Spring 2015

Some combinatorial applications of Sage, an open source program

Jessica Ruth Chowning

SOME COMBINATORIAL APPLICATIONS OF SAGE, AN OPEN SOURCE PROGRAM

by

JESSICA RUTH CHOWNING

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN MATHEMATICS

2015

Approved by

Ilene H. Morgan, Advisor

David Grow

Gerald L. Cohen

ABSTRACT

In this thesis, we consider the usefulness of Sage, an online and open-source program, in analyzing permutation puzzles such as the Rubik's cube and a specific combinatorial structure called the projective plane. Many programs exist to expedite calculations in research and provide previously-unavailable solutions; some require purchase, while others, such as Sage, are available for free online. Sage is asked to handle a small permutation puzzle called Swap, and then we explore how it calculates solutions for a Rubik's cube. We then discuss projective planes, Sage's library of functions for dealing with projective planes, and how they relate to the card game Spot It! Since Sage is a free, open-source program, its limitations are a valid concern and are also discussed.

ACKNOWLEDGMENT

Deepest thanks to Dr. Ilene Morgan for being my academic advisor as well as my thesis advisor. Since I joined the mathematics department during my undergrad work, she has been a wonderful resource for classes to take, books to read for research, and for her endless help with this thesis. If not for her copious corrections, it would not be nearly as complete and consistent as it is now. I am also indebted to Drs. Grow and Cohen for serving on my committee and being my professors. All three have challenged me in various ways, and it has been a wonderful experience to see the limits of my abilities being pushed and expanded.

I would also like to thank my friends and family for supporting me during late nights and vacations spent working on this thesis. Your patience, belief in me, and willingness to bring me coffee and snacks has been instrumental and never forgotten. Particular thanks and gratefulness to my parents, Mark and Jennifer Chowning, for believing that I could successfully pursue mathematics. Unending gratitude to Samantha DiCenso, Kara Mihalik, Danforth Griesenaur, Frank Marshall, Chloe Entwhistle, Amy Cady, and Garion Lovig. Your friendship and tireless support mean the world to me, and I would not be who I am otherwise.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

There are few things as satisfying as pitting one's mind against a puzzle and finding a solution. Whether it's developing a mathematical proof or testing a hypothesis about the laws of nature, the challenge of solving the unknown has driven scientists and mathematicians to develop ever more sophisticated analytical tools. As the problems grow more complicated and intricate, so must the tools of investigation. Many powerful programs exist for various kinds of mathematical computation, but, for the university student trying to research on a budget, they are somewhat out of reach due to expensive subscription or purchase fees. With the introduction of Sage, an open-source program available for online use or free download, an alternative to expensive programs may exist.

A program is only as useful as its functions allow, and a free program is generally not as extensive as its paid counterpart. Open-source code, however, allows for users and developers to add functions as necessary, extending the program's library and usefulness. If Sage is to be a competitor with the programs already available, it will have to be able to handle a wide variety of problems and fields of mathematics. In this thesis, we will explore how Sage can be used to analyze combinatorial designs such as projective planes as well as how it handles permutation puzzles, including one famous example: the Rubik's cube.

2. AN INTRODUCTION TO SAGE

Sage is an open-source mathematical computing package available for free online. The source code itself can also be downloaded and set up on a personal computer for offline use. Given that it is an open-source package, it draws on many other mathematical packages (such as Gap, R, etc.), which opens up its usefulness to mathematicians and programmers from a variety of disciplines and backgrounds. The current functionality of the program is limited only by a particular version of the source code; since the program is open-source, the source code is open to development.

The interface itself, whether used online or built on a computer from the source code, resembles a command line. A user has only to open a worksheet to begin a new project. For new users, Sage provides a comprehensive tutorial, beginning with simple arithmetic and moving up to more specific, advanced fields such as group theory.

The blank worksheet for Sage (Figure 2.1) features a command line. Calculations in progress are denoted by a red line to the left of the entry and a green line underneath the output, and finished calculations take the place of the green "in progress" line (Figure 2.2).

Figure 2.1: The Sage worksheet.

Figure 2.2: A basic calculation in Sage (assigning values to a variable), and how Sage gives the output.

Sage's library encompasses many basic commands necessary for anything from simple graphing to working with more advanced branches of mathematics such as finite fields. It also allows users to define functions, which opens up opportunities for more specific calculations, based on their purpose for using Sage.

Figure 2.3 features an example of a defined function. The first line names the function `is_divisible_by()`, and in parentheses names variables that the function should be able to work with, as well as giving a default for the variable "divisor". In the case that a value is not assigned to the variable "divisor", it has a default and will not cause the program to output an error. The default also gives this particular function a wider use: testing whether a number is even or odd.

```
def is_divisible_by(dividend, divisor=2):
    return dividend%divisor == 0

is_divisible_by(6); is_divisible_by(6,2); is_divisible_by(6,4)
```

```
True
True
False
```

Figure 2.3: Defining a function that will test divisibility of two numbers.

The second line of the function definition has Sage perform several small steps at once. When calling the function, as shown in the third line of the figure, Sage assigns the number 6 to the variable “dividend” and the number 2 to the variable “divisor”. The calculation “dividend%divisor” tells Sage to find the remainder when “dividend” is divided by “divisor”. If the value assigned to the variable “dividend” is a multiple of the divisor, the remainder should be 0, and therefore the calculated remainder is tested. If it is equivalent to 0, Sage is told to output a True value, and if not, it outputs False.

This particular function has all of these calculations performed in one step. The same function could also be defined as in Figure 2.4. This function takes multiple steps, but is essentially the same as the first; the difference lies only in the number of lines. A default is still assigned to the variable “divisor” in case one is not assigned when the function is called, and the remainder is still compared to the value 0. The output is also either True or False, since the function’s purpose is to test divisibility.

```
def is_divisible_by(dividend, divisor=2):
    remainder=dividend%divisor
    output=remainder==0
    return output

is_divisible_by(3)
```

```
False
```

Figure 2.4: An alternate function for testing divisibility, demonstrated by assigning 3 to the variable “dividend” and allowing “divisor” to default to 2.

Both functions are valid. More possibilities exist, based on how in-depth the user wishes to get in the definition.

A worksheet can be accessed from the main page (whether a user is working online or offline), or the results can be printed as a PDF and saved to the computer. The PDF displays the name of the worksheet at the top of the page, and the top left corner gives the date the worksheet was printed.

3. PERMUTATION PUZZLES

One question that any student in a math class will eventually ask is, “But how can I use this in my everyday life?” Calculus finds application in calculating projectile motion in physics, while linear algebra and matrices are used in coding theory and finite fields in cryptography, to name a few examples. Mathematics even shows up in the games that people play every day, not just to predict how likely someone is to win a hand of poker, but even to provide strategies for winning. When it comes to permutation puzzles, it grants the player a solution for winning the game.

A permutation rearranges the elements of a set from a certain starting order. Elements are neither added nor removed, merely assigned a position that may or may not be different from the starting position. For example, given the set of integers from 1 to 10, a possible permutation would be to switch each even number with the odd number directly preceding it, changing the order 1,2,3,4,5,6,7,8,9,10 to 2,1,4,3,6,5,8,7,10,9. There are 10!, or 3,628,800, possible ways to order the integers 1 to 10, each of them being a permutation of the initial arrangement from least to greatest. Given n distinct elements, S_n (read as “the symmetric group on n elements”) is the group comprising permutations of said elements and whose operation is composition of those permutations. There are $n!$ permutations in S_n.

Various representations exist for the same arrangement of elements. One can visualize a set of tiles, labeled as the integers from 1 to 10, arranged in the same order as given in the previous paragraph. The same arrangement can be represented as:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 2 & 1 & 4 & 3 & 6 & 5 & 8 & 7 & 10 & 9 \end{bmatrix}$$

with the first row denoting the tile numbers and the second row indicating location of the tiles. Tile 2 would be in the first position, tile 1 in the second, etc.

Another representation combines the two rows as follows: $(1,2)(3,4)(5,6)(7,8)(9,10)$, and can be viewed as a function. In general, a cycle α of k elements is denoted as $(a_1, a_2, \ldots, a_k)$. For each i, $1 \leq i \leq k-1$, $\alpha(a_i) = a_{i+1}$, and $\alpha(a_k) = a_1$. General convention places the smallest element in a cycle at the beginning, though the cycle $(1,3,5)$, for example, is the same as $(3,5,1)$ and $(5,1,3)$. In the context of tiles and position, $\alpha(a_i) = a_{i+1}$ means that tile a_i is in position a_{i+1}.

As another example, suppose ten tiles numbered 1-10 have been arranged in the following order: 7,2,4,3,9,1,6,5,10,8. This is an example of a permutation of the first ten positive integers. This ordering can be expressed in the two given ways to express permutations:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 6 & 2 & 4 & 3 & 8 & 7 & 1 & 10 & 5 & 9 \end{bmatrix}$$

and $(1,6,7)(2)(3,4)(5,8,10,9)$.

The last representation can omit the cycle consisting solely of 2 since the element is already in its home position. If the game was to order the tiles from least to greatest, 2 would already be "solved." Additionally, the inverse of the permutation switches the rows, with the first being tile positions and the second being the tiles themselves.

Games have been built around this concept; provided with a set of elements that have been jumbled up and a set of legal moves, the player is challenged to rearrange the elements back into their home positions. The most famous example is the Rubik's cube, a six-faced cube broken into movable pieces called cubies so each face is divided into nine squares, called facelets, that are the elements permuted in the Rubik's cube. Legal moves

consist of series of rotations of the movable faces, which are fixed in the center of the cube. Once the cube has been scrambled, each piece must be moved back into its proper place so each side is the same color. The closer the cube is to being solved, the cleverer the moves have to be so that a new piece can be put in place without disturbing previously-solved cubies. Solving strategies will be discussed later.

A simpler permutation game is built from the aforementioned image of jumbled tiles. To simplify the process of solving such a puzzle and reducing the number of necessary and possible moves, only the integers 1-6 shall be considered.

The game of Swap, proposed by Jamie Mulholland in his course "Permutation Puzzles: A Mathematical Perspective" at Simon Fraser University, is presented to this end. Suppose tiles numbered from 1 to 6 are given in a certain jumbled order, or permutation, and the object is to put them back in ascending order. The game is complicated by defining only certain moves to be legal; since the game is more of an exercise to illustrate mathematical properties rather than a commercially-available system, there are no formal rules for legal moves. If legal moves are swapping any two tiles, then it can be shown by properties of permutations that any ordering of the tiles can be eventually solved. Any ordering of tiles can be represented as a permutation α, and switching two tiles can be represented as the 2-cycle, or transposition, (a, b), with tile a moving to position b and tile b moving to position a. Any permutation α can be represented as a series of transpositions (Dummit), and it will be shown later that a series of transpositions can be found that solves the permutation back to the "home" state.

