

Department of Computer Science



Submitted in part fulfilment for the degree of BSc.

Solving the Genius Square: Using Functional Programming for Polyomino Tiling

Elzbieta Goralczyk

May 2024

Supervisor: Simon Foster

Acknowledgements

I would like to thank my supervisor, Simon Foster, for his support during the completion of this project.

I would also like to thank my family for their relentless encouragement, and for pushing me to finish my degree. Their motivational words went a long way.

Contents

Executive Summary	vii
1 Introduction	1
2 Background	3
2.1 What is the Genius Square?	3
2.2 Combinatorial Geometry	4
2.2.1 Polyominoes	4
2.2.2 Parity Example	5
2.3 Existing Polyomino Tiling Algorithms	6
2.3.1 Genetic Algorithms	6
2.3.2 The Garvie-Burkardt Model	7
2.3.3 Dancing Links	7
2.4 Matrix Libraries in Haskell	8
2.4.1 HMatrix	8
2.4.2 LAoP	8
2.4.3 Data.Matrix	9
3 Methodology & Implementation	10
3.1 Success Criteria	10
3.2 Model Setup	11
3.3 Implementing Garvie-Burkardt	12
3.3.1 Creating the linear system	12
3.3.2 Direct Solution	16
3.3.3 Linear Optimiser Solution	16
3.4 Labelled Matrix Algorithm	17
3.4.1 Labelling the matrices	17
3.4.2 Recursive filtering	18
3.5 User Interface	18
3.6 Hint System	19
3.6.1 Usage	19
4 Testing	21
4.1 Unit Testing	21
4.2 Solving the Example Board	22
4.3 Testing Solve Times	22

Contents

5	Analysis	24
5.1	Evaluation of Success Criteria	24
5.2	Time Test Evaluation	25
5.3	Potential Improvements	25
5.3.1	Graphical User Interface	26
5.3.2	Hint System Development	26
5.3.3	Formal Proof	27
6	Conclusion	28
A	Appendix	29
A.1	Polyomino Implementation	29
A.2	Labelled Matrix Validity Checker	30
A.3	User Interface	31
A.4	Using the User Interface	32
A.5	Placement Function	36
A.6	Direct Implementation	37
A.7	Linear Optimiser Implementation	38

List of Figures

2.1	The Genius Square Board divided into dice zones	3
2.2	The "mutilated" 8x8 checkerboard.	5
3.1	Tiling of \mathcal{R} with blockers B1, C3, D2, F1, F2, E6 and F6 . .	11
3.2	A possible M derived from \mathcal{R} and the set \mathcal{G}	15

List of Tables

2.1	Polyominoes used in the Genius Square Puzzle Game. . . .	4
3.1	Success Criteria Defined	10
4.1	Solution Times of Randomly Generated Boards	23
5.1	Success Criteria Evaluated	24

Executive Summary

The rising interest in computational puzzle-solving techniques has, within the last 60 years, raised some compelling questions which have produced some even more compelling answers. Combinatorial Geometry stands out among mathematical disciplines for its innate familiarity, often encountered early on through games like Tetris [1].

One such puzzle that has accumulated attention is the Genius Square [2]. Beyond its role as a stimulating puzzle, the mathematical principles underlying solutions to the polyomino tiling problem within the Genius Square offer practical applications, such as in transportation logistics and to streamline production processes. The aim of the project is to implement a solver for any potential Genius Square board using Haskell [3] [4].

The motivation for accomplishing this is to explore whether the mathematical strength of functional programming allows for a viable method of solving the Genius Square. This was done with the following steps:

1. Conduct a background review
2. Establish an appropriate methodology
3. Establish appropriate success criteria
4. Implement the model
5. Test the model
6. Analyse the model's functionality
7. Conduct an investigation of potential improvements

The implementation was constructed using an agile methodology [5] [6] [7] in order to complement the uncertainty of the chosen technique and whether or not it was the most appropriate for building the solver. A modular design approach [8] [9] was therefore utilised, and a board generator was created for testing purposes throughout the development process.

The design underwent continuous evolution and refinement, adapting to emerging insights and requirements. An appropriate model was eventually implemented which fulfilled the basic needs of the project, in terms of functionality and speed.

The implementation, while functional, lacked some extended features necessary for complete user satisfaction. Out of the 12 success criteria

Executive Summary

defined, 7 were completely met, and 3 were partially met. Two criteria were not met at all due to time constraints.

A vigorous analysis of the design was conducted, resulting in an extensive review of steps that could be taken to improve the model.

It was concluded that the utilisation of functional programming to solve polyomino tiling problems could be comparable to more dynamic approaches already in place, and the optimisation of the achieved model is a worthy pursuit.

1 Introduction

Of all the fields of mathematical study, Combinatorial Geometry is perhaps the most intuitive; we encounter it at a young age, through games such as Tetris [1]. Surprisingly, however, the study of it has only been a recent endeavour, with the first significant breakthroughs taking place in the 20th century. Nevertheless, since its beginning researchers have slowly begun to realise its importance; at a lecture for the Courant Institute in 1990, mathematician Israil Gelfand stated *"The older I get, the more I believe that at the bottom of most deep mathematical problems there is a combinatorial problem"* [10].

In recent years, there has been a surge of interest in computational puzzle-solving techniques, driven by the advancements in algorithm design and the increasing availability of powerful computing resources. One puzzle that has captured the attention of mathematicians and logic game enthusiasts alike is the Genius Square [2], due to its deceptively simple rules yet complex spatial constraints. This project endeavours to develop a computational solver for the Genius Square, as its blend of geometric constraints and combinatorial complexity makes it an ideal testbed for exploring the capabilities of modern computational methods.

