



Spring 2023

Solving the Genius Square: Using Math and Computers to Analyze a Polyomino Tiling Game

Noah Jensen

Follow this and additional works at: https://cedar.wwu.edu/wwu_honors

 Part of the [Applied Mathematics Commons](#)

Recommended Citation

Jensen, Noah, "Solving the Genius Square: Using Math and Computers to Analyze a Polyomino Tiling Game" (2023). *WWU Honors College Senior Projects*. 711.
https://cedar.wwu.edu/wwu_honors/711

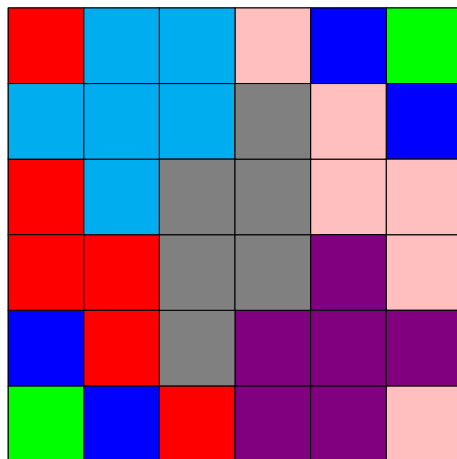
This Project is brought to you for free and open access by the WWU Graduate and Undergraduate Scholarship at Western CEDAR. It has been accepted for inclusion in WWU Honors College Senior Projects by an authorized administrator of Western CEDAR. For more information, please contact westerncedar@wwu.edu.

Application of a Mathematical Tiling Model to the Board Game Genius Square

Noah Jensen

Advised by Dr. Stephanie Treneer

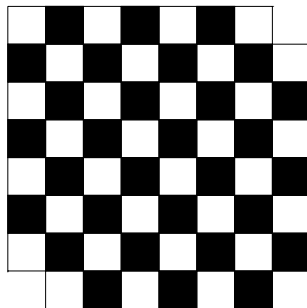
June 12, 2023



1 Introduction

Polyominoes are a generalization of dominoes [5], where instead of only two squares joined at an edge, there can be any number of squares joined along shared edges. Solomon W. Golomb coined both polyominoes and pentominoes, the special case of 5 connected squares, at a talk to the Harvard Math Club in 1953. After Golomb coined the term polyomino, he had, as he himself put it, been “irrevocably committed to their care and feeding” [3]. He also published the book “Polyominoes: Puzzles, Patterns, Problems, and Packings”, which is likely the most popular book solely focusing on polyominoes. Beyond books, he published at least 10 papers specifically about polyominoes, their properties, how to use them to tile the plane, and how to determine if they can tile the plane. His work with polyominoes helped to jump start their study in the wider community. In his Polyominoes book, Golomb noted that an ancient master of the game Go is credited as having made the observation that there are 12 pentominoes [3]. While it is very easy to connect any number of squares together to form a polyomino, there is no known formula to tell us the exact number of polyominoes of

Figure 1: An 8×8 checkerboard pattern that is missing a pair of opposite corners. This pattern cannot be tiled by dominoes.



a given square count.

In his book, Golomb mentioned many different polyomino problems and methods of attempting to solve them. The most important one of these methods that we will use is to treat your region and pieces as if they had a checkerboard pattern overlaid on them [2]. To use this method, the first thing that you do is imagine the checkerboard pattern overlaid on the region and the polyominoes. Then you count the number of black and white squares on the region and on each polyomino. If the count of black and white squares in the region and the count of black and white squares on the polyominoes do not match, then it is impossible to tile your region with those polyominoes.

Consider this example from Golomb's Polyominoes book. If you take a standard chess board (an 8×8 grid) and remove one pair of opposite corners, would it be possible to fill in the rest of the board with only dominoes? The answer is no. If we assume that the black corners are removed, then we have 30 black squares and 32 white squares remaining on the board. When you place down a domino, it must always cover exactly one black and one white square, which means no matter how many dominoes you place, an equal number of black and white squares are covered. However, since we have 32 white and only 30 black squares we cannot cover this 8×8 board. This checkerboard can be seen in Figure 1.

In his long running Mathematical Games column in the Scientific American, Martin Gardner discussed polyominoes, which he called "super dominoes" [5], several times and popularized the idea for a general audience. Among the many examples of polyominoes in popular culture, the two most prevalent currently are dominoes, made of 2 squares, and tetrominoes, which are made up of 4 squares and seen in the game Tetris. There is also a board game called Hexed made around 1978 which uses pentominoes. While these are the most familiar polyominoes, there are infinitely many more that exist as you can adjoin any number of squares to form a single polyomino.

1.1 The Genius Square

In this project, I investigated a polyomino based board games called The Genius Square. Its premise is simple, but this premise leads to many interesting mathematical questions. The object of the game is to use a variety of polyominoes to tile a 6×6 board of squares. There are 9 unique polyominoes at the player's disposal and they are comprised of 1, 2, 3, and 4 squares. These 9 polyominoes account for 29 squares of the grid, which leaves 7 squares open. This is where the blockers come in. There are 7 wooden blockers whose positions on the board are determined by rolling seven six-sided dice at the start of each game. The sides of the dice are each labeled with one board square, but the dice don't all have 6 distinct squares. A table which has all of the coordinates for the squares associated with each die can be seen in Table 1. A visual¹ representation of the squares associated with each die is on page 1, before the start of the introduction. These dice are the default ones that come with the game. Once you have placed the blockers, you then solve the game by filling in the remaining 29 squares with the polyominoes. An example of a board and one possible solution is seen in Figure 2.

Table 1: A table listing each die and the coordinates of the squares onto which each die can roll a blocker.

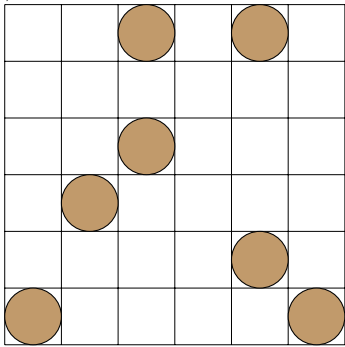
| Die | Coordinates |
|-----------|---|
| Die One | $\{(1,6),(6,1)\}$ |
| Die Two | $\{(5,1),(6,2),(1,5),(2,6)\}$ |
| Die Three | $\{(1,1),(3,1),(4,1),(4,2),(5,2),(6,3)\}$ |
| Die Four | $\{(1,4),(2,5),(3,5),(3,6),(4,6),(6,6)\}$ |
| Die Five | $\{(1,2),(1,3),(2,1),(2,2),(2,3),(3,2)\}$ |
| Die Six | $\{(4,5),(5,4),(5,5),(5,6),(6,4),(6,5)\}$ |
| Die Seven | $\{(2,4),(3,3),(3,4),(4,3),(4,4),(5,3)\}$ |

The game proclaims that there are 62,208 possible ways for the blockers to be placed and that every single one of these placements can be solved. This is an interesting claim given that there are 8,347,680 different ways to place 7 blockers on our grid, if we ignore the dice. There are also many ways to place blockers that are immediately clear as being unsolvable. For example, we can choose a pattern which isolates 2 monominoes, polyominoes of 1 square, and since we only have 1 monomino, we cannot fill in the rest of the board. This will be discussed further in a later section along with boards which can isolate 2 dominoes. Figure 3 displays one such example.

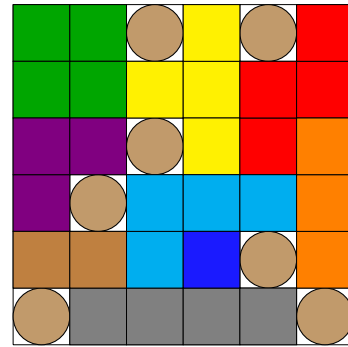
From this claim that every one of the 62,208 boards is solvable, we get to an interesting series of questions. How many of the full set of boards are solvable? Can we classify the solvable boards? Is it true that all of the boards used in the game are solvable? If that it is

¹All images with squares colored multiple ways in this paper are also on the github repository with numbers or letters helping to identify squares in the same group, they can be seen in the file "PaperImagesWithoutColor.pdf" Link: https://github.com/Noah-Jensen/GS_SeniorProject

Figure 2: Two Genius Square boards. The first demonstrates a board containing only blockers, and the second shows one possible solution to that board.

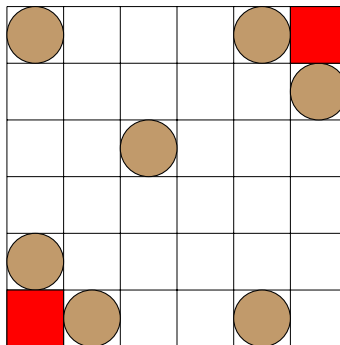


An unsolved Genius Square board, only containing blockers.