Mulholland presents, in his lectures, the ordering $2,6,4,1,3,5$. As a first move, one might, for instance, swap the 1st and 4th tiles, resulting in the arrangement $1,6,4,2,3,5$.

The first tile is now solved and can be left alone. Continuing, one possible sequence of moves involves switching the following positions in this order: 2 and 4, 4 and 6, 4 and 5, and finally 3 and 4. Five moves were all that were necessary for solving the puzzle.

If the puzzle was represented in cycle notation, we would say that tile 1 is in position 4, tile 4 in position 3, tile 3 in position 5, tile 5 in position 6, tile 6 in position 2, and tile 2 in position 1, which ends the cycle. Labelling this as β, the permutation is $\beta = (1,4,3,5,6,2)$. Each of the swaps would be represented as $s = (a, b)$, with $s(a) = b$ meaning that the tile in position a is moved to position b. In order, they are $s_1 = (1,4)$, $s_2 = (2,4),\ s_3 = (4,6),\ s_4 = (4,5),\ s_5 = (3,4)$. Each intermediate arrangement of tiles can also be represented as a permutation.

Mathematically, applying each swap is a composition of permutations. Read from left to right, the first permutation should be the beginning arrangement of the tiles, and each subsequent cycle (in this case, a transposition since legal moves only allow the switching of two tiles at a time) would be the next swap in the sequence. So, the steps followed earlier to solve $2,6,4,1,3,5$ would result in:

$\beta s_1 s_2 s_3 s_4 s_5 = (1,4,3,5,6,2)(1,4)(2,4)(4,6)(4,5)(3,4)$.

The left-to-right convention is not universal. For example, some authors favor composing permutations from right to left (Pinter). Sage and Mulholland favor left-to-right, and so that convention will be used for continuity. What is universal is that the order of the permutations of the composition is important; composition of permutations is not commutative, and so changing the order of the swaps would not necessarily result in a solved puzzle. The moves themselves, as well as the order in which they are applied, matter for solving the puzzle.

The legal moves can dictate the solvability of the puzzle. If legal moves are defined as swapping any two tiles, any scrambled arrangement of tiles can be solved. This hinges on the fact that any permutation can be broken down into a series of transpositions. If α is a cycle of length m $(a_1, a_2, \dots, a_m)$, α can be written as, for example, $(a_1, a_2)(a_1, a_3)\dots(a_1, a_{m-1})(a_1, a_m)$. Using this procedure, any permutation σ in the symmetric group on n symbols, which is the set of all permutations of those n symbols and denoted as S_n, can be written as a composition of 2-cycles. This holds even for products of disjoint cycles, i.e., cycles with no elements in common. For example, if $\sigma = (1,12,8,10,4)(2,13)(5,11,7)(6,9)$ in S_{13}, it may be written as $(1,12)(1,8)(1,10)(1,4)(2,13)(5,11)(5,7)(6,9)$.

Applied to our game of Swap, the starting arrangement can be written as a permutation as before. This permutation can now be expressed as a composition of transpositions, each of which physically translates into switching two tiles. The resulting expression of the permutation gives a possible method for scrambling the tiles, in which case the solution would be reversing the process. It is certainly not the only solution, as the expression of a permutation as transpositions is not unique, but it is nonetheless valid.

If valid moves consist of moving more than two tiles, then not all arrangements can be solved. Mulholland presents another variant of Swap in which valid moves are 3-cycles, or taking three tiles and cycling them left or right amongst themselves. If the tiles themselves can take any free position independent of their fellows (they are not fixed in a track and do not have any other physical restrictions), certain positions may not be solvable by any series of 3-cycles. Solutions exist if and only if the rearranged

permutation can be represented as a series of legal moves. It then becomes necessary to study which permutations can be broken into 3-cycles and which ones can't.

As shown before, all permutations, including 3-cycles, can be expressed as a series of transpositions. The number of 2-cycles that makes up a permutation will be either even or odd; in the former case, the permutation is said to be even, and otherwise, it is odd. It can be shown that if a permutation α can be represented as an odd number of 2-cycles, no even-length series of transpositions exists that also represents α, and the same for an even permutation. So, if α can be described by an odd number of 2-cycles, it can only be represented by odd numbers of 2-cycles and is therefore designated to be odd.

If an arrangement of tiles can be represented by 3-cycles, it is an even permutation since 3-cycles can also be represented as a pair of transpositions. Given an arrangement of tiles, and with the rule that valid moves consist of permuting three tiles in either direction, the first step should be expressing the arrangement as a permutation and determining whether or not it is even. Given our previous example, expressed as $(1,4,3,5,6,2)$, the arrangement cannot be solved using this rule since it can be broken down into an odd number of transpositions, specifically $(1,4)(1,3)(1,5)(1,6)(1,2)$.

One permutation of tiles that would be solvable using 3-cycles would be $3,5,1,6,4,2$, or $(1,3)(2,6,4,5)$ in cycle notation. This is an even permutation and therefore solvable. One possible solution would be first taking the tiles in positions 1,2, and 3 and cycling them left, then 1,5,2 to the left, and then 2,4,6 to the right, i.e., apply $(1,3,2) \rightarrow 5,1,3,6,4,2$, $(1,5,2) \rightarrow 1,4,3,6,5,2$, and $(2,4,6) \rightarrow 1,2,3,4,5,6$. Note $(1,3,2)(1,5,2)(2,4,6) = (1,3)(2,5,4,6)$ is the inverse of the original permutation, as expected.

4. SAGE AND PERMUTATION PUZZLES

So far, solutions for the presented games of Swap have been obtained through guesswork. Sage can be used to obtain solutions for these puzzles, taking guesswork out of the equation. Any permutation puzzle can be expressed as a starting configuration represented by a permutation α, a home configuration ε, and any solution as a series of moves expressed as $\beta_1, \beta_2, \ldots, \beta_n$. Altogether, the beginning permutation and the application of moves makes up the following equation:

$\alpha\beta_1\beta_2 \ldots \beta_n = \varepsilon$.

We can write $\prod_{i=1}^{n} \beta_i = \beta$, where β represents the whole solution, so we can write $\alpha\beta_1\beta_2 \ldots \beta_n = \alpha\beta = \varepsilon$.

From this equation, we can see that α and β must be inverses since they come from the same permutation group, so a solution can be calculated as α^{-1}, or the inverse of the permutation that describes the initial configuration. This was illustrated at the end of Chapter 3.

Inverses of permutations are not difficult to calculate by hand, and Sage can be used to check the solution; for Swap, it can also give a pictorial representation of the initial arrangement of tiles. If necessary, Sage can also check the sign of the permutation as a solvability check. If the solution has been calculated by hand and then broken down into 2- or 3-cycles, Sage can be an accuracy check.

Going back to our first example, the six tiles scrambled to show 2,6,4,1,3,5, we first suppose that legal moves consist of switching the tiles in any two locations. Our solution was switching (in order) positions 1 and 4, 2 and 4, 4 and 6, 4 and 5, and lastly 3

and 4. Figure 4.1 presents the following code that can be used to represent this puzzle and process as well as check the solution:

```
S6=SymmetricGroup(6); S6
```

```
    Symmetric group of order 6! as a permutation group
```

```
begin=S6("(1,4,3,5,6,2)"); begin
```

```
    (1,4,3,5,6,2)
```

```
matrix([[1..6],[begin.inverse()(i) for i in [1..6]]]); #first row denotes the
place position, and the second is the name of the block
```

```
    [1 2 3 4 5 6]
    [2 6 4 1 3 5]
```

```
beta1=S6("(1,4)"); beta2=S6("(2,4)"); beta3=S6("(4,6)"); beta4=S6("(4,5)");
beta5=S6("(3,4)");
beta1, beta2, beta3, beta4, beta5;
```

```
    ((1,4), (2,4), (4,6), (4,5), (3,4))
```

```
beta=beta1*beta2*beta3*beta4*beta5; beta
```

```
    (1,2,6,5,3,4)
```

```
beta==begin.inverse()
```

```
    True
```

```
end=begin*beta; matrix([[1..6],[end.inverse()(i) for i in [1..6]]]);
```

```
    [1 2 3 4 5 6]
    [1 2 3 4 5 6]
```

```
def transpositions(perm):
    initial=Permutation(perm).to_cycles()
    a=len(initial)-1
    for i in [0..a]:
        b=len(initial[i])-1
        for j in [1..b]:
            print "(", initial[i][0], ",", initial[i][j], ")"
```

```
    ( 1 , 2 )
    ( 1 , 6 )
    ( 1 , 5 )
    ( 1 , 3 )
    ( 1 , 4 )
```

```
gamma1=S6("(1,2)"); gamma2=S6("(1,6)"); gamma3=S6("(1,5)"); gamma4=S6("(1,3)"); gamma5=S6("(1,4)");
gamma=gamma1*gamma2*gamma3*gamma4*gamma5;
gamma==beta
```

```
    True
```

Figure 4.1: Using Sage to check permutation representations of a game of Swap and the solution, as well as a function written to calculate the moves necessary to solve the puzzle, outputted as a series of 2-cycles. This given output is then tested for equivalence to our solution.

The first line sets up the symmetric group of order 6, which calculates all permutations of the integers 1-6. Our representation of the starting tiles, as well as each move, will be pulled from this group. The second line takes the variable ‘begin’ and assigns it to the permutation in S_6 that represents the starting arrangement of the tiles, and then displays the permutation. In addition, a matrix representation of the tiles is set up to check that the correct permutation is being used. The # starts a comment, just to serve as a reminder of what the code is doing and an explanation of the output.

From there, the variables beta1 to beta5 are assigned to the move permutations. If applied in the correct order to begin, and if in that order they are a solution, the end permutation should be the identity permutation, and the variables beta1 to beta5 should be the inverse of the begin permutation. The variable ‘end’ is assigned to the permutation that comes from multiplying ‘begin’ and ‘beta’ together (‘beta’ being the whole solution), and the end positions of the tiles are displayed in the same way their beginning positions were, allow a visual check that the solution is valid. Additionally, a function has been written to take the initial permutation and gives a series of swaps that solve the puzzle.

To check the sign (+1 or -1) of a permutation, corresponding to its parity (even or odd, respectively) in Sage, the following command is used in Figure 4.2:

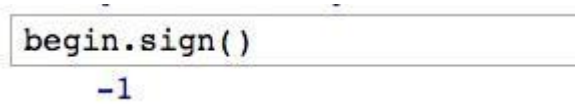


Figure 4.2: Using Sage to check the sign of the permutation describing the tile setup of 2,6,4,1,3,5; Sage displays a -1 for odd permutations and a 1 for even permutations.

