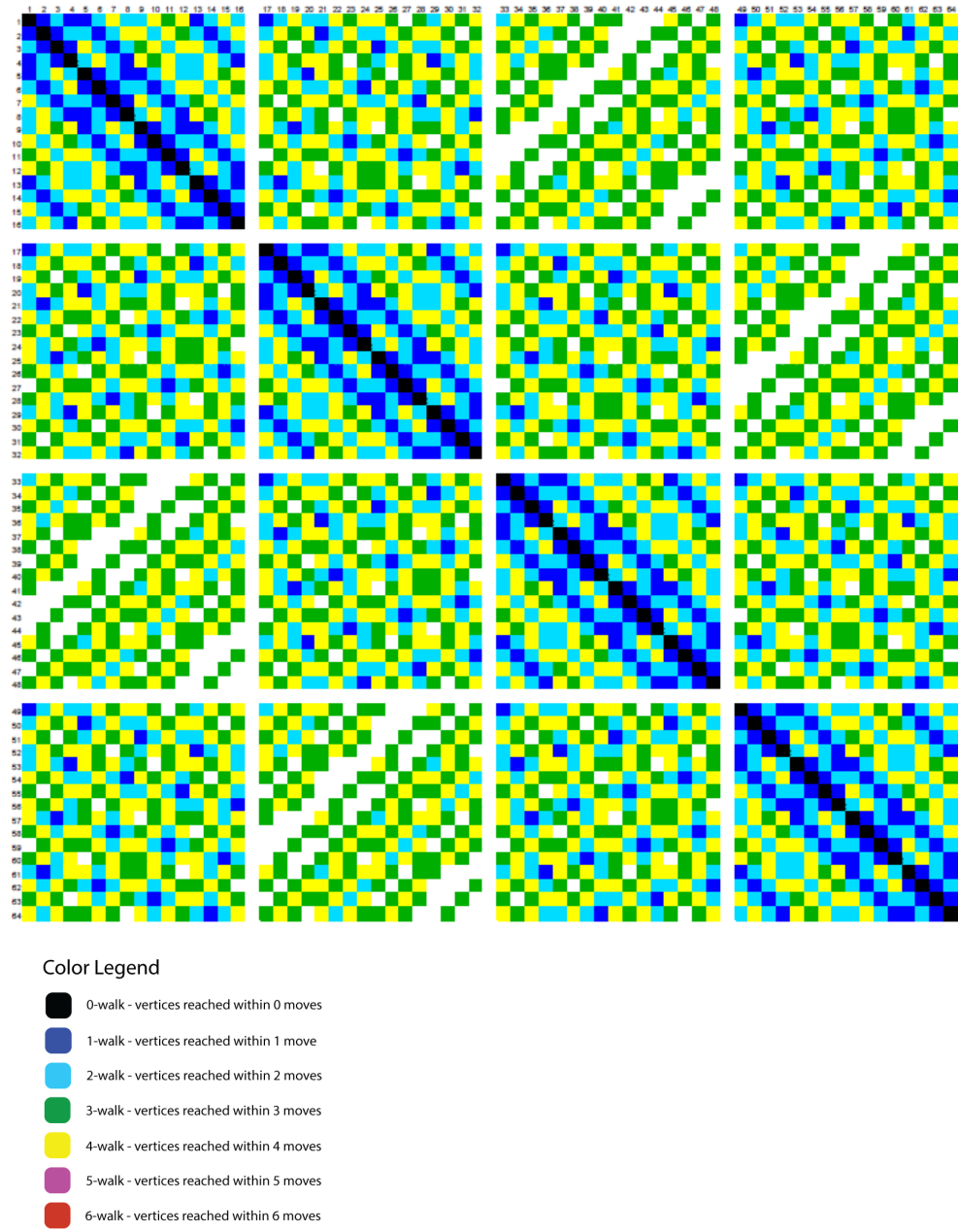


Figure 2.9: 4 - walk adjacency matrix organized by cosets of the subgroup $H = \langle R, U \rangle$

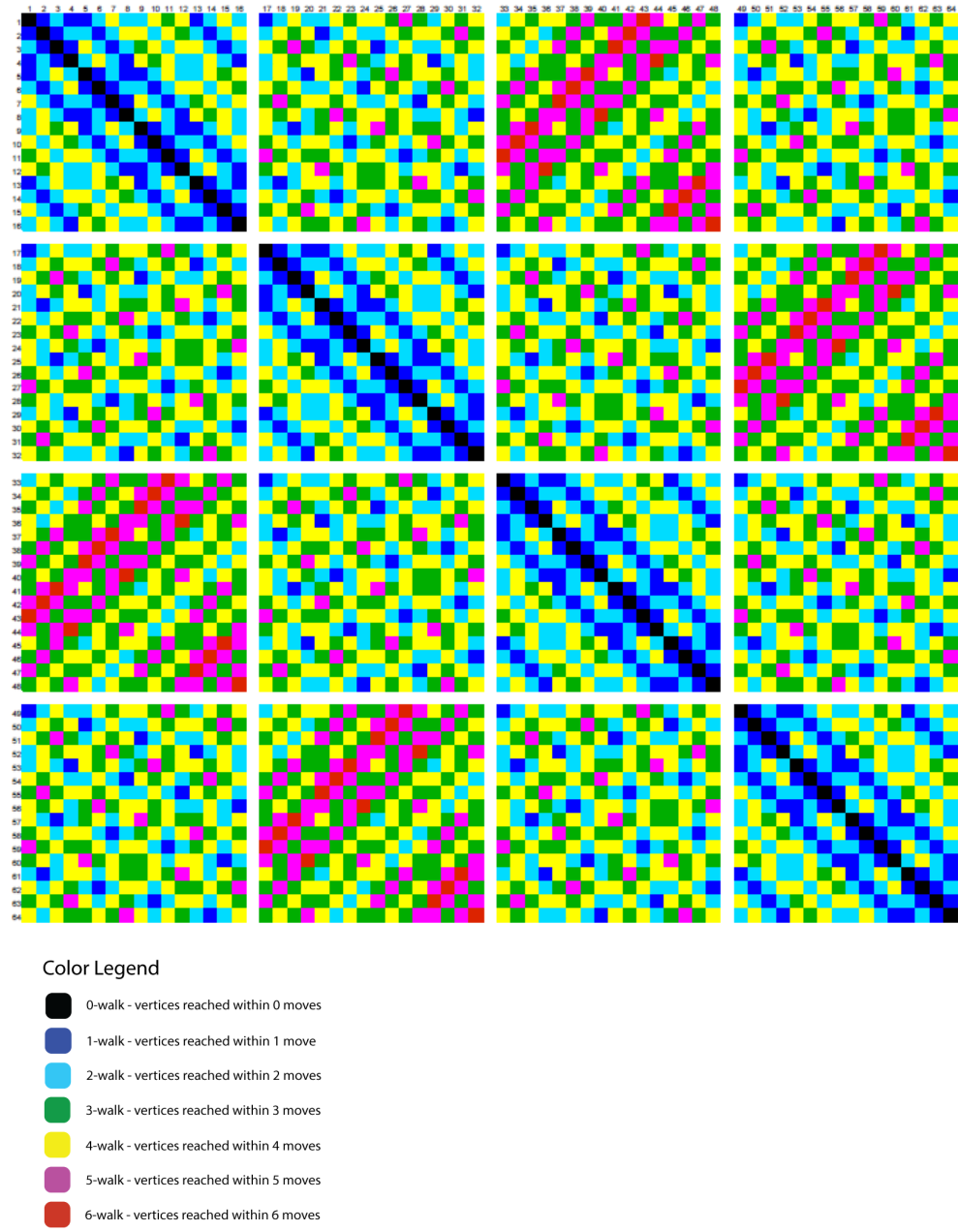


Figure 2.10: 6 – *walk* adjacency matrix organized by cosets of the subgroup $H = \langle R, U \rangle$ with red vertices being the most distant.

2.5 Subgames of the 4×4 Easy Mode

The overall analysis of the game so far has been related to a game played of 16 uniquely colored boxes. In reality the game, as produced by the Rubik's company, is only played with 1-3 colored boxes of up to 2 colors. The full group analysis provides an upper bound on our graphs diameter but it can be significantly smaller. A trivial subgame is the game involving one colored box. There exists only one subgame of this configuration and its diameter of 4 is obvious. Figure 2.11 displays a graph of this group.