A solved Genius Square board, containing both pieces and blockers.

Figure 3: A set of blocker placements which is unsolvable with normal Genius Square pieces. The two red corners would require a monomino each, but the player can only use one during the game.



true, why so? And lastly, can we make our own set of dice, distinct from the set provided by the game, which when played can also only generate solvable boards?

When most people attempt to solve a board by hand, they use a method called backtracking in which the player places polyomino pieces until they either solve the board, or find a situation where they can no longer place pieces on the board. If they run into this issue, they remove one of their pieces and attempt to place different piece. This process repeats until the board has been solved. This method translates fairly well to programming so there are many backtracking programs that can be found on the internet for solving Genius Square boards. These programs are not necessarily the most efficient as many cannot determine if a board is unsolvable. They instead rely on a time limit, that if reached, labels the board unsolvable.

The code that we have written in the pursuit of the previous questions solves some of these issues. This code is written based on a model developed by Marcus R. Garvie and John

Burkardt [1]. This model works by taking in the desired polyominoes and every possible location they can occupy, then generating a binary linear system of equations. If there is a binary solution to this equation then the board is solvable and the solution gives the placement for all 9 polyominoes. Using our implementation of this model, we verified the following theorem.

Theorem 1.1. *Every Genius Square board possible with the default set of dice is solvable.*

1.2 Organization of the Paper

Before we can get to alternate sets of dice and further interesting findings, we must become accustomed to the material that we're working with. Section 2, the Preliminaries, will give a more thorough introduction to polyominoes and Genius Square, as well as introduce some counting tools and the symmetries of the square. It is followed by Section 3, The Garvie and Burkardt Model, which gives an in-depth explanation of the model which is used in the code developed for this project. Section 4, Implementation, describes the most important pieces of code for this project and how those pieces work. Afterwards, in Section 5, Dice Patterns, we are able to investigate several patterns that give unsolvable boards. We then can use these patterns to create a procedure to form new sets of dice. These sets are not always solvable, but are more likely to be than randomly assigning dice. With this procedure we find an alternate set of dice which always produces solvable boards. Then in Section 6, Applying Group Theory, we use properties of the group D_4 to find solutions to all 8,347,680 possible blocker placements. We then end with Section 7, Conclusion and Further Inquiry which quickly summarizes this paper and considers some questions which may be interesting for further investigation.










2 Preliminaries

Before moving on to the core parts of this project, we'll need to introduce the terminology used for polyominoes in this project. We will also mention the coordinates used in Genius Square and how they are adapted here. This is followed by a brief introduction to combinations and the multiplication principle, which are used in counting total boards in Section 6. This is followed by a quick explanation of the group D_4 .

2.1 Polyominoes

When referring to polyominoes there are some key components to keep in mind. First, the size, or order, of the polyomino is the number of squares in its makeup. The general name for a polyomino of order n is an n -omino, though there are some specific names like tetromino, referring to a 4-omino. Next, we have the shape that each polyomino takes on, for example there is a polyomino of order 3 that looks like an L, so we use the name 3-L for it. Table 2 contains all the names we use for the polyominoes that are relevant to this project.

Table 2: Table of Polyomino Categories, Names, and Shapes

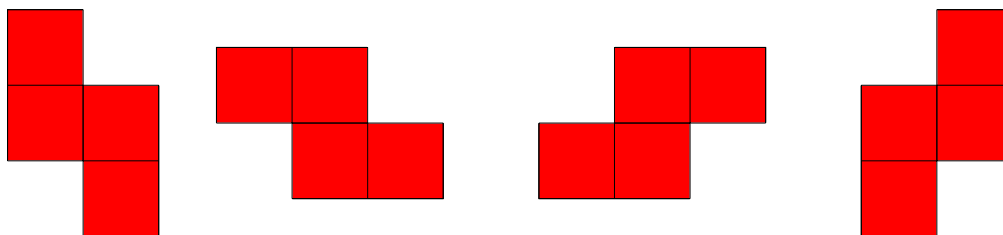
| Category | Full Name | Shorthand | Shape |
|-------------|------------------|-----------|---|
| Monomino | Monomino | 1 |  |
| Domino | Domino | 2 |  |
| Trominoes | L-Tromino | 3-L |  |
| | I-Tromino | 3-I |  |
| Tetrominoes | L-Tetromino | 4-L |  |
| | I-Tetromino | 4-I |  |
| | T-Tetromino | 4-T |  |
| | Z-Tetromino | 4-Z |  |
| | Square-Tetromino | 4-Square |  |

When enumerating polyominoes of a given order, we must consider whether the rotations or reflections of the polyominoes are distinct. To do this, we use the terms *fixed* and *free* polyominoes. When we deal with free polyominoes, we consider the reflections and rotations of a polyomino to be the same polyomino. On the other hand, when we deal with fixed polyominoes, we consider the reflections and rotations to be distinct from their progenitor and each other. In this project, we are primarily dealing with free polyominoes. Briefly returning to the introduction, I mentioned that there are 12 pentominoes when we don't consider reflection or rotation. The better way to say this is that there are 12 free pentominoes.

In Figure 4, we can see a Z-tetromino and its reflection and rotations. If we're dealing with free polyominoes, like we are for most of this paper, we consider these four to be the same. If we instead work with fixed polyominoes, as we do when writing the code in the Implementation section, we consider these to be distinct tetrominoes.

The Genius Square board is a 6×6 grid of squares, which labels the rows, top to bottom, A through F, and the columns, left to right, 1 through 6. Instead of referring to the squares

Figure 4: A picture of all 4 distinct fixed Z-tetrominoes which are the same free tetromino.



by these alphanumeric coordinates, they will be referred to in a way that is similar to the way that entries of a matrix are labeled, i.e. row 1 column 1, (1,1), is row A, column 1. The coordinates that will be used are all the integer ordered pairs of $[1, 6] \times [1, 6]$.

2.2 Counting techniques

In Section 6, we will end up counting how many boards come from particular ways to split the board into quadrants. To do this there are two tools we need to be able to use that come from enumerative combinatorics, the mathematics of counting sets. The first tool is called a combination. In his text *Applied Combinatorics*, Alan Tucker formally describes a combination as follows.

Definition (Combinations [4]). A combination of r objects from n total objects is “an unordered selection, or subset, of r objects out of the n objects”. This is calculated with the formula $\frac{n!}{r!(n-r)!}$.

Less formally, you can think of this as counting the number of ways that you can pick r objects from a set of n objects while not caring about the order in which you pick those objects. When talking about combinations we use $\binom{n}{r}$ to represent picking this count, which is read as n choose r . For example, if we wanted to find the number of ways we could deal out a 5-card hand from a 52 card deck, we would want to find $\binom{52}{5}$, which is 2,598,960 possible ways. More related to this project, an important number to consider is how many different ways can blockers be placed on the Genius Square board if we ignore the default dice. Since we want to count the number of ways to pick 7 squares on the 6×6 board, we just need to calculate $\binom{36}{7}$, which turns out to be 8,347,680 possible placements.

The next tool that we need is called the multiplication principle.

Definition (The Multiplication Principle [4]). For a sequence of events which each have a distinct number of possible outcomes, and where the number of outcomes of each event is independent from previous events, then the total number of outcomes possible is the product of the number of outcomes from each individual event.

For example, suppose we were to roll a six-sided die twice. The first time we roll it, there are 6 possible outcomes, and the second time we roll it, there are another six possible outcomes. If we wanted to count the total number of possible outcomes from rolling this die twice, we

would just multiply 6 times 6 to get 36 possible outcomes.

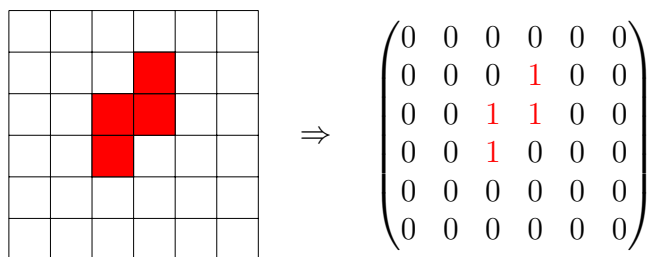
2.3 The symmetries of a square

Another important idea for section 6 is the group D_4 , which is the set that describes all symmetries of a square. Symmetries here are rotations and reflections that won't visibly alter a square. For example, if you rotate a square 90° about its center, it is indistinguishable from a square that is not rotated at all. Similarly, if you place a line that connects corners of a square diagonally across from each other, and reflect the square across that line, it appears exactly the same as if you did not do anything at all. There are 8 symmetries which compose D_4 . These 8 symmetries are the rotations of the square by 0° , 90° , 180° , and 270° , as well as horizontal, vertical, and both diagonal reflections. In this paper, we'll consider these rotations clockwise, but note that if they were considered counterclockwise, the rotations would be equivalent. When referring to these symmetries we'll use the following notation: e is 0° rotation, also called the identity, r_{90} is 90° rotation, r_{180} is 180° rotation, r_{270} is 270° rotation, v is a reflection across the vertical line bisecting the square, h is a reflection across the horizontal line bisecting the square, d is a reflection across the main diagonal, and d' is a reflection across the anti-diagonal. In Section 6, we will consider how these symmetries can be used on a Genius Square board.