In addition to its intrinsic appeal as a challenging puzzle, the underlying mathematical principles of the solution to the Genius Square problem have applications beyond mere entertainment. One example of this is transportation logistics, when irregularly shaped objects need to be fitted into a shipping container, while minimising wasted space and maximising efficiency. Furthermore, the ability to efficiently arrange components within a limited space in manufacturing is crucial for the optimisation of the production process and helps to reduce energy waste; the techniques we use for this project may be extended to inform strategies for layout design in factories, which will reduce costs and increase productivity. Perhaps a more obscure usage for this system lies within the realm of computational biology and drug discovery; the problem of fitting molecular structures together in 3-dimensional space shares similarities with the challenges of spatial puzzle-solving.

By considering these applications, it becomes increasingly evident that this project will not only serve as a platform for exploring the fundamental

1 Introduction

principles of mathematics and computer science; the insights we gain by solving puzzles like the Genius Square algorithmically could help us to solve problems in the real world.

Utilising the functional programming language Haskell [3] [4] will allow for a novel and efficient solution to this challenge. Its mathematical rigour makes it a natural choice for this project, offering the potential for concise, elegant and highly maintainable code. It would also allow me to experiment with functional programming paradigms within combinatorial geometry.

The objectives for this project can be described as follows:

1. Conduct a literature review and look into existing approaches for solving the Genius Square.
3. Conduct a review of state-of-the-art computational techniques.
4. Develop a computational solver for the Genius Square using Haskell.
5. Implement a user-friendly interface and hint system into the solver.
6. Validate the solver's performance through extensive testing.
7. Conduct a thorough analysis of my implementation.

These objectives aim to analyse computational puzzle-solving techniques, explore the capabilities of Haskell in geometric contexts and provide practical solutions to the Genius Square puzzle.

2 Background

In this chapter, I will first introduce the Genius Square and its rules (Section 2.1). I will then look into some ideas within the study of combinatorial geometry, and how they are connected to the Genius Square (Section 2.2). Next, I will propose some models that have previously been used to solve similar puzzles, and evaluate which of these would best fit my purposes (Section 2.3). Section 2.4 assesses matrix tools specific to Haskell.

2.1 What is the Genius Square?

The Genius Square is a logical puzzle first created by Algerian mathematician Salim Berghiche in 2018. It was redesigned into a board game for two players and manufactured by The Happy Puzzle Company [2]. The premise is simple: At the beginning of the puzzle, 7 dice are thrown which correspond to different squares in a 6x6 grid. Single-square "blockers" are placed on those points and the player must then fit a series of shapes around these blockers, filling up the rest of the space. Each die has between 2 and 6 possible points in can land on, and of course these cannot overlap. A representation of these zones is given in Figure 2.1, with different colours corresponding to different dice.

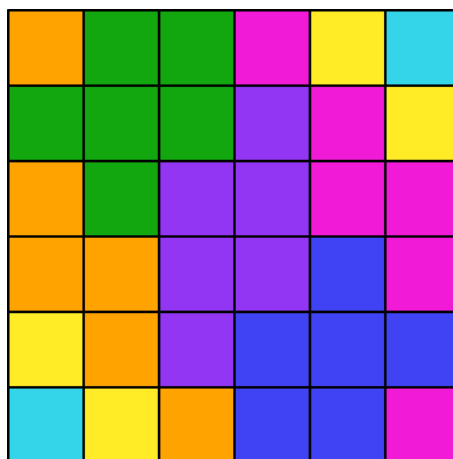











Figure 2.1: The Genius Square Board divided into dice zones

2 Background

The shapes that are used to cover the rest of the space are called "Polyominoes", which we will further explore in Section 2.2.1. The board is tiled with exactly one of each type of "free polyomino" of at most order (area) 4. This means that these polyominoes may be reflected and rotated before being placed, creating the additional feature of "instances" of a given polyomino. The names, orders, pictures and number of instances of each of these is presented in Table 2.1.

Table 2.1: Polyominoes used in the Genius Square Puzzle Game.

Number of Squares	Name	Picture	Instances
1	P_1 : Monomino		1
2	P_2 : Domino		2
3	P_3 : Straight Tromino		2
3	P_4 : Right Tromino		4
4	P_5 : Straight Tetromino		2
4	P_6 : Square Tetromino		1
4	P_7 : T Tetromino		4
4	P_8 : L Tetromino		8
4	P_9 : Skew Tetromino		4

The remarkable feature of this puzzle is that although there are 62,208 different boards that can be created this way, it has been proven that there is always at least one solution for every board [11].

2.2 Combinatorial Geometry

2.2.1 Polyominoes

The word "Polyomino", first coined by Solomon W. Goulomb during a talk for the Harvard Mathematics Club in 1953, is the name for a shape made of

2 Background

squares which are joined together at the edges. There are a few every-day cases of these: for instance, dominoes are 2 squares joined together.

In his book, *"Polyominoes: Puzzles, Patterns, Problems, and Packings"* [12], Goulomb gives a detailed analysis of Polyominoes and their place in the world of combinatorial geometry, which seems to be a neglected branch of mathematics due to its lack of systematic methods, especially with problems that require shapes to be fitted together in an efficient fashion.

For the purposes of the Genius Square puzzle, we will look at all free polyominoes up to and including an order of 4, as depicted in Table 2.1.