Once a variable has been assigned in Sage, it will have certain properties. For example, a list will have a length and elements in a certain order. These properties, referred to in Sage's documentation as methods, can be called and worked with by appending `.[method]()` to the end of the variable name. In Figure 5, the variable in question is begin, which Sage recognizes as a permutation. Permutations have a sign, which is called in Sage by the method `.sign()`. From these, we can see that, while the 2,6,4,1,3,5 arrangement can be solved using any rules involving switching two boxes, it is an odd permutation, and therefore cannot be solved using 3-cycles.

Another example present in Figure 4.3, using eight tiles, can be represented by (1,3)(2,4,7,6,8). This time, valid moves are 3-cycles, so it is necessary to check the sign of the beginning permutation before proceeding.

```
S8=SymmetricGroup(8); S8
    Symmetric group of order 8! as a permutation group
begin=S8("(1,3)(2,4,7,6,8,5)"); begin; matrix([[1..8],[begin.inverse()(i) for
i in [1..8]]])
    (1,3)(2,4,7,6,8,5)
    [1 2 3 4 5 6 7 8]
    [3 5 1 2 8 7 4 6]
begin.sign()
    1
begin.inverse()
    (1,3)(2,5,8,6,7,4)
delta1=S8("(1,2,3)"); delta2=S8("(2,3,4)"); delta3=S8("(4,5,7)"); delta4=S8("
(6,7,8)");
delta1, delta2, delta3, delta4
    ((1,2,3), (2,3,4), (4,5,7), (6,7,8))
delta=delta1*delta2*delta3*delta4; delta; delta==begin.inverse()
    (1,3)(2,5,8,6,7,4)
    True
end=begin*delta; end; matrix([[1..8],[end.inverse()(i) for i in [1..8]]])
    ()
    [1 2 3 4 5 6 7 8]
    [1 2 3 4 5 6 7 8]
```

Figure 4.3: Applying the same process as before to check a solution for Swap.

As before, the symmetric group is defined that contains all of the permutations being used. The beginning permutation is defined, and the output is visually displayed. Since a check for solvability is necessary, the `.sign()` method must be called. The inverse is displayed, and a possible set of solving moves are assigned to the variables delta1 through delta4, which are multiplied together and checked against the inverse to see if they are equivalent. This is not the only valid series of moves that will grant a solution, but composed together, they should be the inverse of the initial permutation.

Once checked, the end variable is assigned, and this time displayed. 'begin' multiplied by a valid solution yields the identity permutation, since no positions should have switched with each other, and as a failsafe, the matrix showing 'end' as tiles is displayed.

Utilizing Sage as a physical check as well as a calculator can cut down on the time needed to solve certain puzzles. Not only can it handle small permutation puzzles, but it possesses the functionality to deal with more complicated puzzles, as well. The Rubik's cube, mentioned earlier as a famous example of a permutation puzzle, has been programmed into Sage, with all legal moves and positions residing in a permutation group called `CubeGroup()`, a subgroup of S_{48}. The generators, or permutations that combine to make all elements of the permutation group, represent the base moves available to a Rubik's cube: moving each face of the cube clockwise.

Writing each face move as a permutation is tedious as each is made up of five cycles. For example, turning the front face clockwise results in a configuration that can be written as $(6,25,43,16)(7,28,42,13)(8,30,41,11)(17,19,24,22)(18,21,23,20)$. One way to refer to each move is by the face that is turned clockwise; the before permutation

can therefore be called F. In the same way, the other five principal moves are B, R, L, U, D. This notation is referred to as Singmaster notation, after David Singmaster, an American mathematician who coined the notation and presented his own solution to the Rubik's cube in his *Notes on Rubik's 'Magic Cube'*. The numbering of each facelets can be found in Figure 4.4.

```
Rubik.display2d("")
                +--------------+
                |  1    2    3 |
                |  4   top   5 |
                |  6    7    8 |
+---------------+--------------+--------------+---------------+
|  9  10  11    | 17   18   19 | 25   26  27  | 33  34  35    |
| 12 left 13    | 20  front 21 | 28 right 29  | 36 rear 37    |
| 14  15  16    | 22   23   24 | 30   31  32  | 38  39  40    |
+---------------+--------------+--------------+---------------+
                | 41   42   43 |
                | 44 bottom 45 |
                | 46   47   48 |
                +--------------+
```

Figure 4.4: Initial labeling of the facelets of the Rubik's cube, which will be used throughout the Rubik group to display a given configuration of the cube.

The programming for the Rubik's cube in Sage is extensive enough that it can display 2D and 3D representations of the cube in a certain legal orientation, as well as after each move applied to the cube. This provides the same usefulness that Sage had in checking and providing solutions for the Swap games; the program can be used to solve the cube and display the solution as a series of legal moves. Sage contains methods to check whether a given configuration of the cube is legal or possible, return the current

state of a cube, and to return a random scrambling of the cube. Given enough time and a scrambled state of the cube, Sage can even give a solution in terms of face rotations, or in Singmaster notation. This takes quite a long time, as the algorithm used by the `.solve()` method takes the following steps:

This algorithm

1. constructs the free group on 6 generators then computes a reasonable set of relations which they satisfy
2. computes a homomorphism from the cube group to this free group quotient
3. takes the cube position, regarded as a group element, and maps it over to the free group quotient
4. using those relations and tricks from combinatorial group theory (stabilizer chains), solves the "word problem" for that element.
5. uses python string parsing to rewrite that in cube notation. ("Rubik's cube group functions")

The "solution" is then displayed as a sequence of moves necessary to put a solved cube in the given scrambled position, and would therefore have to be inverted. Figure 4.5 illustrates a scrambled cube and how to solve it using Sage.

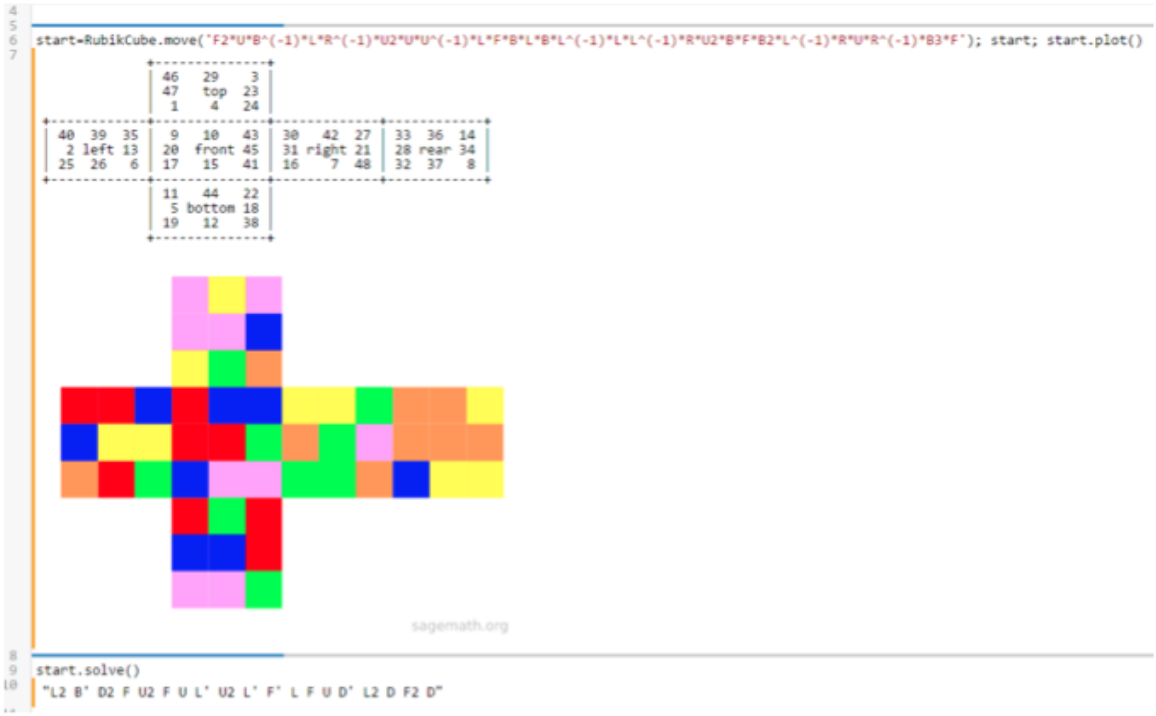


Figure 4.5: The position of the cube is given as the sequence of moves performed to scramble it.

The facelet positions have been given in Singmaster notation as well as in a colored layout. The cube colors in Sage do not match up with commercially-available cubes, but serve to illustrate a basic idea of what the cube would look like unfolded. Then, Sage is told to solve the cube, outputting the sequence “L2 B’ D2 F U2 F U L’ U2 L’ F’ L F U D’ L2 D F2 D”. This is actually a simplified sequence to scramble the cube, and the solution would therefore be the inverse, starting with a counterclockwise rotation of the bottom face and ending with two rotations (in this case, counterclockwise or clockwise does not matter) of the left face. This is not immediately clear and could stand to be updated in later versions of Sage.

While the program cannot yet solve other Rubik's cubes, the current functionality could be extended to *n*x*n*x*n* cubes for $n > 3$. To start, the facelets of the 4x4x4 cube (called Rubik's Revenge) or the 5x5x5 cube (called Professor's Cube) would be assigned numbers in the same way that the Rubik's cube was, and the generating permutations would be built from there (see Appendix B). Legal moves would be sequences of these permutations. Each group would have twelve generators (the $n = 5$ cube follows suit for $n = 3$ and ignores the middle slice). We therefore consider 96 facelets for the $n = 4$ cube, and 144 for $n = 5$.

Practically, when solving larger cubes by hand, many prefer to use the reduction method, which takes a larger cube and puts it in the state of the $n = 3$ cube. For example, for $n = 4$, the four center facelets of a face are treated as one center piece, where it doesn't matter the orientation of each individual facelet; any move permuting the four centers is seen to be invisible. Additionally, the 2 center facelets along an edge are seen as one facelet, and each corner is handled as a single cubie. The first half of solving the cube is making all of the centers, edges, and corner pieces the same color, therefore reducing the cube to an $n = 3$ cube; the solution can then be completed using methods to solve the $n = 3$ cube. Specialized algorithms of moves exist that will set a facelet in place. Opposite colors must be kept in mind, as they are used by solvers to orient the colors correctly. Possible errors in the setting of edges and corners may arise, but algorithms exist to correct those parity errors, as well. Whether n is odd or even, the technique works; the only differences come in moves that will set the reduced "cubies".