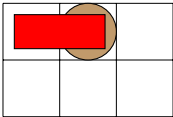
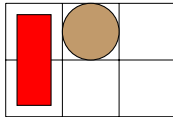
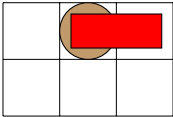
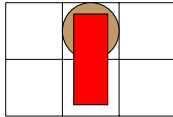
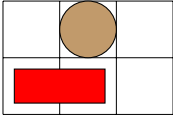
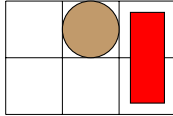
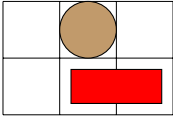
3 The Garvie and Burkardt Model

The model [1] we implemented with our code has been our primary tool for insight into the Genius Square boards. This model is able to tile finite grids using a select set of polyominoes. To do this, we wrote code which attempts to place each polyomino in every position on the board. The code then saves all valid placements of those polyominoes as distinct matrices. For the actual project, all of the matrices are 6×6 , as that is the size of the Genius Square board. More generally though, the matrices take on the size of the smallest rectangle that contains the region you attempt to tile. Each of the matrices that correspond to a placement of a polyomino will have a 1 at the coordinates which relate to where the polyomino lies in the region and a 0 everywhere else. This can be seen in Figure 5 and a more expansive example of the whole process can be seen in Example 1 with a 2×3 grid, one blocker, and a domino.

Figure 5: Example of a polyomino and the matrix corresponding to its placement.



Example 1. A small example of how the code finds valid placements of dominoes on a 2×3 grid with 1 blocker. It also shows the corresponding matrices for all placements.

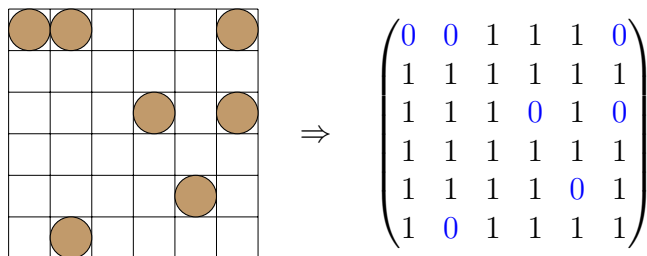
| Placements | Matrix | Validity | Placements | Matrix | Validity |
|---|--|----------|--|--|----------|
|  | $\begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ | Invalid |  | $\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$ | Valid |
|  | $\begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$ | Invalid |  | $\begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$ | Invalid |
|  | $\begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix}$ | Valid |  | $\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$ | Valid |
|  | $\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}$ | Valid | | | |

The maximum number of these placement matrices that we can have is 625. To find this number, we must count all possible placements of all orientations of each polyomino when there are no blockers on the board. To start, the monomino has 36 possible placements and only one orientation, so there are a maximum of 36 placement matrices corresponding to monominoes. For the rest, you can consider how many different placements a polyomino can have in a single orientation. For example, a Z-tetromino has 20 possible placements for a particular orientation. The reason why it is 20 is the smallest rectangle which encloses the Z-tetromino is a 2×3 rectangle. For either orientation of this rectangle, we can place it in a 6×6 board 20 different ways. Once you have the number of placements, you then multiply that number by the number of different orientations there are for the polyomino. A Z-tetromino has 4 possible orientations, as seen in Figure 4, which means that overall, a Z-tetromino has $20 \cdot 4 = 80$ possible placements when you don't consider blockers. After finding this number for every polyomino and summing them, we get 625 possible placements.

Each of the placement matrices is saved in a list which contains all of the possible placement matrices for each polyomino piece. Then, we use these placement matrices to set up a linear system in which the matrices are the weights. In the following equations P_{ij} will refer to a particular placement matrix that stores the j th placement of the i th polyomino. The variables x_{ij} of this system are binary and if a particular $x_{ij} = 1$, it tells us that the P_{ij} matrix represents a polyomino placement in the solution to the board. We let B be the 6×6 matrix which represents a given Genius Square board only containing blockers. So, we set B to be comprised of 1s except for 0s where a blocker is located. An example B is shown in Figure 6.

We can then form the following system ([1] Equation 2.4a), with n_i being the number of

Figure 6: Example of blockers and the matrix corresponding to their placement.



valid placement matrices created for a particular polyomino.

$$\sum_{i=1}^9 \sum_{j=1}^{n_i} x_{ij} P_{ij} = B \quad (3.1)$$

Since we must use only one of each type of polyomino, we include additional constraints in the form of another linear system which we will combine with the one above to get the final system which we will solve. Here, we have vectors of length 9, where each entry corresponds to one of our polyominoes. These vectors, v_{ij} , have a 1 in the i th position and zeros in all other positions. The v_{ij} vector tells us which kind of polyomino the P_{ij} placement matrix represents. We require that the sum of all of these vectors equals the vector with all ones. This creates our desired limit as for a fixed i , we must have $x_{ij} = 1$ for exactly one j . This equation, which is a modified version of equation 2.4b from [1], is:

$$\sum_{i=1}^9 \sum_{j=0}^{n_i} x_{ij} v_{ij} = \vec{1}. \quad (3.2)$$

It is important to note that the x_{ij} in equations 3.1 and 3.2 are in fact the same. However, 3.1 is a system of equations with matrix coefficients and 3.2 is a system with vector coefficients. To be able to work with these equations simultaneously, we turn the matrices P_{ij} into vectors p_{ij} . We do so by taking the (r, c) entry, and sending it to the $(r - 1) * 6 + c$ entry of the vector. This can be seen in Figure 7.

This turns our 6×6 matrix into a length 36 vector. We don't only change the P_{ij} matrices, but we also change the B matrix in the same manner. We will call this vector \vec{b} . This now gives us 2 equations with vector coefficients, which we combine to form our final linear system. The variables are the same x_{ij} that have been used previously, but now the weights are the vectors $\begin{pmatrix} p_{ij} \\ v_{ij} \end{pmatrix}$, which have a length of 45. So, we now have:

$$\sum_{i=1}^9 \sum_{j=0}^{n_i} x_{ij} \begin{pmatrix} p_{ij} \\ v_{ij} \end{pmatrix} = \begin{pmatrix} \vec{b} \\ \vec{1} \end{pmatrix}. \quad (3.3)$$

Figure 7: Demonstrating how a placement matrix will be transformed into a vector.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ \vdots \\ 35 \\ 36 \end{pmatrix}$$

Let $\sum_{i=1}^9 \sum_{j=0}^{n_i} 1 = n$. This is the total number of P_{ij} matrices that were created to use in this system.

Equation 3.3 gives us the linear system that we solve. Since each column of that linear system represents a particular placement for a polyomino, we know that there are at most 625 columns and there are 45 rows as that is the length of the weights $\begin{pmatrix} p_{ij} \\ v_{ij} \end{pmatrix}$. Since the code will only generate the valid placements for a polyomino, the actual size of the system will be $45 \times n$. After running the code, we found that the average number of columns is 300, which means the average system will be a 45×300 system. When a binary solution to this system exists, there is a way to tile our Genius Square board. The following is an example of what such a matrix equation would look like:

$$\begin{pmatrix} 1 & 0 & 0 & & 1 & 0 & & 0 & 0 & 1 \\ 0 & 1 & 0 & \dots & 1 & 1 & \dots & 0 & 1 & 1 \\ 0 & 0 & 1 & & 0 & 1 & & 1 & 1 & 1 \\ \vdots & & \ddots & & \vdots & \vdots & \ddots & & \vdots & \\ \hline 1 & 1 & 1 & & 0 & 0 & & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 1 & 1 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & & 0 & 0 & & 0 & 0 & 0 \\ \vdots & & \ddots & & \vdots & \vdots & \ddots & & \vdots & \\ 0 & 0 & 0 & \dots & 0 & 0 & \dots & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_{11} \\ x_{12} \\ x_{13} \\ \vdots \\ x_{21} \\ x_{22} \\ \vdots \\ x_{9,28} \\ x_{9,29} \\ x_{9,30} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ \vdots \\ 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \quad (3.4)$$

In the next section, I will discuss how I implemented this model using Matlab, Python, and the CPLEX linear optimization package.

4 Implementation

All the code from this project can be found at this Github page: https://github.com/Noah-Jensen/GS_SeniorProject. Any code that I did not create has been linked to in this page and the particular programs stated by name.