2.2.2 Parity Example

One of the many fascinating ideas first introduced in Goulomb's book, [12], and mentioned many times in academic papers since then, is the idea of using "parity" to check whether an area can be filled by a set of polyominoes.

His famous example includes an 8x8 checkerboard with two of its opposite corners cut off. In this case these are two black squares in the top left and bottom right corners, as shown below.

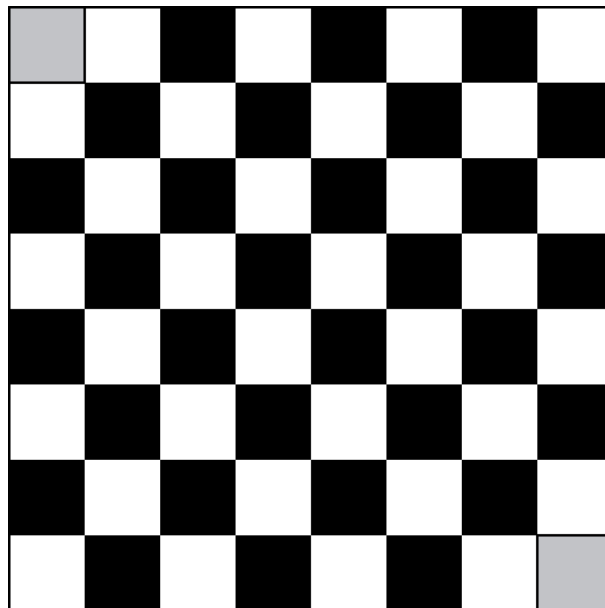


Figure 2.2: The "mutilated" 8x8 checkerboard.

The question is, could you tile this region using only dominoes? The answer is no. While previously this could be shown using a "brute-force"-style approach, requiring more computational time than was feasible, a new

way of thinking around the problem was introduced.

Dominoes have something called an "even" parity; if you were to place a domino on a checkerboard, one of the squares would always be on a white cell, and the other would always lie on black, and thus it always includes the same number of black and white tiles. However, our new checkerboard has 32 white tiles and 30 black ones. Since a domino can only cover two tiles at any given time, and it is impossible that one domino covers 2 white tiles, it follows that tiling this plane with dominoes is also impossible.

This idea is just one of many which demonstrate the uniqueness of combinatorial geometry and how it differs from other disciplines within mathematics.

2.3 Existing Polyomino Tiling Algorithms

In this section I will explore different methods for tiling polyominoes which already exist, conduct an evaluation of each one and decide which is best for my purposes.

2.3.1 Genetic Algorithms

In a paper by B.H. Gwee and M.H. Lim, *"Polyominoes Tiling by a Genetic Algorithm"* [13], the authors suggest that coding a configuration of polyominoes as a permutation string would be more suitable when solving a tiling problem. Their definition of a permutation string is "A coding structure whereby all the values assigned to the string are unique in order to enforce constraints imposed on the problem".

For instance, The string structure for x could be written as $x = \langle \lambda_1 \lambda_2 \dots \lambda_m \rangle$, where $0 \leq \lambda_i \leq m - 1$ and $\lambda_i \neq \lambda_j$ for $i \neq j$.

The model uses a coding structure called "circular placement" to tile the plane. The algorithm first selects one corner of the board and tiles it with one polyomino from the set. It then moves to the next corner (perhaps just anticlockwise of the first) and tiles another polyomino, then continues in this way until a successful tiling has been achieved (i.e. all polyominoes have been placed and there is no overlap between them). The algorithm backtracks if it is unsuccessful and tries again, perhaps using a different polyomino during the previous move of a different instance of the same polyomino. Therefore it is guaranteed that if a solution exists, the algorithm will (eventually) find it.

While this is a valid method for our problem, this "brute-force"-style approach would take a lot of computational resources and therefore may not be the most efficient way to compute a tiling of the board.

2.3.2 The Garvie-Burkardt Model

Another paper suggests that tiling a plane can be thought of as an algebraic problem. The Garvie-Burkardt Model, described by their 2019 paper [14], describes how to tile finite regions of a plane with Polyominoes by converting it first into a linear algebra system. The model also allows for "holes" in the plane, which is useful to us as these can reflect the blockers of our puzzle, as well as the space already filled.

Firstly, the algorithm generates a list of matrices representing every placement of every instance of every polyomino possible for a given board. Next, it converts each of these into a column vector and places them side by side to make a large matrix, with space at the bottom for specific constraints. After that, it reduces the large matrix into row echelon form and uses it to find a solution to the problem.

Unlike previous backtracking and other exhaustive methods, this model describes a highly systematic approach. This technique gives room for constraints, a feature which goes well with the rules of the puzzle game. Moreover, its emphasis on abstraction and modularity make it particularly well-suited to an implementation in Haskell, making this model the best selection for this project.

2.3.3 Dancing Links

There exists an effective way of solving a polyomino tiling, which utilises the features of dynamic programming. Knuth's Dancing Links technique [15] is an algorithmic approach primarily used to efficiently solve the Exact Cover problem [16] [17], particularly in the context of solving Sudoku puzzles and other combinatorial problems.

The Dancing Links technique, utilised by Algorithm X (which is also described in Knuth's paper), is based on the idea of representing a set of constraints as a matrix, where each row corresponds to a constraint, and each column corresponds to an element that satisfies one or more constraints. The goal is to find a subset of rows such that each column is covered exactly once. This is done via a depth-first backtracking algorithm, which is efficient as it uses doubly linked lists to simulate both "covering"

and "uncovering" a region of a plane.