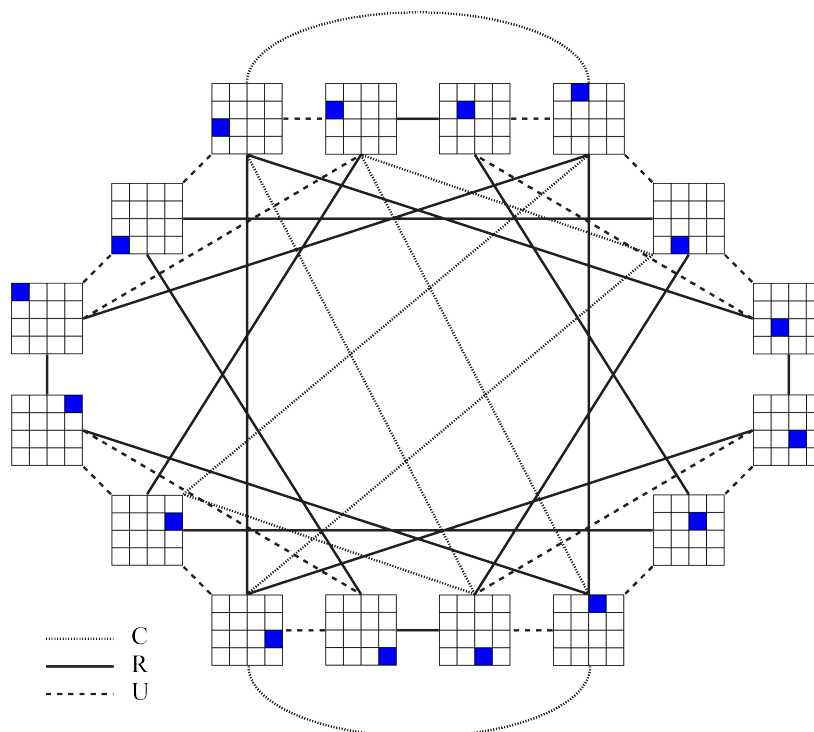


Figure 2.11: Subgame - 1 colored box

There are 5 single color 2 box games of the associated groups. Three have order 32, one has order 16, and the remaining one has order 8. All subgames are actually subgroups of the 4×4 easy mode group therefore the orders of each subgame will be less than or equal to 64. By Lagrange's Theorem, we know that the order of the subgroup will divide into the order of the group. There are also $\binom{16}{2} = 120$ possible ways to color a board of 16 boxes with a single color. No subgames of equal colorings will share the same configurations so the order of each subgame will sum up to $32 + 16 + 32 + 32 + 8 = 120$. To generate subgames start with an initial coloring configuration and then apply

the predefined transformations R , U , and C onto that configuration, continue re-applying the transformations onto the resulting configurations until all possible configurations have been realized. Figure 2.12 presents the Cayley graph of the subgame of single coloring with an initial coloring configuration of boxes 1 and 11 colored.

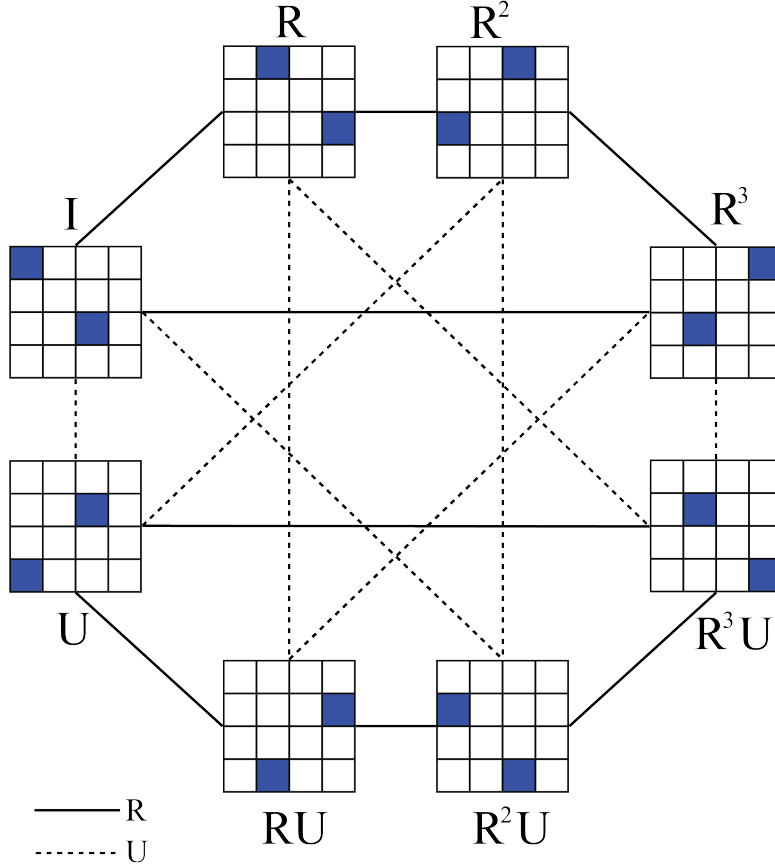


Figure 2.12: Subgame - 1 color 2 colored boxes

There are total of 11 three single colored subgames with a total of $\binom{16}{3} = 560$ possible colorings. Figure 2.13 summarizes the single colorings of 1-3 boxes along with the orders and God's number of each subgame. The subgames of a single coloring of 4 boxes has a total of $\binom{16}{4} = 1820$ possible colorings, Figure 2.15 presents one such subgame and its associated graph. The number of subgames grows significantly for colored boxes of four through eight and a summary of the information is presented in Figure 2.14.


















Image	Initial Coloring	Order	God's Number	Image	Initial Coloring	Order	God's Number
	[1]	16	4		[1,2,8]	64	6
	[1,2]	32	4		[1,2,9]	64	6
	[1,3]	16	3		[1,2,11]	64	6
	[1,6]	32	5		[1,3,6]	64	6
	[1,7]	32	4		[1,3,9]	16	4
	[1,11]	8	2		[1,3,10]	32	4
	[1,2,3]	32	4		[1,6,11]	32	4
	[1,2,5]	64	6		[1,6,12]	64	6
	[1,2,7]	64	6				

Figure 2.13: Summary of all single-colored games where 1-3 boxes are colored.

Single Color Group Order Distribution Frequency						
# of Colored Blocks	2	4	8	16	32	64
1				1		
2			1	1	3	
3				1	3	7
4		1	3	6	9	22
5				3	9	63
6			1	6	33	107
7				3	16	170
8	1	1	4	8	37	180

Figure 2.14: Single Color Group Order Distribution Frequency for 4×4 Easy Mode Rubik's Slide

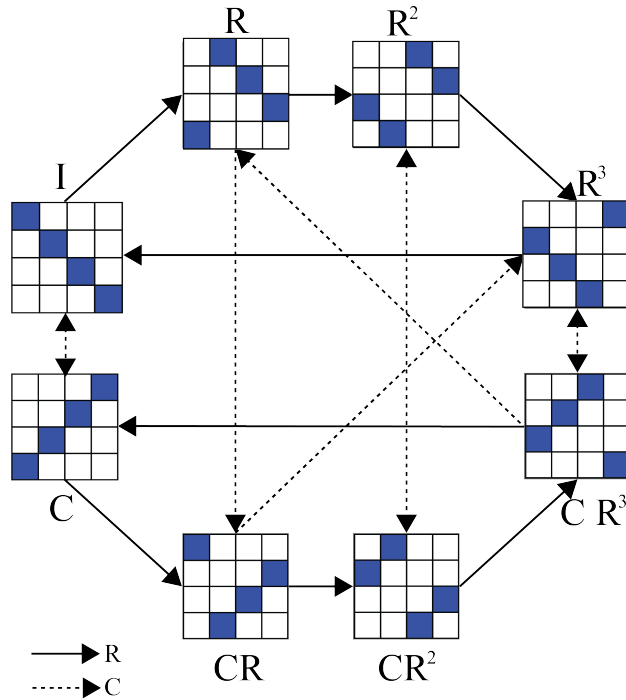


Figure 2.15: Graph of single colored subgame with an initial coloring of $[1,6,11,16]$

By adding a second color to our subgame the number of subgames continue

to grow. In Figure 2.12 a subgame of a single color of two colored boxes was presented, compare that to Figure 2.16 where the same initial configuration is used but this in this instance with two different colors and observe that the number of vertices doubles from 8 to 16 by the addition of a second color. This subgame is actually a hypercube and is represented in Figure 2.17.