In order to implement the model by Garvie and Burkardt [1], it's necessary to generate matrices for all possible placements for every polyomino. It is important to make sure that these placements don't overlap on blockers. We generate all of these matrices with the `polyMatGen` function.

This function takes a vector with a given set of blocker coordinates and returns 9 lists of placement matrices, each list corresponding to one of the polyominoes in Genius Square, as well as a scalar, n , which is the sum of the lengths of these 9 lists. This code considers the placements of fixed polyominoes as opposed to free polyominoes. This means the rotations and reflections of pieces, even at the same location count as different placements are considered to be different. The code starts by creating a board matrix, which is all zeros save for the blocker coordinates which are ones. Then, for each fixed polyomino, the code tries every placement on the board that does not fall off of the edge. If there is an overlap with a blocker, that placement is thrown out.

There are two methods of checking the placements for each polyomino.

- For the monominoes, the code cycles through placements on an empty board. In a particular part of the cycle, it checks if there is a blocker at its current position. If there is, then this is the case of overlapping a blocker and the placement is thrown out. If not, a one is placed in that position, and a copy of that placement matrix is saved.
- For the non-monominoes, the code starts in the top left of the board, then starts cycling through the board. When attempting to place a polyomino, the current position in the cycle is used as the top left square in the smallest rectangle bounding the particular piece. Then, in a 6×6 matrix, the code adds one at every location in which the polyomino occupies a square. This matrix is added to the board matrix. If the sum of these two matrices contains a 2, that indicates that there is an overlap in blockers, and thus that placement is discarded. On the other hand, when there is not a 2, the placement matrix is saved.

Figure 8 displays the code that generates all the placement matrices for the monominoes. It cycles by checking every column in the first row left to right, then the second row, continuing until the sixth and final row. The first part of the if statement checks if there is not a blocker. If there is not, then the position is saved in the list of placement matrices. The second part of the if statement says that if there is a blocker where we check, we set what would have been the placement matrix to be a matrix full of ones. This matrix is saved, but is easy to filter out. Next, a tally counter is increased. This counter keeps track of how many invalid placement matrices have been found. Lastly for an invalid placement, we make note of where that matrix is located using a binary list, `MLoc`.

Figure 8: Code from the Matlab function `polyMatGen.m`. It generates and stores the possible placements of monominoes for a particular set of blockers.

```

for t = 0:35

col = mod(t,6)+1;
row = floor((t)/6+1);

if B(row, col) == 0
    Monom(row, col, t+1) = 1;

elseif B(row, col) == 1
    Monom(:, :, t+1) = 1;
    MCount = MCount + 1;
    MLoc(t+1,1) = 1;

end

end

```

Once all of these placement matrices are generated by `polyMatGen`, they are given to the `ModelMat` function, along with the vector containing the positions of the blockers and the total number of placement matrices n . This function is used to generate the $45 \times n$ matrix, seen in the matrix equation (3.4). The function then uses the vector with the coordinates of the blockers to create the vector $\begin{pmatrix} \vec{b} \\ \vec{r} \end{pmatrix}$ seen in equation (3.3).

Before the next function, it's necessary to introduce CPLEX. CPLEX is a linear optimizer developed by IBM. A linear optimizer is a program which will optimize a linear system of equations to find an optimal outcome, such as a maximum or minimum.

The function `genSqPartBoard` uses `polyMatGen` and `ModelMat` to generate a matrix and vector. Then it uses a function created by John Burkardt called `polyomino.lp.write` to generate a linear programming (.lp) file that CPLEX can read and solve. It takes in the file location where this .lp file will be stored, the matrix and vector generated by `ModelMat`, and the size of said matrix.

From here, the .lp file can be read and optimized by CPLEX. Since the Genius Square board has 62,208 possible starting positions, it is unfeasible to generate every .lp file and tell CPLEX to solve them by hand. This is where we implemented some Python code. Using the Matlab and CPLEX APIs, the Python file `FullDiceSolver` runs a series of for loops which correspond to each of the Genius Square dice. In each iteration of the for loop, it generates the relevant positions for the blockers as a vector and feeds that into `genSqPartBoard`. From

there, it calls CPLEX to solve the .lp file it just generated and if there is a solution, it saves the solution (.sol) file.

You can read the solution that CPLEX finds with the Matlab file `genSqSolRead`. It takes in the file location and the matrix generated by `ModelMat`, then spits out a list of 9 matrices, which correspond to the placements of the 9 polyomino pieces. In this code, there is another function written by John Burkardt, `cplex_solution_read`, which reads the actual solution to the matrix equation. The rest of the program, which we wrote, uses the matrix from `ModelMat` and this solution to find the polyomino placements which will be the output.

There is also code used for finding unsolvable boards by using a set of dice created by the user. The primary file used to find the boards is the Python file `CustomDice`. Instead of using the hard-coded default Genius Square dice, this program allows you to determine your own dice. The code then attempts to solve all possible blocker placements for these custom defined dice and whenever there is no solution, it saves a .txt file which contains information on the dice pattern used to solve the dice, and the particular entry in each die used to generate the unsolvable board. Within this file, the function `userDefinedDice` is run in Matlab. This function takes as an input the custom dice that the user created, and the particular entry of each die to generate a vector containing blocker coordinates.

To read the .txt files into Matlab, you use the `UnsolvableBoards` function. This function takes a folder location as an input. This folder is the one which contains all of the .txt files generated by `CustomDice`. It returns a list of 6×6 matrices which represent the blocker patterns stored in the .txt files. From this point it is possible to isolate patterns which have no clear reason to be unsolvable yet.

The Matlab functions work well and quickly run to completion. Running the function `genSqPartBoard` 1000 times produces an average run time on the order of 80 milliseconds. In a realistic scenario where it is run a single time before other programs are run, it has an average run time of about 100 milliseconds, which is fairly quick for the purposes of this project. Each time CPLEX attempts to solve an .lp file, it most often took 20 milliseconds, but varied between 10 and 30 milliseconds. When running the `FullDiceSolver` file, it took about 2 and a half hours to run all 62,208 boards to completion. This gives a solve time per board on the order of 150 milliseconds, which makes sense given that the sum of time for `genSqPartBoard` and CPLEX is about 120 milliseconds. The 30 millisecond difference between the overall average solve time for a board and the 120 millisecond sum can be accounted for in the other code that runs in between `genSqPartBoard` and CPLEX solving the board.

While 150 milliseconds per board is quick, it's not fast enough to reasonably solve every possible blocker pattern in a reasonable amount of time. To solve all 8,347,680 boards without any other time saves would take about 15 days of pure computation.

This issue is addressed in section 6. To implement this solution, we wrote a Python program called `SymmetryFullSolutions`, which takes into account the tools developed in section 6 to

solve what is equivalent to all of the boards. It does so by generating individual quadrants of a board, then piecing them together, and finally attempting to solve the board. The function developed in that program was used to run 1,521,054 unique boards, which is 18.22% of the total and it was able to find what is equivalent to 172,440 unsolvable boards out of the full 8,347,680 boards, which is about 2% of the total.

As mentioned earlier, the time component was not an issue for running the default Genius Square dice. Once `FullDiceSolver` completed its run time, I had proved Theorem 1.1.

Each board was solved and its solution saved. If one generates these files themselves, they can be read with matlab using the `genSqSolRead.m` function.

Since I saved every `.sol` file for the default Genius Square dice, the next problem to mention is storage. Each `.sol` file is about 20 kilobytes, so if we consider just the 62,208 boards possible from the official Genius Square dice, we reach over a gigabyte to store just the `.sol` files. Each `.txt` file which contains the information on an unsolvable board requires approximately 0.5 kilobytes to store. So, depending on the proportion of the 8,347,680 boards which are actually solvable, it could require between 4 gigabytes and 167 gigabytes to store everything.

So, to optimize on storage space, the instances in which `.sol` files have been saved have been when there are dice patterns which are always solvable. Here, those dice patterns are the default Genius Square dice and the alternate dice set which I found, discussed in section 5. Otherwise, the only files saved have been the `.txt` files which indicate that a particular board is unsolvable. After running `SymmetryFullSolutions`, the total space taken up by all of the `.txt` files generated is 7.87 megabytes.