This algorithm was utilised in a Polyomino Solver created by Chase Meadors [18]. It will serve as a comparison of the functionality of the solver, in order to evaluate functional vs dynamic approaches to polyomino tiling problems.

2.4 Matrix Libraries in Haskell

It has by now become clear that matrix manipulation is very important for this project, and I will therefore need to decide the tools found within Haskell that will allow me to do this in the most straightforward way. This section is about finding the best of these tools and establishing which would be best to use.

2.4.1 HMatrix

HMatrix [19] [20] is widely used within several publications, however, there are some necessary methods which are missing from the ones available. For instance, it does not allow for easy extension of smaller matrices into ones with specific larger dimensions.

This would be problematic due to the nature of the project, as being able to place a smaller matrix onto a larger matrix is crucial. Even though I could do this using a less efficient method, it would become rather tedious when having to generate thousands of matrices, and would take too much computational time than feasible.

2.4.2 LAoP

Another tool that Haskell has to offer is the Linear Algebra of Programming library, or LAoP [21] [22]. It is unique because of its representation of matrices as an inductive data type, which gives the matrices a more calculational and elegant style as opposed to the more traditional "vector of vectors" representation. This allows for a simple and more effective manipulation of submatrices than other libraries, which could prove useful during the generation of placement matrices.

However, this also means that it would be more difficult to convert the matrices into column vectors, which would be a significant obstacle in

the implementation of the Garvie-Burkardt model. Therefore, it is also unsuitable for this project.

2.4.3 Data.Matrix

Data.Matrix [23] has a large range of operations to be used, and its built-in methods have generic types where possible. This makes it very good for my purposes, as it involves less restriction and makes it easier to manipulate matrices.

For instance, the "extendTo" feature would be useful for comparing specific placements and their submatrices against a given board while avoiding type and logical errors. Also, The "fromLists" and "toLists" features make it easy to convert between matrix and vector format.

Furthermore, it has a built-in function to calculate the row reduced echelon form of a matrix. Even though this would produce a matrix with a "Double" type rather than the preferable "Integer", I believe that this library is more suited for the purposes of this project than the others and therefore the one I have opted to use.

3 Methodology & Implementation

This chapter goes through the implementation and methodology of my model, starting with defining a set of success criteria (Section 3.1).

I opted to use an agile methodology [5] [6] [7] to work through this project, as this would allow me to efficiently and flexibly respond to any issues and easily change my implementation. To aid this, I decided on a modular (or step-by-step) approach [8] [9] to implement my code, as this works well with the functional nature of Haskell and it allow me to change functions as needed, without affecting the rest of the code.

3.1 Success Criteria

In order to keep on track, a set of success criteria was defined as a guideline for the implementation. These are shown in Table 3.1 below.

Table 3.1: Success Criteria Defined

Marker	Criterion
C1	The solver must include an effective way to represent each polyomino
C2	The solver must include an effective way to represent each potential game board
C3	The solver must include an effective way of generating potential boards
C4	The solver could check that there are indeed 62208 possibilities
C5	The solver must be able to find at least one solution to any possible board
C6	The solver must include a user interface that allows a player to input coordinates corresponding to a set of blockers and then returns at least one possible solution
C7	The user interface should include an option to generate a board randomly
C8	The user interface must be easy to use

C9	The solver could find all possible solutions of a board
C10	The solution board must be clearly presented
C11	The user interface could include an interactive board for a user to try and solve it themselves
C12	The user interface could include a hint system to give the user a "nudge" if they get stuck

3.2 Model Setup

Our aim is to solve for a "tiling" of any appropriate region, where a tiling corresponds to the following goals being met:

- Every cell of the region is covered exactly once
- Every polyomino in the set \mathcal{P} is used exactly once

Here, \mathcal{P} represents the set of "free" (able to be rotated and reflected) polyominoes that we will be using to tile the region: $\mathcal{P} := \{P_i\}_{i=1}^{n_s}$. Note that $n_s = 9$, as we are using 9 types of polyominoes. Each P_i is described in our polyomino table 2.1.

We will represent the tiling region \mathcal{R} as a 6x6 binary matrix with a 1 in every place with a blocker and a 0 in every unused space. As a running example for the rest of this section, let's use the board that is generated with the blockers B1, C3, D2, F1, F2, E6 and F6:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 3.1: Tiling of \mathcal{R} with blockers B1, C3, D2, F1, F2, E6 and F6

The order of \mathcal{R} , labelled $c_{\mathcal{R}}$ is 29. This will be the same at the beginning of each puzzle (as there are exactly 7 blockers). We will also describe the order of each polyomino in the following set: $\mathcal{C} := \{c_1 = 1, c_2 = 2, c_3 = 3, c_4 = 3, c_5 = 4, c_6 = 4, c_7 = 4, c_8 = 4, c_9 = 4\}$.

Thus, $\sum_{i=1}^{n_s} c_i = c_{\mathcal{R}}$.