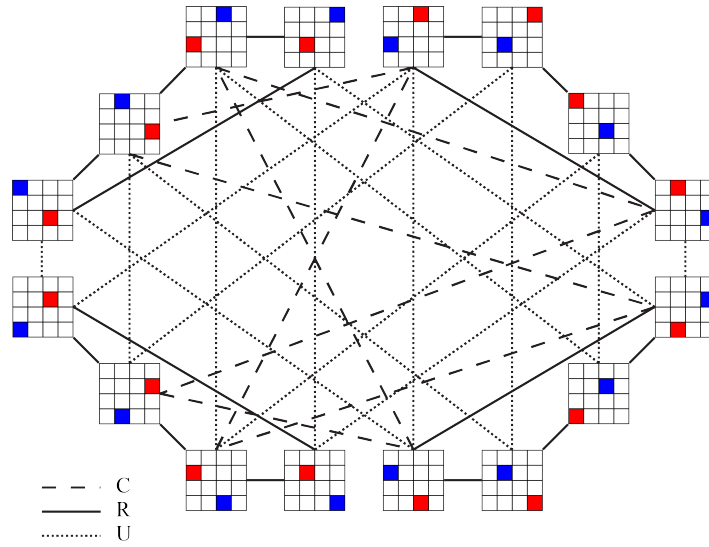


Figure 2.16: Subgame - 2 colors 2 colored boxes.

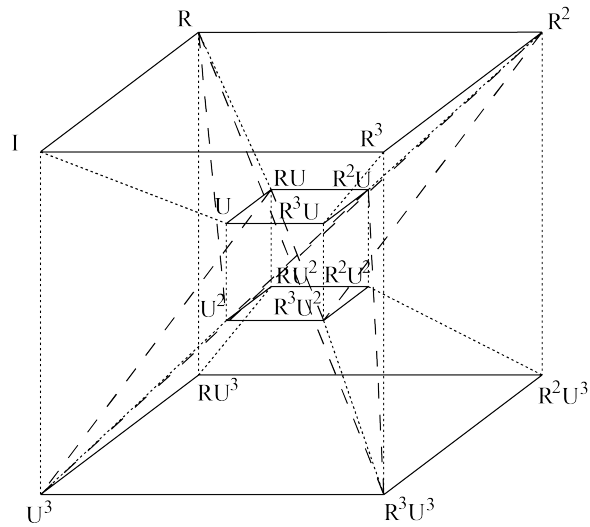


Figure 2.17: Subgame - 2 colors 2 colored boxes as a hypercube.

To conclude the investigation of the 4×4 easy mode, Figure 2.18 presents a chart summarizing the number of all two color subgames of up to eight colored boxes by their orders.

Two Color Group Order Distribution Frequency						
# of Colored Blocks	2	4	8	16	32	64
2				1	1	3
3				1	4	24
4				3	38	230
5				3	33	775
6				6	116	2986
7				3	94	5246
8			1	17	223	10196

Figure 2.18: Two Color Group Order Distribution Frequency for 4×4 Easy Mode Rubik's Slide

3 4×4 Hard Mode

3.1 Defining the Group

The 4×4 hard and easy modes share the same R and U transformations but the C rotation becomes a 30 degree clockwise rotation, Figure 3.1 displays the new rotation.

$$R = (1 \ 2 \ 3 \ 4)(5 \ 6 \ 7 \ 8)(9 \ 10 \ 11 \ 12)(13 \ 14 \ 15 \ 16)$$

$$U = (1 \ 13 \ 9 \ 5)(2 \ 14 \ 10 \ 6)(3 \ 15 \ 11 \ 7)(4 \ 16 \ 12 \ 8)$$

$$C = (1 \ 2 \ 3 \ 4 \ 8 \ 12 \ 16 \ 15 \ 14 \ 13 \ 9 \ 5)(6 \ 7 \ 11 \ 10)$$

The 4×4 hard mode inverses of the generating transformations are $R^{-1} = R^3$, $U^{-1} = U^3$, $C^{-1} = C^{12}$. This group, as with all of the other previous groups, is non-abelian.

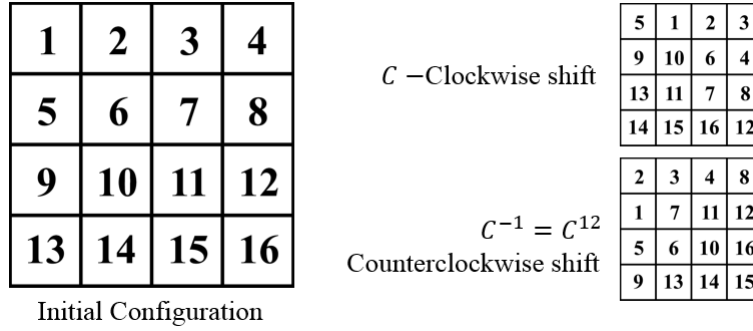


Figure 3.1: The 4×4 hard mode clockwise rotation, C .

3.2 On God's Number and Subgames

Limitations on computing have restricted advances in our insight on the God's number / diameter of the resulting graph from the 4×4 hard mode along with its group identification through GAP. The order of this group is $1,625,702,400 = 2^6 \cdot (7!)^2$. Even with these limitations I have developed a lower bounds on the diameter of the graph through a partial analysis on the subgames of the 4×4 hard mode. The lower bound for the diameter of the graph generated by $\langle C, R, U | R^4 = U^4 = C^{12} = I \rangle$ is 8. The single coloring of boxes 1,2,3,6 creates a subgame of order 896 with a diameter of 8.

A partial list of subgame initial configurations of a single color, order, and God's number is presented in Figure 3.2. All subgames where 1-5 boxes are colored are listed along with the orders and most of their God's numbers. The colorings of 3 subgames of 6-8 box colorings are acknowledged as existing but no order or God's number has been identified. Finally a partial list of subgame initial configurations of two colors of up to 5 boxes colored, order, and God's

number is presented in Figure 3.3. A subgame that may provide some interesting results in developing a sharper lower bound is the subgame of two colors where boxes 1 and 2 are colored blue followed by boxes 3,4,5 colored red. The order of this subgame is 18816 and the largest value encountered from this research of any subgames presented.



















Image	Initial Coloring	Order	God's Number	Image	Initial Coloring	Order	God's Number
	[1]	16	4		[1,2,3,6,8]	1120	
	[1,2]	64	5		[1,3,6,8,9]	112	6
	[1,3]	56	5		[1,2,3,4,5,6]		
	[1,2,3]	448	7		[1,2,3,4,5,6,7]		
	[1,3,6]	112	6		[1,2,3,4,5,6,7,8]		
	[1,2,3,4]	784	7		[1,2,3,4,5,6,7,8]		
	[1,2,3,6]	896	8		[1,2,3,4,5,6,7,8]		
	[1,3,6,8]	140	6		[1,2,3,4,5,6,7,8]		
	[1,2,3,4,5]	3136			[1,2,3,4,5,6,7,8]		

Figure 3.2: Partial list of single-colored games where 1-8 boxes are colored.

























Image	Initial Coloring	Order	God's Number	Image	Initial Coloring	Order	God's Number
	[1],[2]	128	6		[1,3],[6,8]	840	
	[1],[3]	112	6		[1],[2,3,4,5]	6272	
	[1],[2,3]	896	6		[1],[2,3,4,6]	9408	
	[1],[2,4]	448	7		[1],[2,3,6,8]	4480	
	[1],[3,6]	336	7		[1,2],[3,4,5]	18816	
	[1],[2,3,4]	3136			[1,2],[3,6,8]	4480	
	[1],[2,3,6]	2688			[1],[2,4,5,7]	1120	
	[1],[3,6,8]	560			[1],[3,6,8,9]	560	
	[1,2],[3,4]	3136			[1,3],[2,4,6]	9408	
	[1,2],[3,6]	2688			[1,3],[2,4,5]	3136	
	[1,3],[2,4]	1568			[1,3],[2,6,8]	6720	
	[1,3],[2,6]	2688			[1,3],[6,8,9]	1120	

Figure 3.3: Summary of all single-colored games where 1-3 boxes are colored.

3.3 Some 4×4 Configurations of Interest

Through the study of this group, a number of notable sequences of transformations for configurations of interest have been found, along with some that still remain elusive. Figure 3.4 displays examples of configurations of interest and the proceeding list of facts provides some insight to the game.