Once the reduced "cubies" have been solved, the task is to solve the 3x3x3 cube. One method for beginning is to solve the top "cross" first, then the whole first layer

(making sure edge facelets follow the same order that the center facelets of the side faces follow). The second layer is solved, and then the bottom face only, leaving the third layer's corner and edge facelets. Other algorithms exist, some preferred by speedcubers and others depending on the developer of the method. Like a traditional puzzle, there is no "right" way to solve a permutation puzzle, and the Rubik's cube is no exception.

When Sage presents a solution for a cube, it does not necessarily follow this formatting. The solution more directly follows as an inverse of the beginning permutation, much like the solutions for the Swap games were found. This is a faster, though far less intuitive solving of the cube as the solution could potentially solve many cubies at once, and many algorithms presented to solve the cube by hand focus on one cubie at a time.

If Sage was also applied to $n > 3$ cubes, the solution process would take advantage of the existing code that solves $n = 3$ cubes. While reduction makes use of the fact that certain facelets will never be in certain places on the cube (for example, a center will never be a corner, and vice versa) and moves single cubies into place without disturbing already solved ones, it does not present a quick solution. Sage takes advantages of a few diverse algorithms to calculate solutions, depending on the contributing developer, but the default algorithm gives a solution in the fewest moves necessary. All that would be necessary for Sage to handle $n > 3$ cubes would be to create a cube group with twelve generators and modify the existing code to call those generators.

5. SPOT IT! AND PROJECTIVE PLANES

Permutations can appear in different games whose objectives are not necessarily reordering scrambles tiles or facelets. Spot It!, a card game focused on matching symbols, presents cards with small sets from a given list of symbols. The object of the game is to find the shared symbol in a pair of cards. Each card has the same number of symbols, and for Spot It!, there are two less cards than there are symbols total.

The collections of symbols on each card are not randomized; they are specifically chosen such that any pair of cards will have a common symbol. The combinations of symbols are not what make this game mathematically interesting. Rather, it is the fact that the game is a physical representation of what is in mathematics called a projective plane. A projective plane is made up of a series of lines, or blocks, containing a certain number of points. Any two blocks will share exactly one point in common.

The projective plane is a specific case of a balanced incomplete block design (BIBD), which takes a set of elements and organizes them in such a way that certain interesting characteristics arise. Five parameters are considered for a BIBD: number of elements or points (v), number of blocks or lines (b), length of block or number of elements in each block (k), degree of element or number of blocks in which the element appears (r), and number of times a subset of elements will appear in the design (λ). If the subsets have two points, then the design is called a 2-design; a projective plane is a type of 2-design in which $\lambda = 1$. Projective planes are also symmetric, with $v = b$.

Any block design is subject to the following necessary conditions: $vr = kb$ and $\lambda(v - 1) = r(k - 1)$. Given that a projective plane satisfies $\lambda = 1$ and $v = b$ (therefore

reducing the first equation to $r = k$) the second equation simplifies to $v - 1 = k(k - 1)$. The order n of a projective plane is defined to be $k - 1$. Substituting, we obtain $v - 1 = k(k - 1) = (n + 1)n$. So, the number of points in a projective plane is $v = n^2 + n + 1$. Since the design is symmetric, there must also be $n^2 + n + 1$ blocks, and each point has degree $n + 1$. (Beth)

A children's version of the game, called Spot It! Jr., exists with thirty-three cards and thirty-three symbols; the degree of each symbol as well as the number of symbols on each card is 6, making Spot It! Jr. a projective plane of order 5. In this game, no block is missing. The fact that this game as well as the Spot It! game plus its missing blocks are projective planes will be verified later.

In a game of Spot It!, there are 57 symbols total, meaning the plane has order 7. Unlike Spot It! Jr., which represents a complete projective plane, there are only 55 cards in a Spot It! deck, which means that the game is missing two cards to be considered a complete projective plane. Each card has eight symbols, and each symbol would ideally appear eight times if the game was a full projective plane. The two missing cards means that there is one symbol of degree 6, fourteen of degree 7, and forty-two of degree 8. The missing cards do not affect the playability of the game, as each of the remaining cards share pairs independent of each other.

For Spot It! to fully represent a projective plane, it would need two more blocks. Each symbol can be assigned a number from 0-56; particulars can be found in Appendix C. The degrees of the points, given our particular assignments, can be found in Table 5.1.

Table 5.1: Orders of the points in Spot It! Game.

Degree	Points
6	44
7	0, 1, 8, 17, 21, 23, 27, 31, 32, 33, 40, 41, 47, 50
8	2, 3, 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 16, 18, 19, 20, 22, 24, 25, 26, 28, 29, 30, 34, 35, 36, 37, 38, 39, 42, 43, 45, 46, 48, 49, 51, 52, 53, 54, 55, 56

Since the two missing blocks, which shall be called B1 and B2, must share a point in common, and point 44 must have degree 8, both B1 and B2 must contain point 44. The existing blocks were also assigned a number from 0 to 56, and the blocks in which points of degree 7 can be found in Table 5.2.

Table 5.2: Blocks in which the points of degree 6 and 7 of Spot It! appear

Point	Blocks
0	5, 23, 24, 36, 39, 43, 51
1	11, 28, 35, 38, 39, 44, 49
8	15, 19, 25, 32, 45, 46, 51
17	22, 27, 28, 37, 40, 46, 50
21	8, 9, 16, 19, 38, 42, 48
23	6, 9, 23, 33, 34, 37, 47
27	13, 16, 20, 27, 30, 36, 53
31	1, 2, 14, 20, 25, 33, 49
32	0, 1, 5, 18, 40, 42, 54
33	6, 11, 15, 18, 26, 41, 53
40	2, 4, 12, 22, 41, 43, 48
41	10, 12, 13, 32, 34, 35, 54
47	0, 4, 17, 30, 44, 45, 47
50	8, 10, 14, 17, 24, 26, 50

From here, it is a matter of sorting these points into B1 and B2. Since each must appear with point 44, it becomes necessary to have another point of comparison. Sorting point 0 into B1, the blocks of each subsequent point are compared to those of point 0's. If they already share one in common, the point is sorted into B2, as the point and 0 have appeared together before. If not, the point goes in B1. Following this method, B1 and B2 were constructed as {0,17,21,31,33,41,44,47} and {1,8,23,27,32,40,44,50}. Combining these with the existing blocks forms a projective plane of order 7, which will be verified in the next chapter.

6. USING SAGE TO ANALYZE PROJECTIVE PLANES

So far, Sage has been used primarily as a calculator for solutions of various puzzles. Given its wide library of pre-existing functions, as well as easily-accessed source code, Sage can be used as an analysis tool, as well, especially for projective planes.

Block designs can be built from just a few parameters, and some of the more specialized ones, such as projective planes, can be called by their names. Most generally, a BIBD can be built by the following line of code: `balanced_incomplete_block_design(v, k, existence=False, use_LJCR=False)`. Assuming that the inputs for each parameter correspond to a valid design, Sage will build it and can go to the La Jolla Covering Repository if use_LJCR is set to True when it does not know how to build the design. Figure 6.1 demonstrates alternatives for building an incidence structure.

```
FanoPlane=IncidenceStructure(7, [[0,1,2],[0,3,4],[0,5,6],[1,3,5],\
                                 [2,4,5],[2,3,6],[1,4,6]]); FanoPlane; \
                                    FanoPlane.blocks()
Incidence structure with 7 points and 7 blocks
[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6], [2, 3, 6], [2, 4, 5]]

Incidence=matrix([[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[1,0,0,0,0,1,1],\
                  [0,1,0,1,0,1,0],[0,1,0,0,1,0,1],[0,0,1,1,0,0,1],[0,0,1,0,1,1,0]])\
                     ; Incidence
[1 1 1 0 0 0 0]
[1 0 0 1 1 0 0]
[1 0 0 0 0 1 1]
[0 1 0 1 0 1 0]
[0 1 0 0 1 0 1]
[0 0 1 1 0 0 1]
[0 0 1 0 1 1 0]

fanoplane=IncidenceStructure(Incidence); fanoplane; fanoplane.blocks()
Incidence structure with 7 points and 7 blocks
[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6], [2, 3, 6], [2, 4, 5]]
```

Figure 6.1: Alternative commands for building projective planes.

General incidence structures can be built in a few different ways: by giving the total number of points and a set of blocks or by giving the incidence matrix of the points and blocks. This is demonstrated by building the Fano plane, an incidence structure that also happens to be a projective plane of order 2 as defined in Chapter 5. In the first part of the figure, the plane is built given the total number of points 7 (this can also be handled by giving a list of the points) and the construction of the individual blocks. The next uses the incidence matrix of the structure to build the Fano plane.

When it comes to building a projective plane, there are various approaches. Users can use the methods demonstrated previously, or they could call `designs.projective_plane(n)`, which will construct a projective plane of order n. This doesn't allow for any control of which points are assigned to which blocks, but usually isn't a problem for smaller orders. According to the Bruck-Ryser-Chowla theorem, "If a symmetric design with $\lambda = 1$ and order n exists and if $n \equiv 1$ or 2 (mod 4) then n can be expressed as a sum of two integral squares." (Hughes) This combined with the necessary conditions for projective planes allows for the possible existence of planes of order 10 and 12, but not for order 6. In fact, exactly one plane of each order exists (up to isomorphism) for order 2, 3, 4, 5, 7, and 8 (Cherowitzo). This means that it doesn't matter how we arrange the individual points in the blocks for a projective plane; if two incidence structures are isomorphic, then there exists a bijection between them that will map the points of one structure to the other and preserve the block structure. For example, if one structure consists of the blocks [a,b] and [c,d], and the other of the blocks [1,2] and [3,4], then these structures are the same with a mapping to 1, b to 2, c to 3, and d to 4, and vice versa.

If a specific design is desired, it can be built from the incidence matrix of the points, as well as by giving the total number of points as well as a list of the blocks themselves. This last method will be how we build and verify the projective planes represented in the Spot It! and Spot It! Jr. games.

Spot It! Jr. is meant to be a projective plane of order 5, meaning 31 points and 31 blocks. Sage doesn't require an incidence structure's points to be integers. For example, the example on the previous page could be built in Sage with the command `A=designs.IncidenceStructure([['a','b'],['c','d']])`. This yields an incidence structure with the points a, b, c, and d, and two blocks [a,b] and [c,d]. The same process could be used to model the Spot It! Jr. game in Sage, with the points being the names of the symbols on each card. The blocks would then be built with the symbol names as elements. To reduce error, each symbol was assigned a number from 0 to 30, as well as each block. A full list can be found in Appendix C.