Before going any further, I decided to create a generator for any board so that I could continuously test my functions as I created them. To do this, I implemented each individual die as a list of pairs to represent the coordinates they may land on, and then created a function that would return a list of all possible boards by placing each coordinate in one set with all coordinates in the other sets:

```
die1 = [(4,5), (5,4), (5,5), (5,6), (6,4), (6,5)]
...
die7 = [(1,5), (2,6), (5,1), (6,2)]
allDice = [die1, die2, die3, die4, die5, die6, die7]
```

```
allBoards :: Num a => [[(Int,Int)]] -> [Matrix a]
allBoards allDice = [matrix 6 6 $ \(i,j) ->
  if (i,j) `elem` [d1,d2,d3,d4,d5,d6,d7] then 1 else 0
  | d1 <- die1, d2 <- die2, d3 <- die3, d4 <- die4,
    d5 <- die5, d6 <- die6, d7 <- die7]
```

I then measured the length of this list, and discovered that there are indeed 62,208 possible boards, as claimed.

The idea is that I would input a random number x between 1 and 62,208 and the function would generate the x^{th} board in the list, which is very sufficient for our testing purposes.

3.3 Implementing Garvie-Burkardt

We now have the foundation to begin the implementation of the Garvie-Burkardt Model, which we introduced in Section 2.3.2.

3.3.1 Creating the linear system

First of all we introduce the set Series i , given by:

$$\text{Series } i = \{A^{i,j} \in \{0,1\}^{r \times c} \mid j = 1, \dots, s_i\}, i = 1, \dots, n_s$$

where each s_i represents the number of ways P_i fits into \mathcal{R} (freely). We will need a way to calculate this for a given set of blockers, and use this to generate this set of placement matrices, where each $A_{i,j}$ represents the j^{th}

3 Methodology & Implementation

placement of the polyomino i . There are $n = \sum_{i=1}^{n_s} s_i$ of these. This will be done by checking each polyomino positioning against the board, and using the fact that $1 + 1 = 2$ to rule out any invalid placements (i.e: any board with a 2 on it would imply an overlap and would therefore be ruled out).

A small example of this is as follows: Take a small subregion of \mathcal{R} , represented by the 3x4 binary matrix \mathcal{R}_1 :

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

To do this, we will represent each P_i as a matrix of its "rectangular hull", i.e. the smallest rectangle containing the given shape. for instance, the corresponding matrix for one instance of the "Right Tromino", or P_4 , is:

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

We will try to find all possible ways P_4 can fit into \mathcal{R}_1 , including any rotations/reflections, and without intercepting any blockers. We will, as before, place a 1 in each filled space and a 0 otherwise. We will then add this rectangular hull representation to each submatrix of \mathcal{R}_1 of those same dimensions. We will use the Data.Matrix [23] datatype and some of its corresponding operations to achieve this. For each submatrix, we will check to see if the board is valid and if so, we will add it to a list of valid placements for that polyomino. I used the functions below to achieve this:

```
ins :: Num a => Matrix a -> Matrix a -> Int -> Int ->
    Matrix a
ins rMat pMat 0 0 = extendTo 0 (nrows pMat) (ncols rMat)
    pMat
ins rMat pMat d 0 = (zero (nrows rMat) d) <|> (extendTo
    0 (nrows rMat) (ncols rMat - d) pMat)
ins rMat pMat 0 e = (zero e (ncols rMat)) <-> (extendTo
    0 (nrows pMat) (ncols rMat) pMat)
ins rMat pMat d e = (zero e (ncols rMat)) <-> ((zero (
    nrows rMat - e) d) <|> (extendTo 0 (nrows rMat - e) (
    ncols rMat - d) pMat))
```

```
potAGen :: (Num a, Eq a) => Matrix a -> Matrix a -> [
    Matrix a]
potAGen rMat pMat = [elementwise (+) rMat (extendTo 0 6
    6 exp)| exp <- [ins rMat pMat d e| e <- [0..(abs((
    nrows rMat) - (nrows pMat))]], d <- [0..(abs((ncols
    rMat) - (ncols pMat))]]]]]
```

3 Methodology & Implementation

```
setnotcontains :: (Num a, Eq a) => a -> Matrix a -> Bool
setnotcontains a potA = if elem a potA then False else
  True
```

```
seriesi :: (Num a, Eq a) => [Matrix a] -> Matrix a -> [
  Matrix a]
seriesi piall rMat = [elementwise (-) p rMat | p <- (
  filter (setnotcontains 2) (concat[potAGen rMat pMat |
  pMat <- piall]))]
```

Here, `piall` would refer to list of rectangular hulls of all instances of a single polyomino. These come from the list `unusedPi`, which can be viewed in Section A.1 in the appendix. The outputs were as follows:

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ \mathbf{1} & 1 & 1 & 0 \\ 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 & 0 \\ \mathbf{1} & 0 & 1 & 1 \\ 0 & 0 & \mathbf{1} & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 & 0 \\ \mathbf{1} & 1 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 & 1 \\ \mathbf{1} & 0 & 1 & 0 \\ 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \\
\begin{pmatrix} 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & 1 & 0 \\ 0 & 1 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 & 0 \\ \mathbf{1} & 1 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 & 0 \\ \mathbf{1} & 0 & 1 & 0 \\ 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 & 1 \\ \mathbf{1} & 0 & 0 & 1 \\ 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \\
\begin{pmatrix} 0 & 0 & 1 & 0 \\ \mathbf{1} & 1 & 1 & 0 \\ 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 & 0 \\ \mathbf{1} & 1 & 1 & 0 \\ 0 & 0 & \mathbf{1} & 0 \\ 1 & 1 & \mathbf{1} & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 & 1 \\ \mathbf{1} & 0 & 1 & 1 \\ 0 & 0 & \mathbf{1} & 0 \\ 1 & 1 & \mathbf{1} & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & 0 & 0 \\ 1 & 1 & \mathbf{1} & 0 \\ 1 & 1 & \mathbf{1} & 0 \end{pmatrix}.$$

For ease of visibility, I have made the 1's representing the blockers bold. We can now see that in terms of this subregion, $s_4 = 12$.