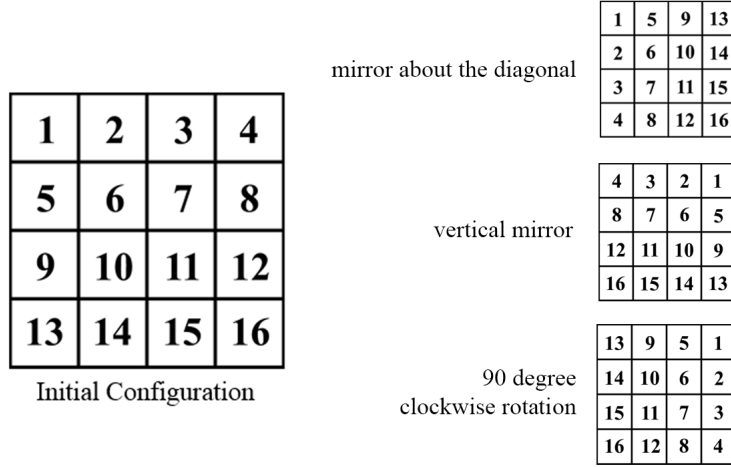


Figure 3.4: Three configurations of interest.

- A 180° clockwise rotation is C^6 .
- To mirror about the diagonal, perform the following sequence of transformations (parenthesis added for readability):

$$(C^4 R^3 U^3)(C^8 RU)(C^4 R^3 U^3)$$

- The sequence of transformations for the mirror about the secondary diagonal is a 180° rotation followed by a mirror about the diagonal or a mirror about the diagonal followed by a 180° rotation.

$$\begin{aligned} C^6 \cdot (C^4 R^3 U^3)(C^8 RU)(C^4 R^3 U^3) &= (C^{10} R^3 U^3)(C^8 RU)(C^4 R^3 U^3) \\ &= (C^4 R^3 U^3)(C^8 RU)(C^4 R^3 U^3) \cdot C^6 \end{aligned}$$

- GAP software identifies that the vertical mirror configuration is an element of the hard mode group. This sequence of transformations remains unknown at this time.
- A short sequence of transformations for horizontal mirroring and 90° and 270° rotations does not exist.
- The sequence of transformations for the horizontal mirroring is a 180° rotation followed by a vertical mirroring or a vertical mirroring followed by a 180° rotation.
- The sequence of transformations for the 90° clockwise rotation is a vertical mirroring followed by a mirroring about the secondary diagonal.
- The sequence of transformations for the 270° clockwise rotation is a vertical mirroring followed by a mirroring about the diagonal.

Part III

Summary

The Rubik's Slide is a planar game similar to the original 3-dimensional Rubik's Cube. The physical differences between the two games is that the original Rubik's Cube is a 3-dimensional object with transformations defined across faces of the cube and restricted by its physical movements while the Rubik's Slide is a 2-dimensional object restricted to three definable transformations across a single face. The Rubik's Slide available for purchase exists as a 2-dimensional 3×3 electronic board made up of nine blocks and was the subject of two recent studies, On God's Number(s) for Rubik's Slide [1] by Jones, Shelton and Weaverdyck and Rubik's on a Torus [2] by Alm, Gramelspacher, and Rice. The object of the game is to move about the board by using predefined transformations from an initial starting configuration to a pre-defined final configuration.

In this study, I have investigated an extension of the Rubik's Slide game on a board of dimensions 4×4 . There exists two modes of play, easy and hard, that are differentiated by the angle of rotation of the clockwise generating action on the elements of this group. A number of theoretical approaches are utilized to analyze the structure of the group generated by the extended game, to find the God's number or diameter of the graph, and to analysis the subgames or subgroups of this theoretical version. This investigation has yielded that the God's number in easy mode is 6. It turns out that the God's number for the 4×4 board is also the maximum value for any of the subgames that can be played on this board with those transformations. Unfortunately, the God's number in hard mode for the group of order 1,625,702,400 still eludes us. New mathematical tools will be required to calculate God's number of groups with larger orders. Increasing computational speeds and increasing the computing data size limitations help to provide insight however, due to calculation size, developing more theoretical strategies will be key to advancing the research. For example, whether we can extract local information about subgroups that provide global information about the entire group is a question that should be drawn out using theory. A list of known God's numbers for each general version of the game is provided in 3.5.

Board Dimensions	Group generated by R , U , and 90° clockwise rotation		Group generated by R , U , and 30° clockwise rotation	
	Order	God's number	Order	God's number
2×2	8	2	8	2
3×3	36	4	362,880	17
4×4	64	6	1,625,702,400	*
\vdots	\vdots	\vdots	\vdots	\vdots
$n \times n$	*	*	*	*

*unknown

Figure 3.5: Summary of known order and God's numbers

Part IV

Works Cited

1. Alm, J., Gramelpacher, M., & Rice, T. (2013). *Rubik's on the torus*. The American Mathematical Monthly, 120(2), 150-160. Retrieved , from <http://www.jstor.org/stable/10.4169/amer.math.monthly.120.02.150>
2. Jones, M.A., Shelton, B., & Weaverdyck, M. (2014, Sept.) *On God's Number(s) for Rubik's Slide*. The College of Mathematics Journal, Vol. 45(04), 267-275.
3. Bollobas, B. (1998) *Modern graph theory*. New York: Springer.
4. Fraleigh, J., & Katz, V. (2003) *A first course in abstract algebra* (7th ed.). Boston: Addison-Wesley.

Resources

- All source code, adjacency tables, game software, and related files are stored at <http://www.jimmyjohnston.org/research/rubikslide>.
- Screen captured simulations of the current software is available for preview at <http://www.youtube.com/watch?v=GvMrn88DgEE>.

Part V

Appendix A - Python Code

The code provided below is used to calculate paths to vertices, whether two vertices are adjacent, the God's number of games, the number of games and the orders of those games. It also creates a space where you can perform permutation operations using sympy. Copy the code to a text editor and save as .py. Run code using a Python interpreter.

```
import math, pickle, random, time, numpy
from sympy.combinatorics import *
from sympy.matrices import *
from sympy.combinatorics.perm_groups import
    PermutationGroup
from sympy.combinatorics.named_groups import *
from operator import itemgetter

#DESCRIPTION RS4x4 Explorer
#AUTHOR Jim Johnston
#EMAIL jimmy.johnston@gmail.com
#LAST MODIFIED 2016-04-26
#LICENSING GNU GPLv3
#DEPENDENCY version >= Python 3.0, sympy needs to be
    installed
#NOTES The functions below are designed for the analysis
    of the Rubik's Slide 4x4 Easy and Hard modes. Both
    modes are algebraic groups where the easy mode is
    generated by the elements <R,U,C90> and the hard mode
    by <R,U,C>. Examples are provided at the bottom of
    the file on usage. Many functions are functions
    dependent on others so starting with the examples is
    recommended.

#game rules defined using sympy permutations
I = Permutation(16)
C90 = Permutation(1,4,16,13)(2,8,15,9)(3,12,14,5)
    (6,7,11,10)
C = Permutation(1,2,3,4,8,12,16,15,14,13,9,5)(6,7,11,10)
R = Permutation(1,2,3,4)(5,6,7,8)(9,10,11,12)
    (13,14,15,16)
U = Permutation(1,13,9,5)(2,14,10,6)(3,15,11,7)
    (4,16,12,8)

T = Permutation(1,2)
```

```

def shift(e,v):
    #sfinds index of element e in list v returns next
    element
    return(v[(v.index(e)+1)%4])

def shiftinv(e,v):
    #finds index of element e in list v returns previous
    element
    return(v[(v.index(e)-1)%4])

def right(e):
    #right shift using lists
    v1 = [1, 2, 3, 4]
    v2 = [5, 6, 7, 8]
    v3 = [9, 10, 11, 12]
    v4 = [13, 14, 15, 16]
    new_e = [] #resulting configuration return list
    for i in e:
        if i in v1:
            new_e.append(shift(i,v1))
        elif i in v2:
            new_e.append(shift(i,v2))
        elif i in v3:
            new_e.append(shift(i,v3))
        elif i in v4:
            new_e.append(shift(i,v4))
    return(new_e)

def up(e):
    #up shift using lists
    v1 = [1, 13, 9, 5]
    v2 = [2, 14, 10, 6]
    v3 = [3, 15, 11, 7]
    v4 = [4, 16, 12, 8]
    new_e = []
    for i in e:
        if i in v1:
            new_e.append(shift(i,v1))
        elif i in v2:
            new_e.append(shift(i,v2))
        elif i in v3:
            new_e.append(shift(i,v3))
        elif i in v4:
            new_e.append(shift(i,v4))
    return(new_e)