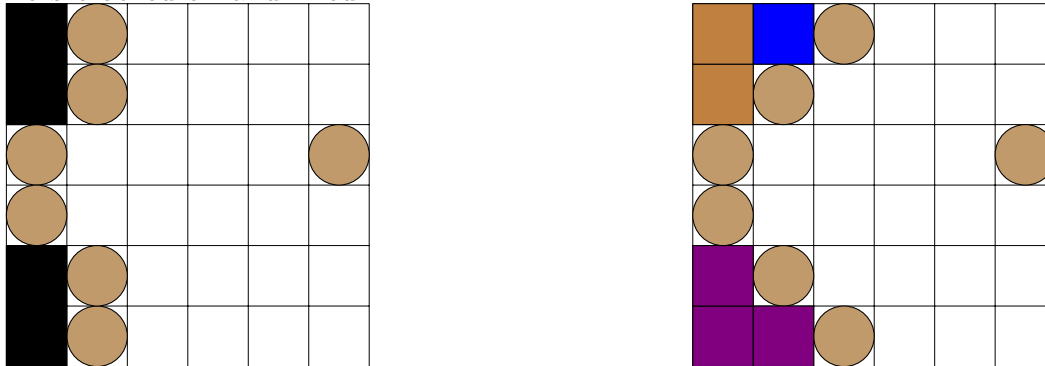
5 Dice Patterns

One of the questions I considered during the pursuit of this project is, is it possible to have a dice pattern that is distinct from the default dice pattern in Genius Square, but would still only allow for solvable boards when rolling? As will be seen in Theorem 5.3, the answer turns out to be yes. In order to start trying to make new valid dice patterns, it's best to first consider what could make a board unsolvable. In this section I will describe some classes of unsolvable boards, and then use those to develop a set of guidelines for constructing new dice sets.

We first consider if it is possible to find blocker patterns where the board necessitates multiple copies of a single polyomino. There are in fact such patterns, the simplest of which require two monominoes to solve despite the game pieces only containing one. For an example see Figure 3 which has monominoes isolated in two corners. It is also possible to have two dominoes isolated, though the blocker placements are much more restrictive than when there are two isolated monominoes. Something to note is that with only 7 blockers, it is impossible to isolate 2 polyominoes that are not monominoes or dominoes. The shapes are too large for 7 blockers to be able to confine them. The closest attempt is to try to isolate 2 L-tromino

pieces in the corners, however, this is not true isolation as you can place the L-tromino in one corner and the domino and monomino in the other. Figure 9 shows an unsolvable board with 2 dominoes and board which attempts to isolate two L-trominoes.

Figure 9: Two Genius Square boards, one with two dominoes isolated and the other with two corners blocked off and filled in.

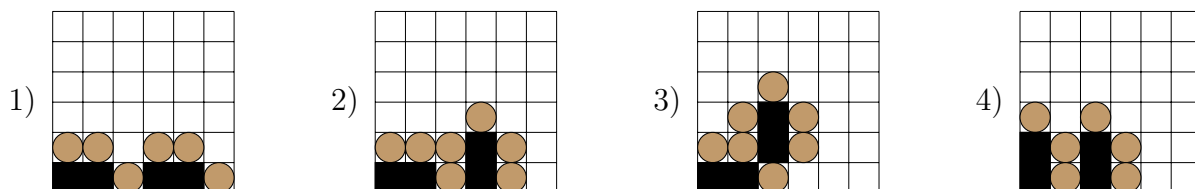


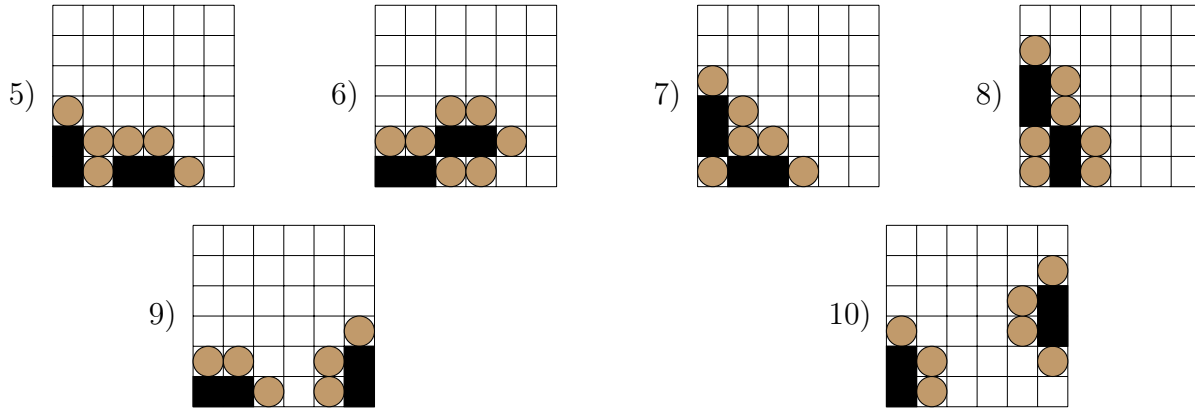
Counting the number of boards exhibiting either isolated monominoes or dominoes is non-trivial. A common counting strategy is to break things down into cases, then count the number of possibilities in each case. However, it is possible for these cases to overlap. This causes you to count the same board multiple times resulting in overshooting the actual count. Counting the number of boards with blockers that isolate two dominoes is the only one I was able to successfully count.

Theorem 5.1. *There are 1,440 different Genius Square boards which isolate two dominoes.*

Proof. To prove this, we will show all 10 cases where there are 2 dominoes isolated on the board. Then we count the number of boards in each case. Each case either does or does not share blockers between the two dominoes. There are 8 cases that share and 2 that don't. The two that don't have some freedom in the orientation and placement of each piece. For example, there is one which has a domino in 2 different corners, we are able to pick which 2 corners they are and which of 2 directions each domino faces. There are also some cases that only use 6 blockers, these cases need to account for the placement of the last blocker.

The following are all 10 cases.





We will show how to count the number of boards from two cases, the rest are all calculated similarly.

The first that we will count is 1), which has two dominoes sharing blockers and flat against the same edge. Here, we start by picking the corner that one domino is in, there are 4 choices, then picking which edge the dominoes will lie flat on, which is 2 choices for each corner. This gives 8 choices to place these two dominoes, however, we only used 6 blockers, meaning that there is 1 left to place, and there are 26 squares where we can place it. Thus, this case can make $4 \cdot 2 \cdot 26 = 208$ boards.

The second that we count is 9), which has dominoes in 2 different corners and they do not share blockers. There are $\binom{4}{2} = 6$ ways to pick the corners that we will place the dominoes in. For each domino, we have a choice of picking which of 2 edges it lies flat against. Then, since we only use 6 blockers to isolate these dominoes, there is one left that we need to place. Thus, we get a final count of $6 \cdot 2^2 \cdot 26 = 624$ boards in this case.

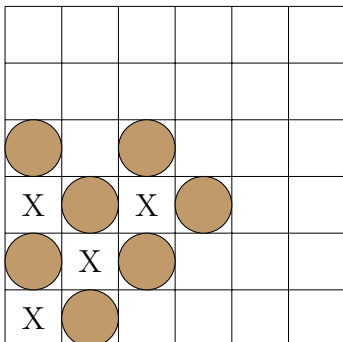
The counting for the 8 remaining cases is similar, and once completed gives a total board number of 1,440 boards. \square

The case of multiple dominoes is fairly straightforward because the number of blockers left over and the number of blockers required to force a domino to be placed are very strict criteria. Monominoes on the other hand, are significantly more complicated. It is extremely easy to overcount these boards because the requirements to isolate a monomino are much less than those for dominoes. It is possible to isolate a monomino with anywhere from 2 to 4 blockers. An example of the ease in which you can overcount isolated monominoes is seen in Figure 10.

Since it is infeasible to count all the unsolvable boards by hand, the next thing one might consider when trying to find patterns is to use the checkerboard method mentioned in the introduction. This leads to a very useful result.

Theorem 5.2. *When considering the Genius Square board using a checkerboard pattern, if every blocker is on the same checkerboard color, the board is unsolvable.*

Figure 10: This figure shows an arrangement of blockers that could be selected when isolating monominoes. If we try to count the number of ways this blocker pattern could have been produced from 2 isolated monominoes, we find 6 possible ways to do so, which overcounts multiple times.



Proof. When we place polyominoes on a board colored with a checkerboard pattern, every polyomino must cover a certain number of black or white squares. Note that in the following table, the counts are written with more black than white squares when the counts are unequal, however, the numbers of black and white squares can be swapped.

| Polyomino | Number of Black Squares | Number of White Squares |
|-----------|-------------------------|-------------------------|
| 1 | 1 | 0 |
| 2 | 1 | 1 |
| 3-L | 2 | 1 |
| 3-I | 2 | 1 |
| 4-L | 2 | 2 |
| 4-I | 2 | 2 |
| 4-Square | 2 | 2 |
| 4-T | 3 | 1 |
| 4-S | 2 | 2 |

Note that 2, 4-L, 4-I, 4-Square, and 4-Z all add the same number of black and white squares to the board. These 5 add 9 black and 9 white squares in total.