To check that Spot It! Jr. is actually a projective plane of order 5, Sage can be made to build a plane of order 5 and check that the two are isomorphic to each other. Two incidence structures are said to be isomorphic if there is a one-to-one and onto mapping from the point set of one to the point set of the other that preserves the block structure. Only one plane of order 5 exists up to isomorphism, so any construction of 31 points into 31 blocks that each share only a single point in common must be isomorphic to the existing plane. By calling the method `[design].is_isomorphic([other], certificate=False)`, Sage compares the two designs to each other to determine if one is just a renumbering of the other. If the certificate is set to True, and the two planes are determined to be isomorphic,

Sage will come up with a bijection between the two. In this case, the bijection would tell us what point in Sage's built plane of order 5 corresponds to a given symbol in Spot It! Jr.

```
Spot_It_Jr=IncidenceStructure(31,[[8,12,16,24,27,30],[2,3,7,15,17,24],\
                                  [1,2,9,25,29,30],[0,4,11,14,15,30],\
                                  [8,9,15,22,26,28],[7,9,11,20,21,27],\
                                  [11,18,19,24,26,29],[0,2,16,21,23,26],\
                                  [6,17,19,21,28,30],[5,15,16,19,20,25],\
                                  [4,9,13,16,17,18],[8,10,11,17,23,25],\
                                  [4,6,7,12,25,26],[3,4,5,8,21,29],\
                                  [0,1,7,8,13,19],[7,10,14,16,28,29],\
                                  [0,12,17,20,22,29],[0,3,18,25,27,28],\
                                  [1,4,20,23,24,28],[2,6,8,14,18,20],\
                                  [5,7,18,22,23,30],[6,13,15,23,27,29],\
                                  [0,5,6,9,10,24],[2,5,11,12,13,28],\
                                  [3,10,13,20,26,30],[1,10,12,15,18,21],\
                                  [2,4,10,19,22,27],[13,14,21,22,24,25],\
                                  [1,5,14,17,26,27],[1,3,6,11,16,22],\
                                  [3,9,12,14,19,23]]); Spot_It_Jr
Incidence structure with 31 points and 31 blocks

pp2=designs.projective_plane(5); pp2
Incidence structure with 31 points and 31 blocks

pp2.is_isomorphic(Spot_It_Jr)
True

pp2.is_isomorphic(Spot_It_Jr, certificate=True)
{0: 0, 1: 1, 2: 7, 3: 8, 4: 19, 5: 18, 6: 4, 7: 16, 8: 9, 9: 17, 10: 25, 11: 24, 12: 14,
13: 22, 14: 21, 15: 27, 16: 23, 17: 29, 18: 15, 19: 6, 20: 3, 21: 20, 22: 10, 23: 26, 24:
30, 25: 28, 26: 11, 27: 2, 28: 5, 29: 12, 30: 13}
```

Figure 6.2: Testing Spot It! Jr for isomorphism to a projective plane of order 5 and the bijection between the two structures.

Now, given the fact that Spot It! is not a full projective plane due to two missing blocks, the existing structure can be put into Sage, but will not be isomorphic to a projective plane of order 7. The missing blocks were reconstructed in the previous

chapter, so the completed structure can be tested against a plane of order 7 that Sage builds, and the blocks are verified.

```
Spot_It=IncidenceStructure(57,[[2,11,13,14,30,32,47,53],[31,32,34,46,51,52,55,56],\
                               [7,11,12,31,35,40,42,54],[7,10,13,18,22,28,44,46],\
                               [6,18,39,40,45,47,48,51],[0,3,28,32,37,42,43,45],\
                               [4,12,14,23,26,33,45,46],[4,9,11,29,37,44,48,55],\
                               [11,15,21,22,36,45,50,56],[6,9,13,21,23,34,38,42],\
                               [12,13,19,37,41,49,50,51],[1,15,18,33,34,35,37,53],\
                               [2,9,15,25,40,41,43,46],[14,16,18,27,36,41,42,55],\
                               [2,3,4,18,24,31,38,50],[2,8,19,22,33,42,48,52],\
                               [3,21,27,30,35,46,48,49],[9,16,26,28,35,47,50,52],\
                               [7,9,24,32,33,36,39,49],[4,7,8,16,21,43,51,53],\
                               [9,10,19,20,27,31,45,53],[3,6,12,25,36,44,52,53],\
                               [4,17,19,28,30,34,36,40],[0,2,10,23,29,35,36,51],\
                               [0,7,14,20,25,34,48,50],[6,8,14,15,28,29,31,49],\
                               [6,10,30,33,43,50,54,55],[2,6,7,17,26,27,37,56],\
                               [1,3,9,14,17,22,51,54],[14,19,35,38,39,43,44,56],\
                               [12,22,24,27,29,34,43,47],[15,20,24,26,30,42,44,51],\
                               [3,8,10,11,26,34,39,41],[16,22,23,25,30,31,37,39],\
                               [23,24,28,41,48,53,54,56],[1,7,29,30,38,41,45,52],\
                               [0,4,13,15,27,39,52,54],[11,17,18,20,23,43,49,52],\
                               [1,2,12,20,21,28,39,55],[0,1,6,11,16,19,24,46],\
                               [10,12,15,16,17,32,38,48],[3,13,16,20,29,33,40,56],\
                               [18,19,21,25,26,29,32,54],[0,22,26,38,40,49,53,55],\
                               [1,4,10,25,42,47,49,56],[8,20,36,37,38,46,47,54],\
                               [8,13,17,24,25,35,45,55],[3,5,7,15,19,23,47,55],\
                               [5,10,14,21,24,37,40,52],[1,5,13,26,31,36,43,48],\
                               [5,17,29,39,42,46,50,53],[0,5,8,9,12,18,30,56],\
                               [2,5,16,34,44,45,49,54],[5,11,25,27,28,33,38,51],\
                               [4,5,6,20,22,32,35,41],[0,17,21,31,33,41,44,47],\
                               [1,8,23,27,32,40,44,50]]); Spot_It
Incidence structure with 57 points and 57 blocks

pp=designs.projective_plane(7); pp
Incidence structure with 57 points and 57 blocks

pp.is_isomorphic(Spot_It)
True
```

Figure 6.3: Testing the incidence structure of the Spot It! blocks and the missing blocks for isomorphism to a projective plane of order 7.

Since Sage is open-source, a solution or improvement can be implemented in a matter of weeks. Code is built and tested, and once it is given a positive review, it can be integrated into the program. For example, developers have pulled an outside program called bliss to expedite the calculation of a projective plane's automorphism group (the set of isomorphisms from the plane to itself). Before, such a calculation could take

several hours and a large amount of memory. Utilizing bliss, the calculation time was trimmed down to a matter of seconds, and the storage needed was also vastly reduced. Until very recently, Sage did not possess the code necessary to calculate, for example, the stabilizer of a block of a system (the set of automorphisms of the design that leave the block unchanged), but code to provide this additional functionality was recently incorporated.

If a particular calculation does not call for the source code to be changed, a function can be written to perform the desired task. Instead of computing the missing blocks of Spot It! by hand, Sage could be made to do that task itself. There are a few possibilities for code that could be written to handle such a task. One might involve comparing the blocks from Spot It! to those in a projective plane of order 7, come up with a bijection between the points, and apply that bijection to the blocks in Spot It! From there, it is a matter of finding the blocks that aren't represented and determining the symbols that make up those blocks from the bijection.

This is somewhat imprecise and roundabout. Another possibility would be to look at the incidence matrix for Spot It!, where the rows represent the points from 0 to 56, and the columns are the blocks from 0 to 54. Calling this matrix M, and letting `M'=M*M.transpose()`, M' is a 57x57 matrix, with entry $m_{i,j}$ telling how many times point m_i appears with point m_j in the design. The diagonal would then give the degree of each point. So, if the first row corresponds to point 0, and the first column would also correspond to point 0, entry $m_{1,1}$ should be the degree of point 0. The missing blocks (again, we can label them B1 and B2) would be built by first finding the row i such that $m_{i,i} = 6$; this will correspond to the point i by Sage's indexing system. This point will go

in both blocks. From there, it is a matter of finding all points of degree 7, picking one, which shall be denoted a, with degree 7, placing it in B1, and then reading across the row to find the columns with a 0; these will correspond to points with which a has not yet appeared, and must also be placed in B1. The remaining points of degree 7 should be sorted into B2.

Doing these calculations in Sage is feasible, but the output can be somewhat hard to read given the size of the matrices. To illustrate the above process on a smaller scale, let's say we're given the following five blocks: [0,1,2], [0,3,4], [0,5,6], [1,3,5], [2,4,5]. Based on the fact that there are seven distinct points, and if we are trying to build a projective plane, then we should be trying for a projective plane of order 2. Inputting this design into Sage, we come up with the following for the incidence matrix and Mstar:

```
reduced_design=IncidenceStructure(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[2,4,5]]); reduced_design
    Incidence structure with 7 points and 5 blocks
M=reduced_design.incidence_matrix(); M;
    [1 1 1 0 0]
    [1 0 0 1 0]
    [1 0 0 0 1]
    [0 1 0 1 0]
    [0 1 0 0 1]
    [0 0 1 1 1]
    [0 0 1 0 0]
Mstar=M*M.transpose(); Mstar;
    [3 1 1 1 1 1 1]
    [1 2 1 1 0 1 0]
    [1 1 2 0 1 1 0]
    [1 1 0 2 1 1 0]
    [1 0 1 1 2 1 0]
    [1 1 1 1 1 3 1]
    [1 0 0 0 0 1 1]
Mstar.diagonal()
    [3, 2, 2, 2, 2, 3, 1]
```

Figure 6.4: Reduced design.

From the Mstar matrix, we see that point 6 has degree 1, placing it in both B1 and B2, and points 1, 2, 3, and 4 have degree 2 and will be sorted based on how often the pairs between them have appeared in the design. The second row of the matrix gives incidences for point 1, and we can see that the pairs [1,4] and [1,6] have not appeared yet. Therefore, B1 can be assigned points 1, 4, and 6. This leaves 2 and 3 to be assigned to B2 along with 6. Our full design then becomes [0,1,2], [0,3,4], [0,5,6], [1,3,5], [2,4,5], [1,4,6], and [2,3,6].

The function written to handle this sort of calculation and sorting could be much more general and applied to different, incomplete projective planes. If handed a design that is two blocks short of being a projective plane, the function should be able to find the point that should be sorted into both of the missing blocks, the points that should be sorted into one block or the either, and then from there, find the point pairs that have not appeared together. One function that could handle this is shown in Figure 6.5.

```
def build_blocks(design, points, order=2):
    p=points-1; n=order+1; M=design.incidence_matrix(); Mstar=M*M.transpose(); diagonal=Mstar.diagonal();
    list1=[i for i in [0..p] if diagonal[i]==(n-2)];
    print "Point to go in both B1 and B2 is", list1;
    list2=[i for i in [0..p] if diagonal[i]==(n-1)];
    print "Points to be sorted into B1 or B2 are", list2;
    a=len(list2)-1;
    for j in [0..a]:
        list3=[k for k in [0..p] if Mstar[list2[j],k]==0];
        print "Point", list2[j], "has not appeared with points", list3;


build_blocks(reduced_design, 7, 2)
```

```
Point to go in both B1 and B2 is [6]
Points to be sorted into B1 or B2 are [1, 2, 3, 4]
Point 1 has not appeared with points [4, 6]
Point 2 has not appeared with points [3, 6]
Point 3 has not appeared with points [2, 6]
Point 4 has not appeared with points [1, 6]
```

Figure 6.5: An example of a function designed to help rebuild two missing blocks from a projective plane.