```

```

def c_easy(e):
    #clockwise 90 degree rotation using lists
    v1 = [1, 4, 16, 13]
    v2 = [2, 8, 15, 9]
    v3 = [3, 12, 14, 5]
    v4 = [6, 7, 11, 10]
    #to handle multiple elements run through a loop that
    computes each piece individually
    new_e = []
    for i in e:
        if i in v1:
            new_e.append(shift(i,v1))
        elif i in v2:
            new_e.append(shift(i,v2))
        elif i in v3:
            new_e.append(shift(i,v3))
        elif i in v4:
            new_e.append(shift(i,v4))
    return(new_e)

def c(e):
    #clockwise rotation using lists
    v1 = [1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5]
    v2 = [6, 7, 11, 10]
    #to handle multiple elements run through a loop that
    computes each piece individually
    new_e = []
    for i in e:
        if i in v1:
            new_e.append(v1[(v1.index(i)+1)%12])
        elif i in v2:
            new_e.append(shift(i,v2))
    return(new_e)

def rightinv(e):
    #right inverse (left) shift using lists
    v1 = [1, 2, 3, 4]
    v2 = [5, 6, 7, 8]
    v3 = [9, 10, 11, 12]
    v4 = [13, 14, 15, 16]
    new_e = []
    for i in e:
        if i in v1:
            new_e.append(shiftinv(i,v1))
        elif i in v2:

```

```

        new_e.append(shiftinv(i,v2))
    elif i in v3:
        new_e.append(shiftinv(i,v3))
    elif i in v4:
        new_e.append(shiftinv(i,v4))
    return(new_e)

def upinv(e):
    #up inverse (down) shift using lists
    v1 = [1, 13, 9, 5]
    v2 = [2, 14, 10, 6]
    v3 = [3, 15, 11, 7]
    v4 = [4, 16, 12, 8]
    new_e = []
    for i in e:
        if i in v1:
            new_e.append(shiftinv(i,v1))
        elif i in v2:
            new_e.append(shiftinv(i,v2))
        elif i in v3:
            new_e.append(shiftinv(i,v3))
        elif i in v4:
            new_e.append(shiftinv(i,v4))
    return(new_e)

def c_easyinv(e):
    #clockwise 90 degree inverse (counterclockwise)
    rotation using lists
    v1 = [1, 4, 16, 13]
    v2 = [2, 8, 15, 9]
    v3 = [3, 12, 14, 5]
    v4 = [6, 7, 11, 10]
    #to handle multiple elements run through a loop that
    computes each piece individually
    new_e = []
    for i in e:
        if i in v1:
            new_e.append(shiftinv(i,v1))
        elif i in v2:
            new_e.append(shiftinv(i,v2))
        elif i in v3:
            new_e.append(shiftinv(i,v3))
        elif i in v4:
            new_e.append(shiftinv(i,v4))
    return(new_e)

```

```

def c_inv(e):
    #clockwise inverse (counterclockwise) rotation using lists
    v1 = [1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5]
    v2 = [6, 7, 11, 10]
    #to handle multiple elements run through a loop that computes each piece individually
    new_e = []
    for i in e:
        if i in v1:
            new_e.append(v1[(v1.index(i)-1)%12])
        elif i in v2:
            new_e.append(shiftinv(i,v2))
    return(new_e)

def one_color_game(mixed_list):
    #Determines the number of blocks being played for sorting purposes
    if len(mixed_list[0])==1:
        gamekey = itemgetter(0)
    elif len(mixed_list[0])==2:
        gamekey = itemgetter(0,1)
    elif len(mixed_list[0])==3:
        gamekey = itemgetter(0,1,2)
    elif len(mixed_list[0])==4:
        gamekey = itemgetter(0,1,2,3)
    elif len(mixed_list[0])==5:
        gamekey = itemgetter(0,1,2,3,4)
    elif len(mixed_list[0])==6:
        gamekey = itemgetter(0,1,2,3,4,5)
    elif len(mixed_list[0])==7:
        gamekey = itemgetter(0,1,2,3,4,5,6)
    elif len(mixed_list[0])==8:
        gamekey = itemgetter(0,1,2,3,4,5,6,7)
    #first for loop sorts each individual position, ex. [1,3,2] >> [1,2,3]
    #second for loop puts only unique positions into list and then returns sorted list
    #reasons behind second loop is that with single color [1,3,2] and [1,2,3] are the same.
    new_mixed_list = []
    for element in mixed_list:
        element.sort()
    for element in mixed_list:
        if element not in new_mixed_list:
            new_mixed_list.append(element)

```

```

    return(sorted(new_mixed_list, key=gamekey))

def sort_element_list(element_list):
    #used by permutations2 function
    sortedlist = []
    for element in element_list:
        newelement = []
        for subelement in element:
            newelement.append(sorted(subelement))
        if newelement not in sortedlist:
            sortedlist.append(newelement)
    return(sortedlist)

def permutations(element, moves):
    #purpose: used to find all permutations of a given
    initial single color configuration
    #element is a list of colored boxes, moves are a list
    of transformations defined by lists
    list_of_elements_in_subgame = [] #will contain all
    elements in specific subgame
    current_element_location = [element] #once empty
    implies coset has been completed
    while(current_element_location): #while there are
    elements in set
        element = current_element_location[-1] #use the last
        element in set
        current_element_location.pop() #remove element being
        used from set
        if element not in list_of_elements_in_subgame: #if
        element not in result list then do
            list_of_elements_in_subgame.append(element) #add
            element to resulting list
        for move in moves:
            new_element = move(element) #track each next
            position given all moves
            if new_element not in current_element_location:
                if new_element not in
                    list_of_elements_in_subgame:
                        current_element_location.append(new_element)
                        #as long as this element exists no where
                        in either set then it is a new element to
                        explore next steps
    return(one_color_game(list_of_elements_in_subgame))

```

```

def permutations2(element, moves):
    #purpose: generates all configurations of a single
    multi-colored game given an initial configuration (
    list) and transformations
    list_of_elements_in_coset = [] #will contain all
    elements in specific coset
    current_element_location = [element] #once empty
    implies coset has been completed
    colorcount = len(element)
    while(current_element_location): #while there are
    elements in set
        element = current_element_location[-1] #use the last
        element in set
        current_element_location.pop() #remove element being
        used from set
        if element not in list_of_elements_in_coset: #if
        element not in result list then do
            list_of_elements_in_coset.append(element) #add
            element to resulting list
            for move in moves:
                new_element = []
                for i in range(colorcount): #critical piece to
                handle multi-color positions
                    new_element.append(move(element[i]))
                #new_element = [move(element[0]), move(element
                [1])] #track each next position given all
                moves
                if new_element not in current_element_location:
                    if new_element not in
                        list_of_elements_in_coset:
                            current_element_location.append(new_element)
                            #as long as this element exists no where
                            in either set then it is a new element to
                            explore next steps
            return(sort_element_list(list_of_elements_in_coset))

def isAdjacent(i, j, moves):
    #purpose: determine boolean truth value of whether two
    elements are adjacent, support multi-color
    configurations
    #method: take element i, calculate its neighbors by
    applying transformations to it after each
    transformation check to see if element j is in
    result if it is no need to continue performing
    transformations
    #assumptions: it is assumed that both i and j are

```



```

        written as a sorted list
    for move in moves:
        new_element = []
        for sub in i:
            new_element.append(sorted(move(sub)))
        if j==new_element:
            return(True)
    return(False)

def adjacencyMatrix(elementlist,moves):
    #purpose: given a list of elements and transformations
    on them return an adjacency matrix in the form of
    a sympy matrix object
    build_list_for_matrix = []
    for elementI in elementlist:
        for elementJ in elementlist:
            if isAdjacent(elementI,elementJ,moves):
                build_list_for_matrix.append(1)
                #print(elementI,elementJ,1)
            else:
                build_list_for_matrix.append(0)
                #print(elementI,elementJ,0)
    M = Matrix(len(elementlist),len(elementlist),
               build_list_for_matrix)
    return(M)

def godnumber(configurations,moves,fname="default-
adjacency-matrix",limit=10,verbose=False):
    #purpose: calculate the god's number / diameter of a
    graph
    #vars: configurations is the game elements written as
    a list, MOVES are a list of generating permutations
    ,
    #vars: FNAME is a string filename, VERSION 1 or 2
    represents how many colors are in game, LIMIT is a
    computational boundary
    #limitations: currently works for only one or two
    colored games
    #last updated: 2015-12-05
    print("Computing_God's_Number_of_"+str(configurations
[0]))
    I = eye(len(configurations)) #creates the identity
    matrix
    if all(isinstance(X,list) for X in configurations
[0]):
        #if version==2: #creates adjacency matrix