Next, we will use the fact that for 1, 3-L, 3-I, and 4-T, we can change the checkerboard color for which the polyomino covers more squares. This will let us find all the possible ways in which the polyomino pieces can cover checkerboard squares on the board. Since there are two coloring choices for each polyomino, there are 16 total color assignments, by the multiplication principle. The table below shows only 6 of these color assignments, but the remaining 10 are similarly easy to check and only repeat totals which we will see from the ones shown.

| Permutation | 1 | 3-L | 3-I | 4-T |
|-------------|-----------------|-----------------|-----------------|-----------------|
| 1 | 1 Black 0 White | 2 Black 1 White | 2 Black 1 White | 3 Black 1 White |
| 2 | 0 Black 1 White | 2 Black 1 White | 2 Black 1 White | 3 Black 1 White |
| 3 | 0 Black 1 White | 2 Black 1 White | 1 Black 2 White | 3 Black 1 White |
| 4 | 0 Black 1 White | 1 Black 2 White | 1 Black 2 White | 3 Black 1 White |
| 5 | 0 Black 1 White | 1 Black 2 White | 2 Black 1 White | 1 Black 3 White |
| 6 | 0 Black 1 White | 1 Black 2 White | 1 Black 2 White | 1 Black 3 White |

If we use this table, and the sum of black and white squares for the other 5 polyominoes, we can find the following possibilities for how many squares the polyominoes can cover.

| Permutation | Squares of Each Color Covered |
|-------------|-------------------------------|
| 1 | 17 Black 12 White |
| 2 | 16 Black 13 White |
| 3 | 15 Black 14 White |
| 4 | 14 Black 15 White |
| 5 | 13 Black 16 White |
| 6 | 12 Black 17 White |

Since the Genius Square board is a board of 6×6 squares, when we consider it with a checkerboard pattern, we notice that there are 18 black square and 18 white squares. When we subtract the number of squares of each color that we can cover with our polyominoes, the number remaining is between 1 and 6. This number is the number of blockers that we have to place on each color.

Since the maximum number of blockers which we can place on a particular checkerboard color, given the polyominoes we have, is 6, we can clearly see that it is impossible to have 7 blockers all on the same color. If there were, there would be a polyomino which would overlap with a blocker.

Therefore, if the number of blockers on a single checkerboard color is 7, the board is unsolvable.

□

Now, we can count the number of unsolvable boards that we can find due to 7 blockers being on the same checkerboard color. First if we overlay the checkerboard pattern on our board, we can consider placing 7 blockers only on black squares. When we place blockers, the order doesn't matter, only their location, and once we've placed a blocker, we cannot place another blocker on top of it. This means the number of ways that we can place 7 blockers on the black squares is $\binom{18}{7} = 31,824$ different ways. However, we can instead place the 7 blockers on all white squares, for which there are also 31,824 ways. This means the total number of ways to place 7 blockers on the same checkerboard color, and thus the number of boards that are unsolvable due to this problem, is 63,648 boards.

Using the knowledge that we can't have more than one particular polyomino on the board at any one time and that blockers can't all be on the same checkerboard color, we develop guidelines for producing new dice patterns. Our procedure below will not always create dice sets that only produce solvable boards, but it is a push in the right direction. It will disallow the boards that clearly require multiple monominoes or dominoes or those which would have all 7 blockers on the same checkerboard color.

While it is certainly possible to do this procedure without any visuals, we always used a visual aid to create dice patterns. This is done by drawing a grid of 6×6 squares, then gradually sectioning off parts of the grid and labelling to which die each section would correspond. Note that a section may cover squares which are not edge-wise connected to each other. Two requirements to keep in mind are that there must be exactly 7 sections, each linked to a particular die, and each section must contain a maximum of 6 grid squares and a minimum of 1. That is, unless you use dice which are not the standard 6-sided dice.

New Dice Procedure

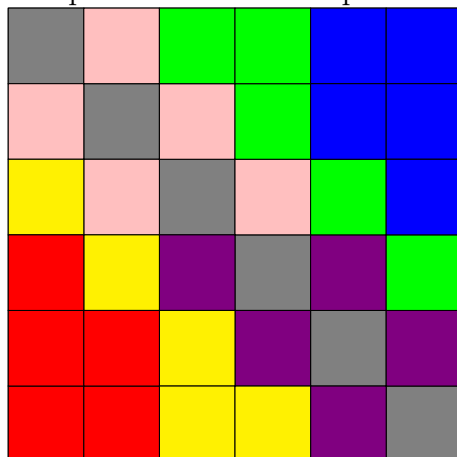
1. From the start make sure that there are 2 dice which will always place a blocker on squares of a different checkerboard color. This will immediately remove any issue of having 7 blockers on the same color.
2. Next, start assigning grid squares to sections. Keep in mind that sections have a maximum of 6 squares.
3. As you are assigning sections, or afterwards, consider each square. If a particular square has n side-adjacent squares, make sure that there are at most $n - 1$ distinct sections represented by each of those side-adjacent squares. This removes the risk of 2 isolated monominoes occurring on the board at once.
4. As you are assigning sections, or afterwards, consider pairs of side-adjacent squares. Similarly to 3., make sure that if there are n squares side-adjacent to the pair, that at most $n - 1$ of those squares come from distinct sections. This removes the risk of 2 simultaneous isolated dominoes.

The New Dice Procedure is able to ensure that the dice created will not generate unsolvable boards caused by all 7 blockers landing on the same checkerboard color, multiple isolated monominoes, or two isolated dominoes. Using the New Dice Procedure and some trial and error, I was able to find a dice pattern which is always solvable. To prove it was solvable, I ran the dice set below in the Python file `userDefinedDice`.

Theorem 5.3. *The following set of dice always generate a blocker pattern which is solvable.*

Die One is $\{(1,1),(2,2),(3,3),(4,4),(5,5),(6,6)\}$,

Figure 11: A visual expression of the dice pattern from Theorem 5.3.



Die Two is $\{(1,2),(2,1),(2,3),(3,2),(3,4)\}$,

Die Three is $\{(4,3),(4,5),(5,4),(5,6),(6,5)\}$,

Die Four is $\{(4,1),(5,1),(5,2),(6,1),(6,2)\}$,

Die Five is $\{(1,5),(1,6),(2,5),(2,6),(3,6)\}$,

Die Six is $\{(3,1),(4,2),(5,3),(6,3),(6,4)\}$,

and Die Seven is $\{(1,3),(1,4),(2,4),(3,5),(4,6)\}$.

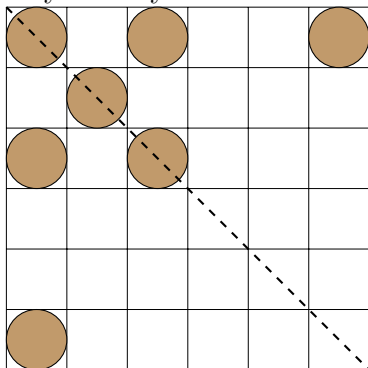
6 Applying Group Theory

Remember that there are 8,347,680 possible boards that can be made with 7 blockers. We mentioned back in Section 4 that it would take about 15 days to try and solve all these boards with our program. So, if we want to make it more feasible to solve all boards, we want to try and cut down the number of boards that actually need to be solved. To this end, recall from Section 2 the group D_4 which is the group of symmetries of a square. If we think of a board, then think about a symmetry acting on a board, for example the board being rotated 90° , any solution to the second board will be a 90° rotation of a solution to the first board.

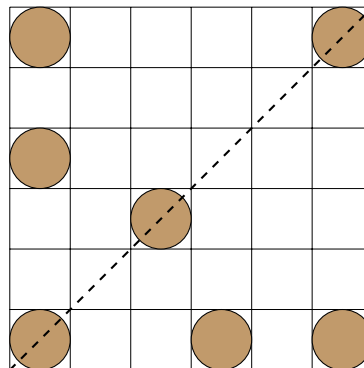
With this in mind, we can set up equivalence classes of boards. Every board of 7 blockers which is a reflection or rotation of another board will be in the same equivalence class as they can all use effectively the same solution, only differing by symmetry.

In order to use these equivalence classes to correctly speed up our solution time, we'll need to know the how many boards are in each class. At first glance, it seems like there should be 8 boards in each class because there are 8 symmetries. However, some boards are invariant under symmetry. When considering under which symmetries a board can be invariant, the

Figure 12: Examples of Genius Square boards which are invariant under diagonal or anti-diagonal symmetry.



Blocker pattern that is invariant under diagonal reflection.



Blocker pattern that is invariant under anti-diagonal reflection.

trivial case is of course the identity. If you rotate by 0° , of course the board is invariant under the symmetry. It is shown in Figure 12 that there are boards invariant under d and d' .

The following theorem shows that these three symmetries are the only ones under which a board may be invariant.