Applying this method to all polyominoes and the region \mathcal{R} , we get:

$\mathcal{S} := \{s_1 = 29, s_2 = 41, s_3 = 27, s_4 = 58, s_5 = 15, s_6 = 12, s_7 = 39, s_8 = 78, s_9 = 40\}$. So in our example, $n = 339$.

Each $A^{i,j}$ matrix will have 1's only on the places occupied by the polyomino, and not the blockers. Each matrix is then written row-wise and placed side-by-side, so that each column represents one placement of one instance of one polyomino. In this way, the first 29 rows will represent each of the spaces on the board.

We will also add 9 rows to the bottom of the matrix (one for each polyomino) to tell the system which polyomino each placement column is representing. Since we are only using one instance of each polyomino, the overall matrix will continue to be binary.

For our example region \mathcal{R} , M could have the following form:

3 Methodology & Implementation

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 1 & 0 & \dots & 1 & 0 & 0 \\ 0 & 1 & 0 & \dots & 1 & 1 & \dots & 1 & 1 & 0 \\ 0 & 0 & 1 & \dots & 0 & 1 & \dots & 0 & 1 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 1 & 1 & 1 & \dots & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & 0 & 0 & \dots & 1 & 1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 & \dots & 1 & 1 & 1 \end{pmatrix}$$

Figure 3.2: A possible M derived from \mathcal{R} and the set \mathcal{G}

In general, the matrix M will have dimensions $m \times n$, where $m = c_{\mathcal{R}} + n_s$ (the number of unoccupied cells in \mathcal{R} plus the number of polyominoes we're using) and n is the amount of placement matrices in Series i . For instance, the M for our example region would have dimensions 38×339 .

We create a column vector b that will represent the "end goal" of our system. This will be comprised of 38 1's: the first 29 represent the need for every cell in the region to be tiled once and the last 9 represent the need to use every polyomino exactly once. We hence have an equation in place:

$$M\alpha = b$$

Another way to think about this approach is described in Knuth's paper on the Dancing Links algorithm [15]: We can think of the columns as elements of a universe, and the rows as subsets of the universe; then the problem is to cover the universe with disjoint subsets. So given a matrix of 0s and 1s, does it have a set of columns containing exactly one 1 in each row?

In this way, the first 29 rows of M correspond to the idea that the polyominoes cannot overlap, i.e. if one cell is occupied by a 1 then there cannot be another 1 in that row of its potential solution set. The last 9 rows ensure that all polyominoes are used exactly once, i.e. once one placement matrix has been taken from a series, no remaining element of that solution set will be taken from that series.

When a binary vector α is found, it is necessary to find all the columns of M which correspond to every nonzero position of α . There will be 9 of these columns, and together they represent a successful tiling of R . These columns would be converted back into 6×6 matrices, labelled in some way and returned to the user as a solution.

3.3.2 Direct Solution

Firstly, I attempted to implement a direct solution to the linear system. I used the `rref` function from `Data.Matrix`, which uses Gaussian elimination to find the row reduced echelon form of a matrix. This also puts the matrix in Double form (rather than Integer). I then created my own functions to put this matrix into upper triangular form and then solve the system by back substitution.

The developed code can be found in Section A.6 of the appendix. It could then be used with another function to check for a binary solution and output it. However, I found this to take up a lot of computational power and time and it was not feasible to use this method.

3.3.3 Linear Optimiser Solution

I then found a Haskell package called `Math.LinearEquationSolver` [24]. It uses the `Data.SBV` [25] [26] library, which in turn uses the `Z3` Prover [27] [28] developed by Microsoft to solve various algebraic systems.

The code I used to link my implementation to the library is shown in Section A.7 of the appendix.

The linear equation solver can do 2 things for either Integer or Rational types: Either give one solution for a system (if one exists) or give a list of x solutions, where x is specified in case there are an infinite amount of solutions. However, the solution to our system **must** be binary. I added a filter to only give me the binary vectors from the list of x solutions, however, after testing several boards I came across 2 big issues:

- Most of these boards don't have a binary vector in the first 3000 solutions
- It takes a lot of time to find even those first 3000 solutions

Of course, I have no way of knowing which solution would be the first binary one. Therefore, this is not a good method to solve this problem.

3.4 Labelled Matrix Algorithm

Even though converting the problem into a linear system isn't feasible, the Garvie-Burkardt model does introduce some beneficial concepts. If we are able to generate hundreds of matrices to compute Series i for each polyomino almost instantly, perhaps we could create additional quick methods which take each layer and add them hierarchically, so that the constraints of each addition are dependent on the previous layers.

For instance, the first level of this method could be between Series 1 and 2, i.e. the monominoes and the dominoes. The algorithm would add each matrix in Series 1 to each matrix in Series 2 elementwise. It would then filter these new matrices and keep the ones where the monomino and the domino don't overlap, and then use this set in the next level, i.e. add them to Series 3.

When all the layers have been added, we would be left with a set of matrices that show every polyomino exactly once, without and overlap. In fact, it would output *all* such matrices for *any* set of blockers.

3.4.1 Labelling the matrices

The first step to achieving this comes from the necessity of "labelling" each Series i , otherwise we'd just end up with a set of solution boards with 0's in the blocker spaces and 1's otherwise, with no other information.