```

```

    A = adjacencyMatrix2(configurations,moves) #multiple
    color games
else:
    A = adjacencyMatrix(configurations,moves) #single
    color games
power = 1
result = I #0-walk
while (power<limit): #setting a computational boundary
    result+=A**power #n-walk or more appropriate power-
    walk haha
    if power==1: #this writes you initial adjacency
    matrix to disk
        f = open(fname+str(power)+".txt", "w")
        f.write(str(A))
        f.close()
    if 0 not in result: #God's number has been
    calculated, write resulting n-walk to disk
        f = open(fname+str(power)+".txt", "w")
        f.write(str(result))
        f.close()
        print("God's_Number_is_"+str(power)) if verbose
        else None
        return(power)
    print("nothing_yet,_just_checked_"+str(power)) if
    verbose else None
    power+=1
print("we_have_reached_the_computational_limit_of_"+
    str(limit)+"set_by_you") if verbose else None
return(-1)

def gameOrder(initialconfig,generators,verbose=False):
    #purpose: calculates the order of the coset given an
    initial configuration and generators
    #vars: initialconfig must be a list such as [1,2,3]
    where the blocks 1,2,3 are colored a single color
    #vars: or a list of lists [[1,2],[3]] where blocks 1,2
    are colored a single color, and block 3 is colored
    another color
    #vars: generators is a list of permutations defined as
    the functions easymoves =[right,up,c_easy,rightinv
    ,upinv,c_easyinv]
    #vars: or hardmoves = [right,up,c,rightinv,upinv,c_inv
    ], you can choose other generators by mixing and
    matching the predefined transformations or you can
    write your own function to define a set of
    permutations, the one requirement is that the

```

```

        permutations are defined as lists , not as a sympy
        permutation
    print (" Calculating_order_of_game_with_initial_
            configuration_of_"+str(initialconfig))
    if all(isinstance(X,list) for X in initialconfig):
        order = len(permutations2(initialconfig,generators))
        print ("Game_order_of_"+str(initialconfig)+"_is_"+
                str(order)) if verbose else None
        return(order)
    elif any(isinstance(X,list) for X in initialconfig)
        :
        return("Your_initial_configuration_is_not_properly_
                formatted.\nMust_be_a_list_of_integers_or_a_list_
                of_lists_of_integers.")
    else:
        order = len(permutations(initialconfig,generators))
        print ("Game_order_of_"+str(initialconfig)+"_is_"+
                str(order)) if verbose else None
        return(order)

def searchpath(elements , rotation=C):
    #This function prints a sequence of transformations of
    the form (C^a * R^b * U^c)(C^d * R^e * U^f)(C^g * R^
    h * U^i)(C^j * R^k * U^m) given a generating set of
    transformations and a LIST of elements you are
    searching for.
    #Default setting is hard mode pass C90 as second
    variable when calling to set as easy mode.
    track = 0
    for a in range(0,12):
        for b in range(0,4):
            for c in range(0,4):
                for d in range(0,12):
                    for e in range(0,4):
                        for f in range(0,4):
                            for g in range(0,12):
                                for h in range(0,4):
                                    for i in range(0,4):
                                        for j in range(0,12):
                                            for k in range(0,4):
                                                for m in range(0,4):
                                                    contender = (rotation**a)*(R**b
                                                                )*(U**c)*(rotation**d)*(R**e
                                                                )*(U**f)*(rotation**g)*(R**h
                                                                )*(U**i)*(rotation**j)*(R**k
                                                                )*(U**m)

```

```

        if contender in elements:
            print(contender, ">>>", a, b, c
                  , d, e, f, g, h, i, j, k, m)
            elements.remove(contender)
            if len(elements)==0:
                return(1, track)
            track+=1

    return(0, track)

#####
#This next function gamelist is lengthy there are no
#functions that follow.
#Immediately following gamelist is the examples section
#####

def gamelist(n, moves, verbose=False):
    #purpose: generates all unique subgames of a defined
    #game
    #vars: moves the permutations that define the group
    #along with their inverses
    #limitations: currently works for one colored games
    #only
    #notes: this function is necessarily long, instead of
    #condensing the code it is purposely left long for
    #readability and understanding
    games = []
    if n==2:
        #2 blocks
        print("Analyzing_2_colored_blocks_games")
        for i in range(2,17):
            initialconfig = [1,i]
            game_exists = False
            for game in games:
                if initialconfig in game:
                    game_exists = True
                    break
            if not game_exists:
                contender = permutations(initialconfig, moves)
                contender.sort()
                games.append(contender)
                print(str(contender[0])+str(len(contender)))
                if verbose else None
        print("completed")
    elif n==3:
        #3 blocks
        print("Analyzing_3_colored_blocks_games")

```

```

for i in range(2,17):
    for j in range(i+1,17):
        initialconfig = [1,i,j]
        game_exists = False
        for game in games:
            if initialconfig in game:
                game_exists = True
                break
        if not game_exists:
            contender = permutations(initialconfig,moves)
            contender.sort()
            games.append(contender)
            print(str(contender[0])+str(len(contender))
                ) if verbose else None
        print("completed")
elif n==4:
    #4 blocks
    print("Analyzing_4_colored_blocks_games")
    for i in range(2,17):
        for j in range(i+1,17):
            for k in range(j+1,17):
                initialconfig = [1,i,j,k]
                game_exists = False
                for game in games:
                    if initialconfig in game:
                        game_exists = True
                        break
                if not game_exists:
                    contender = permutations(initialconfig,moves)
                    contender.sort()
                    games.append(contender)
                    print(str(contender[0])+str(len(contender))
                        ) if verbose else None
                print("completed")
elif n==5:
    #5 blocks
    print("Analyzing_5_colored_blocks_games")
    for i in range(2,17):
        for j in range(i+1,17):
            for k in range(j+1,17):
                for m in range(k+1,17):
                    if i!=j and j!=k and k!=m and m!=i and m!=j
                        and k!=i:
                        initialconfig = [1,i,j,k,m]
                        game_exists = False
                        for game in games:

```

```

        if initialconfig in game:
            game_exists = True
            break
    if not game_exists:
        contender = permutations(initialconfig,
                                moves)
        contender.sort()
        games.append(contender)
        print(str(contender[0])+str(len(
            contender))) if verbose else None
    print("completed")
elif n==6:
    #6 blocks
    print("Analyzing_6_colored_blocks_games")
    for i in range(2,17):
        for j in range(i+1,17):
            for k in range(j+1,17):
                for m in range(k+1,17):
                    for n in range(m+1,17):
                        initialconfig = [1,i,j,k,m,n]
                        game_exists = False
                        for game in games:
                            if initialconfig in game:
                                game_exists = True
                                break
                        if not game_exists:
                            contender = permutations(initialconfig,
                                                    moves)
                            contender.sort()
                            games.append(contender)
                            print(str(contender[0])+str(len(
                                contender))) if verbose else None
    print("completed")
elif n==7:
    #7 blocks
    print("Analyzing_7_colored_blocks_games")
    for i in range(2,17):
        for j in range(i+1,17):
            for k in range(j+1,17):
                for m in range(k+1,17):
                    for n in range(m+1,17):
                        for p in range(n+1,17):
                            initialconfig = [1,i,j,k,m,n,p]
                            game_exists = False
                            for game in games:
                                if initialconfig in game:

```

```

        game_exists = True
        break
    if not game_exists:
        contender = permutations(initialconfig,
            moves)
        contender.sort()
        games.append(contender)
        print(str(contender[0])+str(len(
            contender))) if verbose else None
    print("completed")
elif n==8:
    #8 blocks
    print("Analyzing_8_colored_blocks_games")
    for i in range(2,17):
        for j in range(i+1,17):
            for k in range(j+1,17):
                for m in range(k+1,17):
                    for n in range(m+1,17):
                        for p in range(n+1,17):
                            for q in range(p+1,17):
                                initialconfig = [1,i,j,k,m,n,p,q]
                                game_exists = False
                                for game in games:
                                    if initialconfig in game:
                                        game_exists = True
                                        break
                                if not game_exists:
                                    contender = permutations(
                                        initialconfig,moves)
                                    contender.sort()
                                    games.append(contender)
                                    print(str(contender[0])+str(len(
                                        contender))) if verbose else None
    print("completed")
return(games)

def gamelist2(n,moves,verbose=False):
    #purpose: generates all unique subgames of a defined
    game
    #vars: moves the permutations that define the group
    along with their inverses
    #limitations: currently works for one colored games
    only
    #last updated: 2015-12-09
    #notes: this function is necessarily long, instead of
    condensing the code it is purposely left long for

```

```

        readability and understanding
games = []
if n==2:
    #2 blocks
    print ("Analyzing_2_two-colored_blocks_games")
    for i in range(2,17):
        contender = permutations2 ([[1]],[i] ,moves)
        contender.sort()
        if contender not in games:
            games.append(contender)
            print (str(contender[0])+str(len(contender))) if
                verbose else None
    print ("completed")
elif n==3:
    print ("Analyzing_3_two-colored_blocks_games")
    for i in range(1,2):
        for j in range(i+1,17):
            for k in range(j+1,17):
                initialconfig = [[i],[j,k]]
                game_exists = False
                for game in games:
                    if initialconfig in game:
                        game_exists = True
                        break
                if not game_exists:
                    contender = permutations2(initialconfig ,moves
                    )
                    contender.sort()
                    games.append(contender)
                    print (str(contender[0])+str(len(contender))
                    ) if verbose else None
    print ("completed")
elif n==4:
    print ("Analyzing_4_two-colored_blocks_games")
    for i in range(1,2):
        for j in range(i+1,17):
            for k in range(j+1,17):
                for m in range(k+1,17):
                    initialconfig1 = [[i],[j,k,m]]
                    initialconfig2 = [[i,j],[k,m]]
                    game1_exists = False
                    game2_exists = False
                    for game in games:
                        if initialconfig1 in game:
                            game1_exists = True
                            break

```



```

    for game in games:
        if initialconfig2 in game:
            game2_exists = True
            break
    if not game1_exists:
        contender = permutations2(initialconfig1 ,
            moves)
        contender.sort()
        games.append(contender)
        print(str(contender[0])+str(len(contender
            ))) if verbose else None
    if not game2_exists:
        contender = permutations2(initialconfig2 ,
            moves)
        contender.sort()
        games.append(contender)
        print(str(contender[0])+str(len(contender
            ))) if verbose else None
    print("completed")
elif n==5:
    print("Analyzing_5_two-colored_blocks_games")
    for i in range(1,2):
        for j in range(i+1,17):
            for k in range(j+1,17):
                for m in range(k+1,17):
                    for n in range(m+1,17):
                        initialconfig1 = [[i],[j,k,m,n]]
                        initialconfig2 = [[i,j],[k,m,n]]
                        game1_exists = False
                        game2_exists = False
                        for game in games:
                            if initialconfig1 in game:
                                game1_exists = True
                                break
                        for game in games:
                            if initialconfig2 in game:
                                game2_exists = True
                                break
                        if not game1_exists:
                            contender = permutations2(initialconfig1 ,
                                moves)
                            contender.sort()
                            games.append(contender)
                            print(str(contender[0])+str(len(
                                contender))) if verbose else None
                        if not game2_exists:

```

```

        contender = permutations2(initialconfig2,
                                   moves)
        contender.sort()
        games.append(contender)
        print(str(contender[0])+str(len(
            contender))) if verbose else None
    print("completed")
elif n==6:
    print("Analyzing_6_two-colored_blocks_games")
    for i in range(2,17):
        for j in range(i+1,17):
            for k in range(j+1,17):
                for m in range(k+1,17):
                    for n in range(m+1,17):
                        initialconfig1 = [[1],[i,j,k,m,n]]
                        initialconfig2 = [[1,i],[j,k,m,n]]
                        initialconfig3 = [[1,i,j],[k,m,n]]
                        game1_exists = False
                        game2_exists = False
                        game3_exists = False
                        for game in games:
                            if initialconfig1 in game:
                                game1_exists = True
                                break
                        for game in games:
                            if initialconfig2 in game:
                                game2_exists = True
                                break
                        for game in games:
                            if initialconfig3 in game:
                                game3_exists = True
                                break
                        if not game1_exists:
                            contender = permutations2(initialconfig1,
                                                       moves)
                            contender.sort()
                            games.append(contender)
                            print(str(contender[0])+str(len(
                                contender))) if verbose else None
                        if not game2_exists:
                            contender = permutations2(initialconfig2,
                                                       moves)
                            contender.sort()
                            games.append(contender)
                            print(str(contender[0])+str(len(
                                contender))) if verbose else None

```

```

        if not game3_exists:
            contender = permutations2(initialconfig3,
                                      moves)
            contender.sort()
            games.append(contender)
            print(str(contender[0])+str(len(
                contender))) if verbose else None
    print("completed")
elif n==7:
    print("Analyzing_7_two-colored_blocks_games")
    for i in range(2,17):
        for j in range(i+1,17):
            for k in range(j+1,17):
                for m in range(k+1,17):
                    for n in range(m+1,17):
                        for p in range(n+1,17):
                            initialconfig1 = [[1],[i,j,k,m,n,p]]
                            initialconfig2 = [[1,i],[j,k,m,n,p]]
                            initialconfig3 = [[1,i,j],[k,m,n,p]]
                            game1_exists = False
                            game2_exists = False
                            game3_exists = False
                            for game in games:
                                if initialconfig1 in game:
                                    game1_exists = True
                                    break
                            for game in games:
                                if initialconfig2 in game:
                                    game2_exists = True
                                    break
                            for game in games:
                                if initialconfig3 in game:
                                    game3_exists = True
                                    break
                            if not game1_exists:
                                contender = permutations2(
                                    initialconfig1,moves)
                                contender.sort()
                                games.append(contender)
                                print(str(contender[0])+str(len(
                                    contender))) if verbose else None
                            if not game2_exists:
                                contender = permutations2(
                                    initialconfig2,moves)
                                contender.sort()
                                games.append(contender)

```

```

        print(str(contender[0])+str(len(
            contender))) if verbose else None
    if not game3_exists:
        contender = permutations2(
            initialconfig3 ,moves)
        contender.sort()
        games.append(contender)
        print(str(contender[0])+str(len(
            contender))) if verbose else None
    print("completed")
elif n==8:
    print("Analyzing_8_two-colored_blocks_games")
    for i in range(2,17):
        for j in range(i+1,17):
            for k in range(j+1,17):
                for m in range(k+1,17):
                    for n in range(m+1,17):
                        for p in range(n+1,17):
                            for q in range(p+1,17):
                                initialconfig1 = [[1],[i,j,k,m,n,p,q]]
                                initialconfig2 = [[1,i],[j,k,m,n,p,q]]
                                initialconfig3 = [[1,i,j],[k,m,n,p,q]]
                                initialconfig4 = [[1,i,j,k],[m,n,p,q]]
                                game1_exists = False
                                game2_exists = False
                                game3_exists = False
                                game4_exists = False
                                for game in games:
                                    if initialconfig1 in game:
                                        game1_exists = True
                                        break
                                for game in games:
                                    if initialconfig2 in game:
                                        game2_exists = True
                                        break
                                for game in games:
                                    if initialconfig3 in game:
                                        game3_exists = True
                                        break
                                for game in games:
                                    if initialconfig4 in game:
                                        game4_exists = True
                                        break
                                if not game1_exists:
                                    contender = permutations2(
                                        initialconfig1 ,moves)

```

```

        contender.sort()
        games.append(contender)
        print(str(contender[0])+str(len(
            contender))) if verbose else None
    if not game2_exists:
        contender = permutations2(
            initialconfig2 ,moves)
        contender.sort()
        games.append(contender)
        print(str(contender[0])+str(len(
            contender))) if verbose else None
    if not game3_exists:
        contender = permutations2(
            initialconfig3 ,moves)
        contender.sort()
        games.append(contender)
        print(str(contender[0])+str(len(
            contender))) if verbose else None
    if not game4_exists:
        contender = permutations2(
            initialconfig4 ,moves)
        contender.sort()
        games.append(contender)
        print(str(contender[0])+str(len(
            contender))) if verbose else None

    print("completed")
    #nothing larger necessary
    return(games)

#
#####

##### Welcome to the Workspace/Example Area
#####
#
#####

## Examples of functions above are presented below
## Recommended you only use one function at a time
## Recommended you read each description before
operating as some functions require significant
resources
## To run examples below uncomment the section , save
file , and pass to Python interpreter
#
#####