7. CONCLUSIONS AND FUTURE POSSIBILITIES

Sage's versatility and use of outside programs makes it an ideal tool for analyzing mathematical problems. We have seen how it can be used to handle small puzzles, and even how it can formulate a solution to solve a Rubik's cube. It can also be given an incidence structure and, if the functionality does not yet exist to solve a particular problem, then the open-source code and active forums allow for new code to be added and tested in a matter of weeks, if not days. For instance, after I inquired about the existence of a method for finding the stabilizer of a block in a BIBD, a developer set out to program such a method, and it was recently given a positive review and has been incorporated into Sage. If users have experience with developing programs, they can write scripts themselves and contribute to the ever-growing list of tools that Sage can use to provide answers.

If development is of no interest to the user, then basic functions can be written to handle a process. Limitations here would be that a basic knowledge of programming is a must, as well as quirks with Sage's indexing and the habit of composing permutations left to right rather than right to left as presented in many algebra textbooks, for example. There is also the aforementioned issue of Sage's presented solution to a Rubik's cube. It is not immediately clear that the given sequence that Sage comes up with is a scrambling, rather than a descrambling, of the cube. This could be potentially rectified by adding a line or two of code that would take the "solution" and invert it, or even by adding a comment that states "Reversing the given sequence will unscramble this Rubik's cube."

There is enough interest in $n = 4$ and $n = 5$ Rubik's cubes that one could foresee the possibility of expanding Sage's capabilities to handle these larger cubes.

These are rather minor inconveniences and conventions of the program that, once the user has been familiarized with them, do not detract from the functionality and usefulness of the program. Sage's open-source nature allows for collaboration with many specialized programs and languages (bliss and Python, for example). While still a developing program, there is not much it can't already handle, and the limitations mentioned above are either already being patched or can be fixed. Its limitations do not outweigh the already numerous problems the program can already handle. Since it does not require a subscription or purchase to use, or even that the user download it, Sage could be an incredibly useful analytical tool for students of mathematics in the future.

APPENDIX A

PERMUTATION GROUPS IN SAGE

```
a = Permutation( '(1,2)(3,4)(5,6)' ); a; b = Permutation( '(5,6)' ); b; e = Permutation( '' ) ; e
```

```
[2, 1, 4, 3, 6, 5]
[1, 2, 3, 4, 6, 5]
[]
```

```
e(1)
```

```
Traceback (click to the left of this block for traceback)
...
TypeError: i (= 1) must be between 1 and 0
```

```
a(1)
```

```
2
```

```
a([2,3,1])
```

```
Traceback (click to the left of this block for traceback)
...
TypeError: i (= [2, 3, 1]) must be between 1 and 6
```

```
a(2)
```

```
1
```

```
a*b # permutatons are left to right, and of course SAGE is case-sensitive
```

```
[2, 1, 4, 3, 5, 6]
```

```
S6=SymmetricGroup(6)
a=S6('(1,2)(3,4)(5,6)')
b=S6('(5,6)')
```

Figure 1: Two ways to define a permutation: one as a function, and another as an element of a group; as well as legal and illegal operations to perform

```
a;b
```

```
(1,2)(3,4)(5,6)
(5,6)
```

```
a(1)
```

```
2
```

```
a([2,3,1])
```

```
Traceback (click to the left of this block for traceback)
...
IndexError: list index out of range
```

```
B=[2,3,1]; B; B[0]; [a(B[i]) for i in [0..len(B)-1]]
```

```
[2, 3, 1]
2
[1, 4, 2]
```

```
a.sign(); b.sign()
```

```
-1
-1
```

```
a.order(); b.order()
```

```
2
2
```

Figure 2: Calling methods of permutations (as elements of group)

```
S6.center() #subgroup comprised of elements that commute with every element of S6
```

```
Subgroup of (Symmetric group of order 6! as a permutation group)
generated by [()]
```

```
S6.centralizer(a) #subgroup comprised of elements that commute with a
```

```
Subgroup of (Symmetric group of order 6! as a permutation group)
generated by [(5,6), (3,4), (3,5)(4,6), (1,2), (1,5)(2,6)]
```

```
S6.is_regular(); S6.is_abelian() #various tests for properties of groups
```

```
False
False
```

```
S6.conjugacy_classes()
```

```
[Conjugacy class of () in Symmetric group of order 6! as a permutation
group, Conjugacy class of (1,2) in Symmetric group of order 6! as a
permutation group, Conjugacy class of (1,2)(3,4) in Symmetric group of
order 6! as a permutation group, Conjugacy class of (1,2)(3,4)(5,6) in
Symmetric group of order 6! as a permutation group, Conjugacy class of
(1,2,3) in Symmetric group of order 6! as a permutation group, Conjugacy
class of (1,2,3)(4,5) in Symmetric group of order 6! as a permutation
group, Conjugacy class of (1,2,3)(4,5,6) in Symmetric group of order 6!
as a permutation group, Conjugacy class of (1,2,3,4) in Symmetric group
of order 6! as a permutation group, Conjugacy class of (1,2,3,4)(5,6) in
Symmetric group of order 6! as a permutation group, Conjugacy class of
(1,2,3,4,5) in Symmetric group of order 6! as a permutation group,
Conjugacy class of (1,2,3,4,5,6) in Symmetric group of order 6! as a
permutation group]
```

Figure 3: Calling methods of the symmetric group of six elements

```
S6.orbit(2) #returns all possible values that 2 can take as a result of applying a permutation from S6
```

```
(1, 6, 2, 3, 4, 5)
```

```
S6.orbit((1,2), action='OnSets') #returns all possible values the set {1,2} can take
```

```
({1, 2}, {2, 3}, {3, 4}, {1, 3}, {4, 5}, {2, 4}, {5, 6}, {3, 5}, {1, 4},
{1, 6}, {4, 6}, {2, 5}, {2, 6}, {1, 5}, {3, 6})
```

```
S6.stabilizer(2) #subgroup comprised of elements that do not change the element 2
```

```
Subgroup of (Symmetric group of order 6! as a permutation group)
generated by [(5,6), (4,6), (3,6), (1,6)]
```

Figure 4: Calculating orbits of a point and of a set, and calculating stabilizer of a point

```
S6.subgroups()
```

```
WARNING: Output truncated!
full_output.txt


[Subgroup of (Symmetric group of order 6! as a permutation group)
generated by [()], Subgroup of (Symmetric group of order 6! as a
permutation group) generated by [(5,6)], Subgroup of (Symmetric group of
order 6! as a permutation group) generated by [(4,5)], Subgroup of
(Symmetric group of order 6! as a permutation group) generated by
[(4,6)], Subgroup of (Symmetric group of order 6! as a permutation
group) generated by [(3,4)], Subgroup of (Symmetric group of order 6! as
a permutation group) generated by [(3,5)], Subgroup of (Symmetric group
of order 6! as a permutation group) generated by [(3,6)], Subgroup of
(Symmetric group of order 6! as a permutation group) generated by
[(2,3)], Subgroup of (Symmetric group of order 6! as a permutation
group) generated by [(2,4)], Subgroup of (Symmetric group of order 6! as
a permutation group) generated by [(2,5)], Subgroup of (Symmetric group
of order 6! as a permutation group) generated by [(2,6)], Subgroup of
(Symmetric group of order 6! as a permutation group) generated by
[(1,2)], Subgroup of (Symmetric group of order 6! as a permutation
group) generated by [(1,3)], Subgroup of (Symmetric group of order 6! as
a permutation group) generated by [(1,4)], Subgroup of (Symmetric group
of order 6! as a permutation group) generated by [(1,5)], Subgroup of
(Symmetric group of order 6! as a permutation group) generated by
[(1,6)], Subgroup of (Symmetric group of order 6! as a permutation
group) generated by [(1,2)(3,4)(5,6)], Subgroup of (Symmetric group of
order 6! as a permutation group) generated by [(1,2)(3,5)(4,6)],
Subgroup of (Symmetric group of order 6! as a permutation group)
generated by [(1,2)(3,6)(4,5)], Subgroup of (Symmetric group of order 6!
as a permutation group) generated by [(1,3)(2,4)(5,6)], Subgroup of
(Symmetric group of order 6! as a permutation group) generated by
[(1,3)(2,5)(4,6)], Subgroup of (Symmetric group of order 6! as a
permutation group) generated by [(1,3)(2,6)(4,5)], Subgroup of
```

Figure 5: Truncated output of the subgroups of S_6

APPENDIX B

THE $N = 3$, $N = 4$, AND $N = 5$ RUBIK GROUPS IN SAGE

Modeling a Rubik's cube in Sage

The Rubik's Cube can be dealt with in Sage using one of two classes: `CubeGroup()` and `RubiksCube()`. `RubiksCube()` has already been demonstrated, and an example of how to use `CubeGroup()` is below.

sage: rubikscube=CubeGroup(); rubikscube; rubikscube.display2d("") # last will display the cube after no move has been applied

The Rubik's cube group with generators R,L,F,B,U,D in SymmetricGroup(48).

```
              +--------------+
              |  1    2    3 |
              |  4   top   5 |
              |  6    7    8 |
+-------------+--------------+-------------+-------------+
|  9  10  11 | 17   18   19 | 25   26  27 | 33  34  35 |
| 12 left 13 | 20  front 21 | 28 right 29 | 36 rear 37 |
| 14  15  16 | 22   23   24 | 30   31  32 | 38  39  40 |
+-------------+--------------+-------------+-------------+
              | 41   42   43 |
              | 44 bottom 45 |
              | 46   47   48 |
              +--------------+
```

sage: rubikscube.F(), rubikscube.move("F")[0] # permutation B returned in Singmaster notation, called two ways

((6,25,43,16)(7,28,42,13)(8,30,41,11)(17,19,24,22)(18,21,23,20),

(6,25,43,16)(7,28,42,13)(8,30,41,11)(17,19,24,22)(18,21,23,20))

sage: r1 = {'back': [[33, 34, 35], [36, 0, 38], [37, 39, 40]], 'down': [[41, 42, 43], [44, 0, 45], [46, 47, 48]],'front': [[17, 18, 19], [20, 0, 21], [22, 23, 24]],'left': [[9, 11, 10], [12, 0, 13], [14, 15, 16]],'right': [[29, 26, 27], [28, 0, 25], [30, 31, 32]],'up': [[1, 2, 3], [4, 0, 5], [6, 8, 7]]};

sage: rubikscube.legal(r1) #tests legality when facelets are in the described positions, where 0 is the center facelet