To do this, I decided to use each Series' polyomino number (i) to indicate the placement of each polyomino in every $A^{i,j}$, rather than the previous '1'. For our small example, the first solution matrix in Section 3.3.1,

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ \mathbf{1} & 1 & 1 & 0 \\ 0 & 0 & \mathbf{1} & 0 \end{pmatrix}$$

would be changed to

$$\begin{pmatrix} 0 & 4 & 0 & 0 \\ \mathbf{0} & 4 & 4 & 0 \\ 0 & 0 & \mathbf{0} & 0 \end{pmatrix}$$

The following function does just that:

```
labelSeries :: (Num a, Eq a) =>
  Matrix a -> [[Matrix a]] -> [[Matrix a]]
labelSeries rMat piall = [[elementwise (*) s
  (matrix 6 6 $ \(i,j) -> (fromIntegral pn))
  | s <- (seriesi (piall !! (pn - 1)) rMat)]
  | pn <- [1..9]]
```

3.4.2 Recursive filtering

Next, we have to create some functions that would recursively add the different layers together and filter through them. The same logic can be used as with creating the original Series; we can add all the matrices of Series x and y , where $y > x$, and then filter out each matrix that contains any element greater than y . This will only work if we start with Series 1 and gradually work our way up to 9.

My implementation of this idea is as follows:

```
together :: (Num a, Eq a) =>
  Matrix a -> [[Matrix a]] -> [[Matrix a]]
together rMat unusedPi = recTog 0 labelled
  where
    labelled = labelSeries rMat unusedPi
```

```
recTog :: (Num a, Eq a) =>
  Int -> [[Matrix a]] -> [[Matrix a]]
recTog 8 sets = sets
recTog h sets = recTog (h+1) ((filter (snc h)
  [elementwise (+) s1 s2
  | s1 <- sets !! 0, s2 <- sets !! 1]): (drop 2 sets))
```

In this function, `snc` (presented in Section A.2 of the appendix) is a procedure which checks for unsuitable elements for any layer of elementwise addition and returns a Bool value.

3.5 User Interface

In order to interact with a user, I used a series of IO Monads to pull their responses from the terminal. In this way, the user can indicate which

coordinates they would like the blockers to be placed on, and the set of all solution boards would be promptly generated. There are also instructions on how to "read" the solution boards, i.e. which number corresponds to the label of which polyomino.

I also opted to give a user the option of generating a board based on an inputted number in between 1 and 62,208 (using the functions described in Section 3.2). This is easier than writing in every coordinate, which will make testing quicker.

The code used to achieve this, as well as some example interactions, are shown in Sections A.3 and A.4 of the appendix.

3.6 Hint System

An extension of this project could be to develop a hint system, in which a user can input the parts of the board they've already solved and the solver could return either the same board with one more polyomino tiled or a message that tells the user it's not a possible tiling.

In order to achieve this using the code already written, some adjustments have to be made to existing functions. For instance, a list of integers must be added as an extra constraint to tell functions such as `together` and `labelSeries` which polyominoes should be ignored.

Unfortunately, due to time constraints an implementation of this system could not be achieved. However, a thorough description of a method that could be used is given in the rest of this section.

3.6.1 Usage

The system would be used in the following way:

1. The user would enter their board, and which polyominoes they've already placed on it.
2. The system would ask which spaces the polyominoes are placed in, and use the `placePlaced` function (shown in Section A.5 of the appendix) recursively to reproduce the user's board.
3. The system would then generate one solution to the board, if it exists. Otherwise, it would let the user know that there isn't a solution and

3 Methodology & Implementation

that they should try again.

4. The system would then ask the user which polyomino they would like to see tiled.
5. The system would then filter through the previously generated solution board elementwise and output a matrix which only contains the previously tiled polyomino numbers, plus the new one.

4 Testing

Testing is a critical aspect of the development process, ensuring that the implemented code works as expected and meets the specified demands. In this chapter I will describe the testing procedure of my implementation.

4.1 Unit Testing

Due to the lack of a proven database of all solutions, it was challenging to develop a specific test to check the correctness of my implementation (other than just looking at the solutions). However, due to the modular approach I opted to use, some of the foundational functions could be checked using a unit test approach.

In order to test the functions `ins`, `potAGen`, `setnotcontains` and `seriesi` (introduced in Section 3.3.1), 3 boards were generated and n was manually calculated for each of them. They were then tested against these results, as shown with the code below. The test returned `True`.

```
testSeriesi :: Bool
testSeriesi = length(concat[seriesi p reexample
                           | p <- unusedPi]) == 339
               && length(concat[seriesi p reexample2
                           | p <- unusedPi]) == 290
               && length(concat[seriesi p reexample3
                           | p <- unusedPi]) == 255
```

```
reexample = fromLists [[0,0,0,0,0,0],[1,0,0,0,0,0],
                      [0,0,1,0,0,0],[0,1,0,0,0,0],[0,0,0,0,0,1],
                      [1,1,0,0,0,1]] -- n = 338
reexample2 = fromLists [[0,0,0,0,0,0],[0,1,0,0,0,1],
                      [1,0,0,1,0,1],[0,0,0,0,1,0],[0,0,0,0,0,0],
                      [1,0,0,0,0,0]] -- n = 290
reexample3 = fromLists [[0,1,0,0,0,1],[0,0,0,1,0,0],
                      [0,0,0,0,1,0],[0,1,0,0,1,0],[0,0,0,0,0,0],
                      [0,1,0,0,0,0]] -- n = 256
```