```

```

#
#####

#for ease of use, pre-defined game rules, leave these
  uncommented
easymoves = [right,up,c_easy,rightinv,upinv,c_easyinv]
hardmoves = [right,up,c,rightinv,upinv,c_inv]

####PERMUTATIONS
##LAST MODIFIED: 2014-02-21
##CONDITION: Good
##DESC: Returns all of the elements in a group given an
        initial element and generators
#print(permutations([1,2,3],easymoves)) #designed for
        only single colored games
#print(permutations2([[1,3]],easymoves)) #designed for
        any number of colors
#print(permutations2
        ([[1],[2],[3],[4],[5],[6],[7],[8],[9],[10],[11],[12],[13],[14],[15],[16]],
        easymoves)) #this is the entire 4x4 easy group

####GAMEORDER
##LAST MODIFIED: 2015-12-09
##CONDITION: Good, fast
##DESC: For use on any colorings, identifies the order
        of the subgame generated by the moves and initial
        config passed to it.
#print(gameOrder([1,2],hardmoves))
#print(gameOrder([[1],[2]],hardmoves,True))

####GAMELIST
##LAST MODIFIED: 2015-12-05
##CONDITION: Good, can be expensive when running move C
        with n > 3
##DESC: For use on single colored games, identifies a
        set of initial positions/games given the number of
        colored blocks required and a set of moves
#moves = easymoves #moves can be set to easymoves or
        hardmoves depending on which group you are analyzing
#n = 2 #set n to an integer with domain of 2-8 where
        each n represents the number of single-colored boxes
#g = gamelist(n,moves,True) #True turns on verbose and
        will print results as they are encountered, you can
        remove it and access data through list that is

```

returned.

#####GAMELIST2
##LAST MODIFIED: 2015-12-05
##CONDITION: *Good, can be expensive when running move C with $n > 3$*
##DESC: *For use on two-colored games, identifies a set of initial positions/games given the number of colored blocks required and a set of moves*
#moves = easymoves #moves can be set to easymoves or hardmoves depending on which group you are analyzing
#n = 2 #set n to an integer with domain of 2-8 where each n represents the total number of colored boxes for example if you have box 4 colored boxes of red and blue then let n=4. this will generate the initial configurations of all subgames where 1 box is blue and 3 are red and 2 boxes are blue and 2 are red.
#g = gamelist2(2,moves,True) #True turns on verbose and will print results as they are encountered, you can remove it and access data through list that is returned.

#####GODSNUMBER
##LAST MODIFIED: 2014-02-11
##CONDITION: *Good*
##PURPOSE: *Find single God's number / diameter of graph for one single-colored game*
##DESC: *Creates an adjacency matrix and then computes God's number, printing to file both the adjacency matrix and final matrix $I+A+\dots+A^k$ where k is God's number*
#moves = easymoves
#position = [1,2,3]
#elements = permutations(position,moves) #use this or permutations2()
#fname = "SOMEFILENAME" #used for the name of the adjacency matrices that are saved as TXT files
#k = godsnumber(elements,moves,fname,True) #True turns on verbose with updates of each power currently checked or you may remove it.

#####SEARHPATH
##LAST MODIFIED: 2016-04-20
##CONDITION: *Limited*
##PURPOSE: *Designed to find a path from the initial configuration to any element in the 4x4 hard mode*

```

##DESC: Define a list of elements to be searched for by
using the formatting below. The searchpath function
accepts a single list of elements in scipy
Permutation syntax
##LIMITATIONS: This is an expensive program. It will
take up to about 3 days to search for a single
objective, objectives may be found quicker and they
are immediately returned when they are found.
#objective1 = Permutation(1,16)(2,12)(3,8)(5,15)(6,11)
(9,14)
#objective2 = Permutation(2,5)(3,9)(4,13)(7,10)(8,14)
(12,15)
#objective3 = Permutation(1,4,16,13)(2,8,15,9)
(3,12,14,5)(6,7,11,10)
#objective4 = Permutation(1,13,16,4)(2,9,15,8)
(3,5,14,12)(6,10,11,7)
#objective5 = Permutation(1,4)(2,3)(5,8)(6,7)(9,12)
(10,11)(13,16)(14,15)
#objective6 = Permutation(1,13)(2,14)(3,15)(4,16)(5,9)
(6,10)(7,11)(8,12)
#objectives = [objective1, objective2, objective3,
objective4, objective5, objective6]
#objectives = [objective1, objective2]
#print(searchpath(objectives))

#####ISADJACENT
##LAST MODIFIED: 2016-04-26
##CONDITION: Very good
##PURPOSE: Identify whether two elements of a group
defined by a list of generators are adjacent
##DESC: isAdjacent compares two elements by applying the
predefined transformations to the first element and
checks against the second
#piece1 = [[1,2],[5]] #each configuration is a list of
lists, where each internal list represents a color
and the numbers are the boxes to be configured
#piece2 = [[4,7],[3]] #this list has two colors the
first has boxes 4 & 7 colored the same color and then
box 3 is colored a second color
#print(isAdjacent(piece1,piece2,easymoves)) #returns
false because they're not adjacent
#piece3 = [[1,3,5,7]] #this list defines a single
colored board with boxes 1,3,5,7 colored in
#piece4 = [[2,4,6,8]]
#print(isAdjacent(piece3,piece4,hardmoves)) #returns
true

```


*#piece5 = [[1],[2],[3],[4],[5]] #this is the definition
of a five colored game*

Part VI

Appendix B - GAP Group Definitions

The definitions below are pre-defined GAP groups. Copy the code to a text editor and save as .g. Run code using the GAP system.

```
#GAP program and documentation available at gap-system.org
#The pre-defined groups of the Rubik's Slide for 2x2,3x3,4x4 easy and hard modes as defined by transformations, along with the easy mode groups of 5x5 and 6x6 boards.
#Uncomment read command with correct path to file for loading into GAP
#Read("../rubikslide.g");
#If you intend to use the GRAPE package uncomment line below
#LoadPackage("grape");;

rs2:=Group([(1,2)(3,4),(1,3)(2,4),(1,2,4,3)]);;

rs3e:=Group([(1,2,3)(4,5,6)(7,8,9),(1,7,4)(2,8,5)(3,9,6),
(1,3,9,7)(2,6,8,4)]);;

rs3h:=Group([(1,2,3)(4,5,6)(7,8,9),(1,7,4)(2,8,5)(3,9,6),
(1,2,3,6,9,8,7,4)]);;

rs4e:=Group([(1,2,3,4)(5,6,7,8)(9,10,11,12)(13,14,15,16),
(1,13,9,5)(2,14,10,6)(3,15,11,7)(4,16,12,8),
(1,4,16,13)(2,8,15,9)(3,12,14,5)(6,7,11,10)]);;

rs4h:=Group([(1,2,3,4)(5,6,7,8)(9,10,11,12)(13,14,15,16),
(1,13,9,5)(2,14,10,6)(3,15,11,7)(4,16,12,8),
(1,2,3,4,8,12,16,15,14,13,9,5)(6,7,11,10)]);;

rs5e:=Group([(1,2,3,4,5)(6,7,8,9,10)(11,12,13,14,15)
(16,17,18,19,20)(21,22,23,24,25),(1,21,16,11,6)
(2,22,17,12,7)(3,23,18,13,8)(4,24,19,14,9)
(5,25,20,15,10),(1,5,25,21)(2,10,24,16)(3,15,23,11)
(4,20,22,6)(7,9,19,17)(8,14,18,12)]);;

rs6e:=Group([(1,2,3,4,5,6)(7,8,9,10,11,12)
```

(13,14,15,16,17,18) (19,20,21,22,23,24)
 (25,26,27,28,29,30) (31,32,33,34,35,36)
 , (1,31,25,19,13,7) (2,32,26,20,14,8) (3,33,27,21,15,9)
 (4,34,28,22,16,10) (5,35,29,23,17,11) (6,36,30,24,18,12)
 , (1,6,36,31) (2,12,35,25) (3,18,34,19) (4,24,33,13)
 (5,30,32,7) (8,11,29,26) (9,17,28,20) (10,23,27,14)
 (15,16,22,21)]) ; ;