0

sage: rubikscube.plot3d_cube("F")

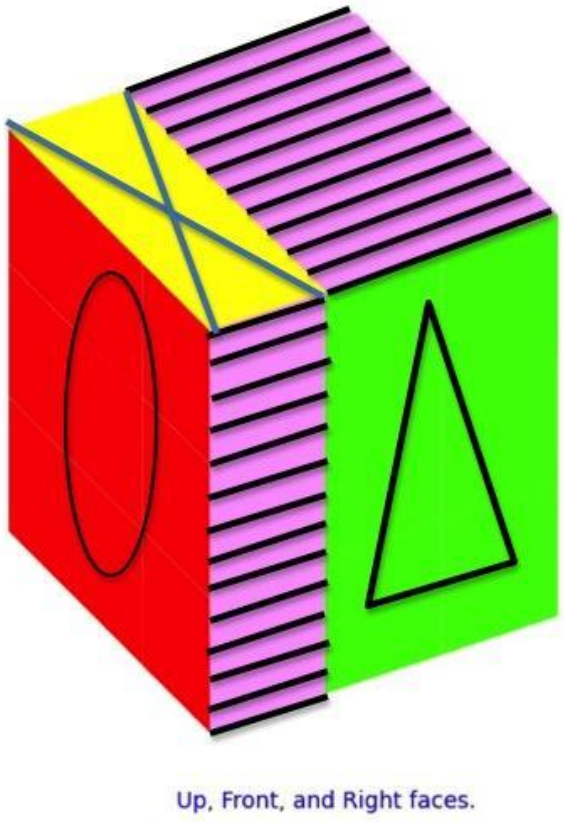


Figure 1: 3D output in Sage of a Rubik's cube with the front face turned clockwise. The colors used do not match what is commercially available, but suffice for demonstration. Here, the topmost face is "Up", the leftmost is "Front", and the last is "Right". Like colors have been given symbols for the sake of clarity.

sage: state=rubikscube.faces("R*F*B2*F^(-1)") # sets cube faces after the sequence of moves has been applied

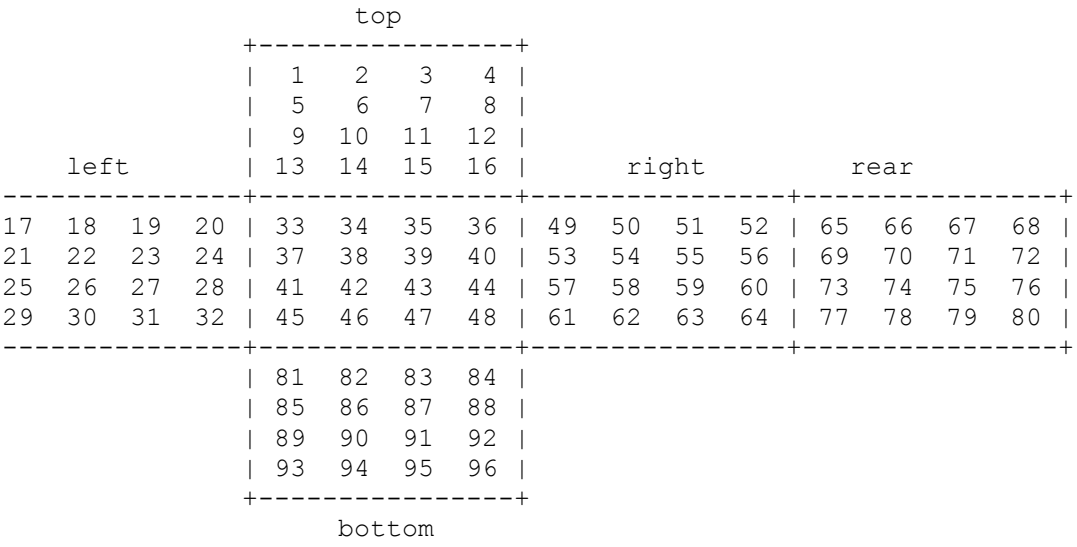
sage: rubikscube.solve(state) # outputs shortest sequence to scramble cube into 'state' which should be inverted for solution

'R B2'

4x4x4 Rubik's Cube (Rubik's Revenge)

The permutation group that would describe the $n = 4$ cube would be a subgroup of S_{96} and have twelve generators, one for each slice of the cube. The group would also have order 1697268890861823893377084924596414796040188723200000000, or 2^50*3^29*5^9*7^7*11^4*13^2*17^2*19^2*23^2. Each of the generators have been assigned a letter based on which direction on the cube they lie (standard left, right, up, down, front, back from Singmaster), and are capitalized if the slice is the outside slice, and lowercased if the inside slice. For example, the leftmost slice is denoted by L, and the slice directly inside L is denoted as l. The topmost slice is U, and the slice directly inside is u.

Facelet Numbering



Generators

L=(1,33,81,80)(5,37,85,76)(9,41,89,72)(13,45,93,68)(17,20,32,29)(18,24,31,25)(19,28,30,21)(22,23,27,26)

l=(2,34,82,79)(6,38,86,75)(10,42,90,71)(14,46,94,67)

r=(3,78,83,35)(7,74,87,39)(11,70,91,43)(15,66,95,47)

R=(4,77,84,36)(8,73,88,40)(12,69,92,44)(16,65,96,48)(49,52,64,61)(50,56,63,57)(51,60,62,53)(54,55,59,58)

U=(17,65,49,33)(18,66,50,34)(19,67,51,35)(20,68,52,36)(1,4,16,13)(2,8,15,9)(3,12,14,5)(6,7,11,10)

u=(21,69,53,37)(22,70,54,38)(23,71,55,39)(24,72,56,40)

d=(25,41,57,73)(26,42,58,74)(27,43,59,75)(28,44,60,76)

D=(29,45,61,77)(30,46,62,78)(31,47,63,79)(32,48,64,80)(81,84,96,93)(82,88,95,89)(83,92,94,85)(86,87,91,90)

F=(13,49,84,32)(14,53,83,28)(15,57,82,24)(16,61,81,20)(33,36,48,45)(34,40,47,41)(35,44,46,37)(38,39,43,42)

f=(9,50,88,31)(10,54,87,27)(11,58,86,23)(12,62,85,19)

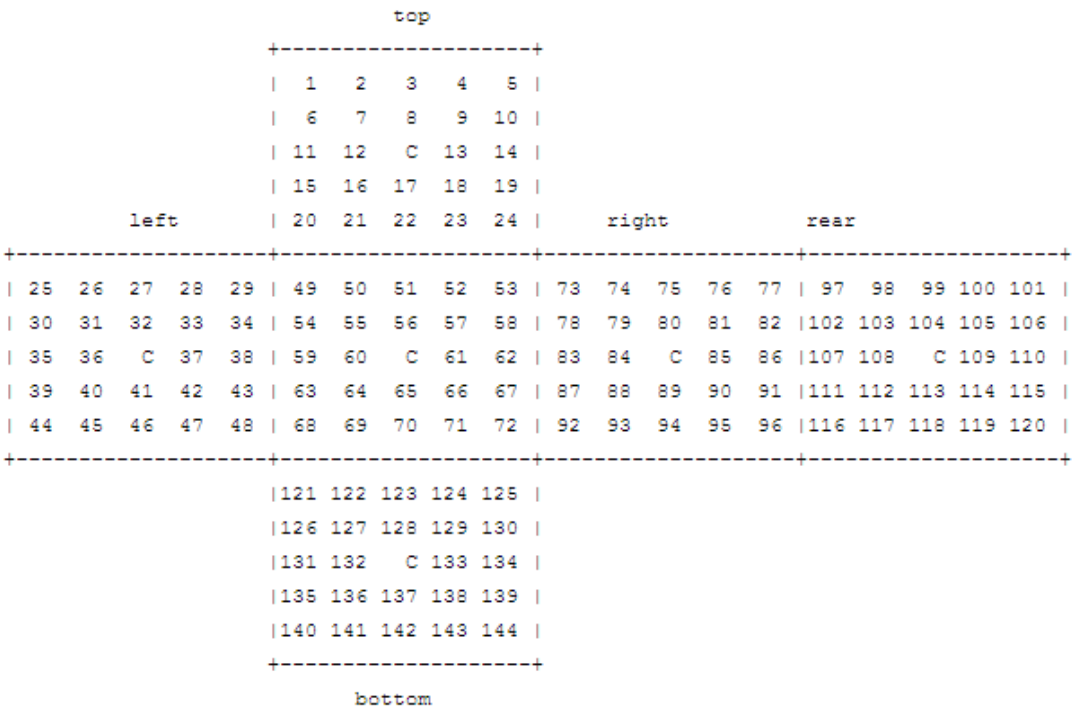
b=(5,30,92,51)(6,26,91,55)(7,22,90,59)(8,18,89,63)

B=(1,29,96,52)(2,25,95,56)(3,21,94,60)(4,17,93,64)(65,68,80,77)(66,72,79,73)(67,76,78,69)(70,71,75,74)

5x5x5 Rubik's Cube (Professor's Cube)

The permutation group that would describe the $n = 5$ cube would be a subgroup of S_{144} and have order 187482944635529732772790873889227901981546631951879544342690743355535455068451826403024481163459041820045980772498551501004551978078086768958630613903092958409450362020068065280000000000000000000000000000000000, or 2^139*3^68*5^32*7^22*11^12*13^10*17^8*19^6*23^6*29^4*31^4*37^2*41^2*43^2*47^2*53^2*59^2*61^2*67^2*71^2. Like the $n = 4$ cube, the Professor's Cube would have twelve generators, one for each slice of the cube that is not the centermost slice as any facelets on that slice can be moved by rotation of other slices. Naming of slices (and therefore the corresponding permutations) follows the same conventions as for the $n = 4$ cube, as there are only twelve slices to consider.