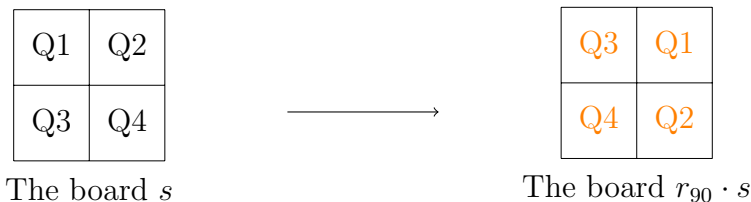
Theorem 6.1. *The only symmetries of D_4 under which a Genius Square board with 7 blockers can be invariant are $e, d,$ and d' .*

Proof. Before anything else, some notation. We will use $i \cdot s$ to represent the board s after the symmetry i has been applied to it. Those familiar with the subject may note that this is the notation for a group action.

We will now apply the symmetries of D_4 to the set of all Genius Square boards, S , to find under which symmetries a Genius Square board may be invariant. The main property we will take advantage of here is the blocker count in each quadrant of a board.

First, note that every board is invariant under e , as it is a 0° rotation.

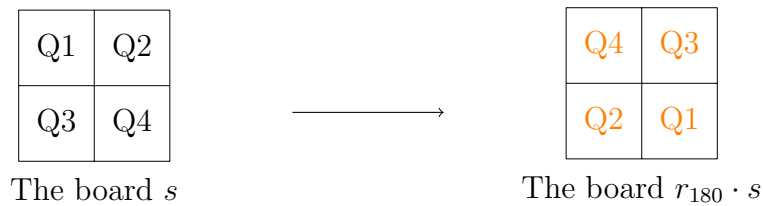
Next, consider r_{90} and let s be an element of S . For s to be invariant under r_{90} , we must have the following equality $s = r_{90} \cdot s$. These two boards, with labeled quadrants, can be seen as follows:



When applying r_{90} , the placement of the blockers within each quadrant will also rotate by 90° , but it turns out we only need to consider the number of blockers in each quadrant. Notice that the quadrant labels have rotated 90° , which means for these two boards to be equal, they must contain the same number of blockers in each quadrant. We will denote the number of blockers in quadrant X as $|X|$ from here on. So, for s to equal $r_{90} \cdot s$, we must have the following equalities: $|Q1|=|Q3|$, $|Q2|=|Q1|$, $|Q3|=|Q4|$, and $|Q4|=|Q2|$. Or equivalently, $|Q1|=|Q3|=|Q4|=|Q2|$, which means the sum of blockers in the board must be $|Q1|+|Q2|+|Q3|+|Q4|=4|Q1|$.

So, to be invariant under r_{90} , the number of blockers must be a multiple of 4. Note that r_{270} follows practically the same steps to land at the same result, so to be invariant under r_{270} , the number of blockers must be a multiple of 4.

Now, let's consider r_{180} . We see boards similar to before below:

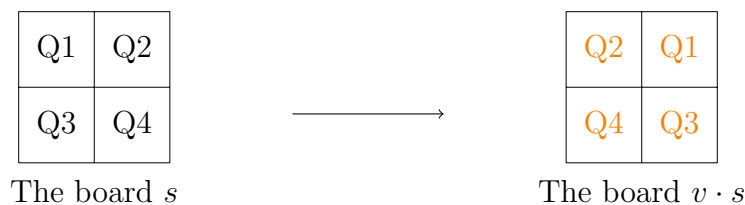


We now see that $|Q1|=|Q4|$ and $|Q2|=|Q3|$. So the total number of blockers on the board is $2|Q1|+2|Q2|$, which is an integer multiple of 2.

So, to be invariant under r_{180} , there must be an even number of blockers on the board.

Next, we will consider reflections.

First, I will cover vertical reflection. Consider v in D_4 . Then we see:



Thus, we see $|Q1|=|Q2|$ and $|Q3|=|Q4|$. So, similarly to the r_{180} , there must be an even number of blockers on the board to be invariant.

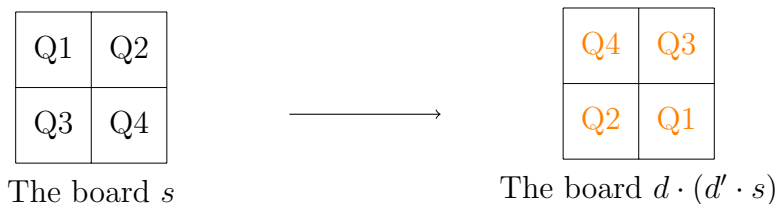
The process of working with h is practically the same as for v , so we know that it too requires an even number of blockers on the board to be invariant.

Since the only other symmetries, e , d , and d' were earlier shown to be capable of invariance, they are the only symmetries under which a Genius Square board can be invariant.

□

Corollary 6.1. *A Genius Square board cannot be invariant under both diagonals simultaneously.*

Proof. Similar to the previous proof, we consider the group of symmetries on a square, D_4 . Now, we consider the d and d' . We then see the following when both reflections are applied:



Now, we see $|Q1|=|Q3|$ and $|Q2|=|Q4|$. So the total sum of blockers must be an even number. Since a Genius Square board has 7 blockers, it cannot be invariant under both diagonals simultaneously. □

Now, we can determine the sizes of the equivalence classes.

Corollary 6.2. *An equivalence class of Genius Square boards will be of either size 4 or 8.*

Proof. There are two cases we will consider. First, we will consider an board invariant under only the identity, and second, one which is invariant under one of the diagonals.

Consider a board s that is only invariant under e . Then it's equivalence class is the set $\{e \cdot s, r_{90} \cdot s, r_{180} \cdot s, r_{270} \cdot s, h \cdot s, v \cdot s, d \cdot s, d' \cdot s\}$, which has size 8.

Next consider a board s , which is invariant under both e and d . Note that $r_{90}d = h, r_{180}d = d'$, and $r_{270}d = v$, which can be seen in the Cayley table for D_4 . Then if we consider the the previous equivalence class, we see the equivalence class is equal to $\{e \cdot s, r_{90} \cdot s, r_{180} \cdot s, r_{270} \cdot s, r_{90}d \cdot s, r_{270}d \cdot s, d \cdot s, r_{180}d \cdot s\}$, which is in turn equal to $\{s, r_{90} \cdot s, r_{180} \cdot s, r_{270} \cdot s\}$. The process is similar if you pick d' instead of d . Now, note that the size of the equivalence class is 4. □

For those familiar with group actions, this analysis of equivalence class sizes can be framed more formally in terms of orbits and stabilizers. This result that the sizes of the equivalence classes are 4 or 8 is in turn consistent with the orbit-stabilizer theorem. Figures 14 and 15 demonstrate the two examples of equivalence classes for boards. The first is the equivalence class of a board that is only invariant under the identity The second is a board invariant under diagonal reflection.

Given what we now know about the sizes of the equivalence classes and invariance under diagonal or anti-diagonal reflection, we will show that there is a method which is equivalent to attempting to solve every possible board of 7 blockers without running all 8 million boards.

A useful tool we will need is breaking boards into quadrants, then recording the number of blockers in each of the four quadrants. This produces an ordered list of four nonnegative integers which sums to 7. We will refer to these as board partitions, though it is important to note that this usage of the word partition is unrelated to the mathematical partition.

For this, the notation for a board partition will be $(Q1, Q2, Q3, Q4)$, where each component refers to the number of blockers in a particular quadrant. The labeling of quadrants is seen in Figure 13.

Figure 13: A representation of the Genius Square board as quadrants, with quadrants labeled. Note that this labeling is different from how it is presented in the proof of theorem 6.1.

| | |
|----|----|
| Q1 | Q2 |
| Q4 | Q3 |

So, the board partition $(3,2,0,2)$ would refer to a board with 3 blockers in the top-left quadrant, 2 in the top-right, 0 in the bottom-right, and 2 in the bottom-left.

Since we have a set of equivalence classes among boards, we can cut down on the number that we need to attempt to solve in order to run all 8,347,680 boards as we only need to solve one board from each equivalence class. If there were no boards invariant under a non-identity symmetry, we would be able to divide the number of boards necessary to run by 8, as the size of the equivalence class is 8. However, for those invariant under a diagonal reflection, we would only be able to divide the number of boards necessary to run by 4 as the equivalence class has size 4.

We will now consider all board partitions. We will only consider boards with the largest of the four blocker counts in the first quadrant as boards with all other board partitions are equivalent to one of these under rotation. Normally, the total number of boards that a particular board partition can generate is 8 times the size of that board partition.

However, note that when a board partition $(Q1, Q2, Q3, Q4)$ has either $Q1=Q3$ or $Q2=Q4$, the total number of boards that board partition can produce under symmetries is only 4 times the size of that board partition. This is because these cases will contain pairs of boards that are equivalent under diagonal symmetries or individual boards that are invariant under diagonal symmetries.

Now that we have our board partitions and we know how to count the number of boards each one can generate, we are able to effectively decrease the number of boards necessary

Figure 14: An example of an equivalence class of boards in which the boards are only invariant under the identity.

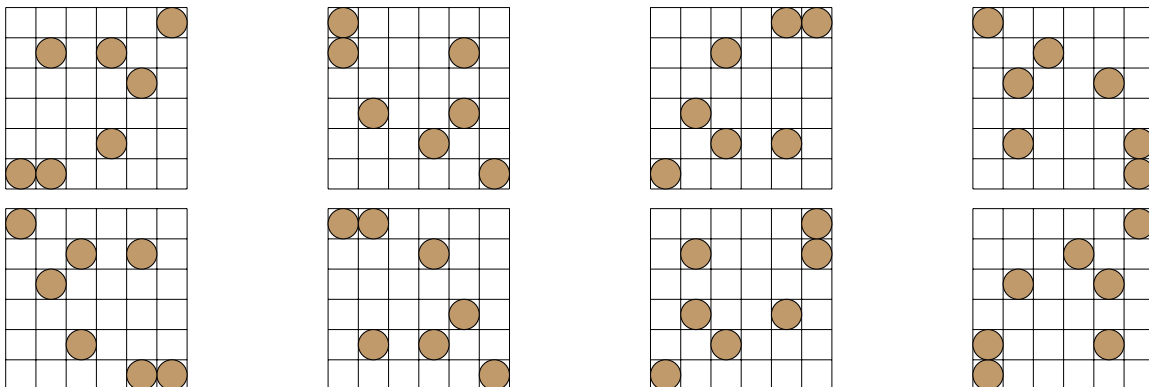
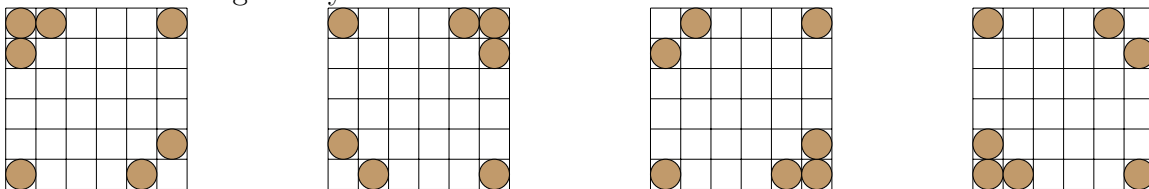


Figure 15: An example of an equivalence class of boards in which each board is invariant under one of the diagonal symmetries.



to run in order to find solutions to every board. Table 3 displays every board partition, the number of boards that board partition contains, the number which we need to multiply the count of board partitions by to get the total count of boards, and the total number of unsolvable boards for which that board partition can account.

If you sum the number of boards contained in each board partition you reach a total of 1,521,054 boards, which is a bit more than the number of distinct equivalence classes, so there are still some redundant boards. However, this is a large decrease in the number of boards necessary to run, which is a huge time save, and it still produces accurate results for every board. If you sum the total number of boards produced by each board partition, you reach the correct total of 8,347,680 boards. We were able to write a program which implemented these savings, so we only ran about 18.22% of all boards and still received a result about all boards.

In the Table 3, the entries of the second column are calculated by finding the number of ways to place blockers in the first quadrant, then multiplying by the number of ways to place blockers in the second, then third, and finally fourth quadrants. We find this number by using combinations to choose a subset from the 9 cells in a given quadrant in which to place blockers.

Consider, for example, the board partition (6,1,0,0). There are $\binom{9}{6}$ ways to place blockers in the first quadrant, $\binom{9}{1}$ in the second, $\binom{9}{0}$ in the third, and $\binom{9}{0}$ in the fourth. So, we get the

Table 3: A table containing board partitions, the number of boards a particular board partition can generate not accounting for symmetry, the multiple needed to calculate the total number of boards, the total number of boards the board partition can generate when taking symmetry into account, and the total number of unsolvable boards found for each board partition.

| Partition | Number of Boards | Multiple | Total Boards | Unsolvable Boards |
|-----------|------------------|----------|--------------|-------------------|
| (7,0,0,0) | 36 | 4 | 144 | 8 |
| (6,1,0,0) | 756 | 8 | 6,048 | 528 |
| (6,0,1,0) | 756 | 4 | 3,024 | 216 |
| (5,2,0,0) | 4,536 | 8 | 36,288 | 2,752 |
| (5,0,2,0) | 4,536 | 4 | 18,144 | 4,536 |
| (5,1,1,0) | 10,206 | 8 | 81,648 | 2,620 |
| (5,1,0,1) | 10,206 | 4 | 40,824 | 992 |
| (4,3,0,0) | 10,584 | 8 | 84,672 | 5,216 |
| (4,0,3,0) | 10,584 | 4 | 42,336 | 11,048 |
| (4,2,1,0) | 40,824 | 8 | 326,592 | 12,032 |
| (4,2,0,1) | 40,824 | 8 | 326,592 | 9,644 |
| (4,0,2,1) | 40,824 | 8 | 326,592 | 1,592 |
| (4,1,1,1) | 91,854 | 4 | 367,416 | 9,416 |
| (3,3,1,0) | 63,504 | 8 | 508,032 | 14,256 |
| (3,1,3,0) | 63,504 | 4 | 254,016 | 15,992 |
| (3,2,2,0) | 108,864 | 8 | 870,912 | 30,600 |
| (3,2,0,2) | 108,864 | 4 | 435,456 | 8,892 |
| (3,2,1,1) | 244,944 | 8 | 1,959,522 | 5,208 |
| (3,1,2,1) | 244,944 | 4 | 979,766 | 14,672 |
| (2,2,2,1) | 419,904 | 4 | 1,679,616 | 22,220 |

product $\binom{9}{6} \cdot \binom{9}{1} \cdot \binom{9}{0} \cdot \binom{9}{0} = 756$.

From there, the fourth column for each row is calculated by multiplying that row's entries in its second and third columns. Continuing with the example of (6,1,0,0), we multiply 756 and 8, to receive 6,048 total boards. And the fifth column is calculated by counting the number of unsolvable boards found for each board partition, then multiplying that number by the third column. When we add up all of the unsolvable boards from the fifth column, we get the following result.

Theorem 6.2. *Exactly 172,440 out of the 8,347,680 total Genius Square boards are unsolvable.*

7 Conclusion and Further Inquiry

Throughout this paper, we have analyzed Genius Square and shown that any dice roll from the default set of dice is always solvable. Along with this fact, we've demonstrated that there exist other sets of dice which can always produce a solvable board. We have also demonstrated two criteria which always produce unsolvable boards. The last major result that was proved is the possibility to use properties of symmetries on the Genius Square board to significantly reduce the number of boards that need to be run by a program in order to check whether every possible board of 7 blockers is solvable.

We were even able to write a program based on those properties to calculate the total number of unsolvable boards, which accounts for approximately 2% of the total 8,347,680 boards.

For the future, this result will be able to be used to describe the likelihood of creating a dice set which can generate unsolvable boards. Alongside finding the proportion of unsolvable boards, we plan on investigating further criteria which cause boards to be unsolvable beyond isolated pieces and blockers all being on the same checkerboard color.

Other topics of inquiry which may be interesting to further investigate include creation of an algorithm to generate always solvable dice patterns, rather than the simplistic procedure presented in the Dice Patterns section. Another topic to pursue is investigating the number of distinct solutions any particular board may have. During this project there was a minimal amount of time spent on this, but it was possible to find blocker positions that had a number of solutions potentially in the range of 10,000.

References

- [1] Marcus R. Garvie and John Burkardt. "A new mathematical model for tiling finite regions of the plane with polyominoes". In: *Contrib. Discrete Math.* 15.2 (2020), pp. 95–131.
- [2] Solomon W. Golomb. "Chapter 1 Polyominoes and Checkerboards". In: *Polyominoes: Puzzles, Patterns, Problems, and Packings - Revised and Expanded Second Edition*. 2nd ed. Princeton University Press, pp. 3–11. ISBN: 9780691024448.
- [3] Solomon W. Golomb. "Preface to the First Edition". In: *Polyominoes: Puzzles, Patterns, Problems, and Packings - Revised and Expanded Second Edition*. 2nd. Princeton University Press, pp. xi–2. ISBN: 9780691024448.
- [4] Alan Tucker. "Chapter 5 General Counting Methods for Arrangements and Selections". In: *Applied Combinatorics (6th edition)*. 6th ed. John Wiley & Sons, Inc., pp. 180 & 190. ISBN: 9780470458389.
- [5] Eric W. Weisstein. *Polyomino*. From *MathWorld*—A Wolfram Web Resource. URL: <https://mathworld.wolfram.com/Polyomino.html>.