Facelet Numbering



Generators

L=(1,12,121,120)(6,54,126,115)(11,59,131,110)(15,63,135,106)(20,68,140,101)(25,29,48,44)(26,34,47,39)(27,38,46,35)(28,43,45,30)(31,33,42,40)(32,37,41,36)

l=(2,50,122,119)(7,55,127,114)(12,60,132,109)(16,64,136,105)(21,69,141,101)

r=(4,117,124,52)(9,112,129,57)(13,108,133,61)(18,103,138,66)(23,98,143,71)

R=(5,116,125,53)(110,111,130,58)(14,107,134,62)(19,102,139,67)(24,97,144,72)(73,77,96,92)(74,82,95,87)(75,86,94,83)(76,91,93,78)(79,81,90,88)(80,85,89,84)

U=(25,97,73,49)(26,98,74,50)(27,99,75,51)(28,100,76,52)(29,101,77,53)(1,5,24,20)(2,10,23,15)(3,14,22,11)(4,19,21,6)(7,9,18,16)(8,13,17,12)

u=(30,102,78,54)(31,103,79,55)(32,104,80,56)(33,105,81,57)(34,106,82,58)

d=(39,63,87,111)(40,64,88,112)(41,54,89,113)(42,66,90,114)(43,67,91,115)

D=(44,68,92,116)(45,69,93,117)(46,70,94,118)(47,71,95,119)(48,72,96,120)(121,125,144,140)(122,130,143,135)(123,134,142,131)(124,139,141,126)(127,129,138,136)(128,133,137,132)

F=(20,73,125,48)(21,78,124,43)(22,83,123,38)(23,87,122,34)(24,92,121,29)(49,53,72,68)(50,58,71,63)(51,62,70,59)(52,67,69,54)(55,57,66,64)(56,61,65,60)

f=(15,74,130,47)(16,79,129,42)(17,84,128,37)(18,88,127,33)(19,93,126,28)

b=(6,45,139,76)(7,40,138,81)(8,36,137,85)(9,31,136,90)(10,26,135,95)

B=(1,44,144,77)(2,39,143,81)(3,35,142,86)(4,30,141,91)(5,25,140,96)(97,101,120,116)(98,106,119,111)(99,110,118,107)(100,115,117,102)(103,105,114,112)(104,109,113,108)

APPENDIX C

SPOT IT! AND SPOT IT! JR. GAMES

Table 1: Spot It! Jr. list of symbols, corresponding number assignments, and degree

Symbol	**Point**	**Degree**
bat	0	6
bear	1	6
camel	2	6
cat	3	6
chick	4	6
crab	5	6
dog	6	6
dolphin	7	6
duck	8	6
fish	9	6
flamingo	10	6
frog	11	6
gator	12	6
gorilla	13	6
grasshopper	14	6
hippo	15	6
horse	16	6
lion	17	6
octopus	18	6
owl	19	6
parrot	20	6
penguin	21	6
pig	22	6
rabbit	23	6
seal	24	6
shark	25	6
skunk	26	6
snake	27	6
squirrel	28	6
starfish	29	6

Symbol	Point	Degree
turtle	30	6

Table 2: List of blocks for Spot It!, Jr.

Block	Points in Block
Block 0	8, 12, 16, 24, 27, 30
Block 1	2, 3, 7, 15, 17, 24
Block 2	1, 2, 9, 25, 29, 30
Block 3	0, 4, 11, 14, 15, 30
Block 4	8, 9, 15, 22, 26, 28
Block 5	7, 9, 11, 20, 21, 27
Block 6	11, 18, 19, 24, 26, 29
Block 7	0, 2, 16, 21, 23, 26
Block 8	6, 17, 19, 21, 28, 30
Block 9	5, 15, 16, 19, 20, 25
Block 10	4, 9, 13, 16, 17, 18
Block 11	8, 10, 11, 17, 23, 25
Block 12	4, 6, 7, 12, 25, 26
Block 13	3, 4, 5, 8, 21, 29
Block 14	0, 1, 7, 8, 13, 19
Block 15	7, 10, 14, 16, 28, 29
Block 16	0, 12, 17, 20, 22, 29
Block 17	0, 3, 18, 25, 27, 28
Block 18	1, 4, 20, 23, 24, 28
Block 19	2, 6, 8, 14, 18, 20
Block 20	5, 7, 18, 22, 23, 30
Block 21	6, 13, 15, 23, 27, 29
Block 22	0, 5, 6, 9, 10, 24
Block 23	2, 5, 11, 12, 13, 28
Block 24	3, 10, 13, 20, 26, 30
Block 25	1, 10, 12, 15, 18, 21

Block	Points in Block
Block 26	2, 4, 10, 19, 22, 27
Block 27	13, 14, 21, 22, 24, 25
Block 28	1, 5, 14, 17, 26, 27
Block 29	1, 3, 6, 11, 16, 22
Block 30	3, 9, 12, 14, 19, 23

Table 3: Spot It! list of symbols, corresponding number assignments, and degree

Symbol	Point	Degree
!	0	7
?	1	7
anchor	2	8
apple	3	8
art	4	8
balloon	5	8
bomb	6	8
bottle	7	8
cactus	8	7
candle	9	8
car	10	8
carrot	11	8
cat	12	8
cheese	13	8
clock	14	8
clover	15	8
clown	16	8
dog	17	7
dolphin	18	8
dragon	19	8
droplet	20	8
eye	21	7

Symbol	Point	Degree
flame	22	8
flower	23	7
ghost	24	8
hand	25	8
heart	26	8
ice cube	27	7
igloo	28	8
key	29	8
knight	30	8
ladybug	31	7
leaf	32	7
lightbulb	33	7
lightning bolt	34	8
lips	35	8
lock	36	8
moon	37	8
ok	38	8
pencil	39	8
person	40	7
pirate	41	7
scissors	42	8
snowflake	43	8
snowman	44	6
spider	45	8
splat	46	8
stop	47	7
sun	48	8
sunglasses	49	8
t-rex	50	7
target	51	8
treble	52	8

Symbol	Point	Degree
tree	53	8
web	54	8
yin yang	55	8
zebra	56	8

Table 4: List of blocks for Spot It!

Block	Points in Block
0	2, 11, 13, 14, 30, 32, 47, 54
1	31, 32, 34, 46, 52, 53, 55, 56
2	7, 11, 12, 31, 35, 40, 42, 51
3	7, 10, 13, 18, 22, 28, 44, 46
4	6, 18, 39, 40, 45, 47, 48, 52
5	0, 3, 28, 32, 37, 42, 43, 45
6	4, 12, 14, 23, 26, 33, 45, 46
7	4, 9, 11, 29, 37, 44, 48, 55
8	11, 15, 21, 22, 36, 45, 50, 56
9	6, 9, 13, 21, 23, 34, 38, 42
10	12, 13, 19, 37, 41, 49, 50, 52
11	1, 15, 18, 33, 34, 35, 37, 54
12	2, 9, 15, 25, 40, 41, 43, 46
13	14, 16, 18, 27, 36, 41, 42, 55
14	2, 3, 4, 18, 24, 31, 38, 50
15	2, 8, 19, 22, 33, 42, 48, 53
16	3, 21, 27, 30, 35, 46, 48, 49
17	9, 16, 26, 28, 35, 47, 50, 53
18	7, 9, 24, 32, 33, 36, 39, 49
19	4, 7, 8, 16, 21, 43, 52, 54
20	9, 10, 19, 20, 27, 31, 45, 54
21	3, 6, 12, 25, 36, 44, 53, 54
22	4, 17, 19, 28, 30, 34, 36, 40

Block	Points in Block
23	0, 2, 10, 23, 29, 35, 36, 52
24	0, 7, 14, 20, 25, 34, 48, 50
25	6, 8, 14, 15, 28, 29, 31, 49
26	6, 10, 30, 33, 43, 50, 51, 55
27	2, 6, 7, 17, 26, 27, 37, 56
28	1, 3, 9, 14, 17, 22, 51, 52
29	14, 19, 35, 38, 39, 43, 44, 56
30	12, 22, 24, 27, 29, 34, 43, 47
31	15, 20, 24, 26, 30, 42, 44, 52
32	3, 8, 10, 11, 26, 34, 39, 41
33	16, 22, 23, 25, 30, 31, 37, 39
34	23, 24, 28, 41, 48, 51, 54, 56
35	1, 7, 29, 30, 38, 41, 45, 53
36	0, 4, 13, 15, 27, 39, 51, 53
37	11, 17, 18, 20, 23, 43, 49, 53
38	1, 2, 12, 20, 21, 28, 39, 55
39	0, 1, 6, 11, 16, 19, 24, 46
40	10, 12, 15, 16, 17, 32, 38, 48
41	3, 13, 16, 20, 29, 33, 40, 56
42	18, 19, 21, 25, 26, 29, 32, 51
43	0, 22, 26, 38, 40, 49, 54, 55
44	1, 4, 10, 25, 42, 47, 49, 56
45	8, 20, 36, 37, 38, 46, 47, 51
46	8, 13, 17, 24, 25, 35, 45, 55
47	3, 5, 7, 15, 19, 23, 47, 55
48	5, 10, 14, 21, 24, 37, 40, 53
49	1, 5, 13, 26, 31, 36, 43, 48
50	5, 17, 29, 39, 42, 46, 50, 54
51	0, 5, 8, 9, 12, 18, 30, 56
52	2, 5, 16, 34, 44, 45, 49, 51
53	5, 11, 25, 27, 28, 33, 38, 52

Block	Points in Block
54	4, 5, 6, 20, 22, 32, 35, 41

BIBLIOGRAPHY

Beth, Thomas, Dieter Jungnickel, and Hanfred Lenz. "Block Designs and Affine and Projective Geometry."*Design Theory*. Second ed. Vol. 1. Cambridge: Cambridge UP, 1999. Print.

Dummit, David Steven, and Richard M. Foote. *Abstract Algebra*. Second ed. John Wiley and Sons, 1999. 108-109. Print.

Hughes, D. R., and F. C. Piper. *Design Theory*. Cambridge [Cambridgeshire: Cambridge UP, 1985. Print.

Mulholland, Jamie. "Permutation Puzzles: A Mathematical Perspective." *Permutation Puzzles: A Mathematical Perspective 15 Puzzle, Oval Track, Rubik's Cube and Other Mathematical Toys Lecture Notes*. 4 June 2013. Web. 25 Nov. 2014. <http://www.sfu.ca/~jtmulhol/math302/notes/302notes.pdf>.

Pinter, Charles C. *A Book of Abstract Algebra.* Second ed. New York: McGraw-Hill, 1990. Print.

"Rubik's Cube Group Functions." *Rubik's Cube Group Functions — Sage Reference Manual V6.6.beta0: Groups*. Web. 25 Feb. 2015. <http://www.sagemath.org/doc/reference/groups/sage/groups/perm_gps/cubegroup.html>.

VITA

Jessica Ruth Chowning was born on July 6, 1991 in California. She graduated valedictorian from St. Charles West High School in 2009 and attended Missouri University of Science and Technology for her undergraduate work, graduating in May 2013 with a Bachelor of Science (Summa cum Laude) in Applied Mathematics. She stayed for her graduate studies, where she earned a Master of Science in Applied Mathematics in May 2015.