4.2 Solving the Example Board

Using the labelled matrix method to generate all the solution boards for our example board (Figure 3.1), we immediately get some outputs:

$$\begin{pmatrix} 1 & 2 & 2 & 3 & 3 & 3 \\ 0 & 9 & 5 & 5 & 5 & 5 \\ 9 & 9 & 0 & 7 & 4 & 4 \\ 9 & 0 & 7 & 7 & 7 & 4 \\ 8 & 8 & 8 & 6 & 6 & 0 \\ 0 & 0 & 8 & 6 & 6 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 2 & 3 & 3 & 3 \\ 0 & 9 & 7 & 7 & 7 & 4 \\ 9 & 9 & 0 & 7 & 4 & 4 \\ 9 & 0 & 5 & 5 & 5 & 5 \\ 8 & 8 & 8 & 6 & 6 & 0 \\ 0 & 0 & 8 & 6 & 6 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 2 & 3 & 3 & 3 \\ 0 & 9 & 5 & 5 & 5 & 5 \\ 9 & 9 & 0 & 7 & 6 & 6 \\ 9 & 0 & 7 & 7 & 6 & 6 \\ 8 & 8 & 8 & 7 & 4 & 0 \\ 0 & 0 & 8 & 4 & 4 & 0 \end{pmatrix}, \dots$$

As we can see, these are valid solutions. After approximately 47 minutes of running the algorithm, all of the 4238 possible solutions were found.

In comparison, when using the full Garvie-Burkardt model with the linear optimiser, it took around 13 hours of runtime to find a binary solution to the linear system (in the first 8000 solutions it produced), which would then correspond to one correct solution board.

4.3 Testing Solve Times

The limitations of the linearly optimised Garvie-Burkardt model, as described in Section 3.3.3, make it infeasible to be tested against the Labelled Matrix model. This is because it takes a lot more time and it's not guaranteed that an input of a maximum number of solutions will produce a valid output.

Instead, I used a random number generator to test 11 more boards and utilised System.Timelt [29] to check how long it would take to generate the first solution, and thus check the feasibility of the model. I also included the time it took to find all possible solutions, in order to gain insight on the wider implications of this model. The results are presented in Table 4.1 below.

4 Testing

Table 4.1: Solution Times of Randomly Generated Boards

Board Number	Total number of solutions	Time to get first solution	Time to get all solutions
542	1656	2.49s	1959.97s (\approx 33 min)
26880	812	0.81s	1546.50s (\approx 26 min)
57615	1342	2.91s	1897.80s (\approx 32 min)
15176	1005	1.04s	1539.72s (\approx 26 min)
3020	1045	1.12s	2736.95s (\approx 46 min)
40112	556	0.56s	1217.98s (\approx 20 min)
32213	868	1.92s	1393.61s (\approx 23 min)
836	1676	1.15s	3881.37s (\approx 65 min)
9115	2203	2.26s	4070.87s (\approx 68 min)
38417	2017	0.21s	2527.90s (\approx 42 min)
55125	4275	2.03s	7465.68s (\approx 124 min)

5 Analysis

This chapter includes a comprehensive analysis of the project's performance against its predefined success criteria. The primary objective is to evaluate the extent to which these criteria have been fulfilled and to identify areas for potential improvement.

5.1 Evaluation of Success Criteria

In Section 3.1, I defined a set of 12 success criteria. In Table 5.1 below, I have described whether or not it was fulfilled, along with a rating scale of 1-5 (where 1 means not fulfilled at all and 5 means perfectly fulfilled).

Table 5.1: Success Criteria Evaluated

Marker	Rating	Comments
C1	5	The matrix representation of polyominoes is effective and easily manipulated
C2	5	The matrix representation of boards is effective and easy to combine with other matrices
C3	5	The generator works quickly and easily, and the generated boards are all valid
C4	5	It was easy to check the number of possible boards
C5	5	The solver can find the first solution in under 3 seconds every time
C6	5	The user can input a possible boards through the command line, and the interface does not allow for incorrect coordinate inputs
C7	4	The user may pick a number, which is connected to a board that is returned. This is not exactly random, but serves our purposes

C8	3	The user interface is somewhat easy to use, dependant on whether the user is comfortable with using terminals. There are clear instructions on how to use the interface and what the user should input. It would, however, be easier to implement a front-end representation of the solver
C9	5	The solver can find all solutions, even if the generation is not instant
C10	4	While it is clear which polyomino goes in which spaces of the solution board, it would be easier with a more graphical representation (such as colour)
C11	1	A user cannot interact with the solver flexibly enough to use it to try and solve the board themselves
C12	1	The hint system was not implemented

5.2 Time Test Evaluation

As shown in Table 4.1, the project's fundamental need of generating one solution board quickly was achieved, as in each case this took less than 3 seconds. This means that the model was successful as a solver and, with further development that is discussed in Section 5.3, could become a preferable option for solving various polyomino tiling problems, even beyond the Genius Square itself.

As perhaps expected, a larger number of potential solutions imply a slower calculation time. Even though being able to generate all solution boards quickly is not part of the current requirements for the project, it is worth exploring the optimisation of this process to showcase the computational power offered by functional programming, and extend the capabilities of the model.

5.3 Potential Improvements

While the current version of the project represents a significant milestone in achieving its objectives, there are several areas where further enhancements could be explored. By addressing these potential improvements the project can continue to evolve, allowing for expansion into larger endeavors.

5.3.1 Graphical User Interface

We could increase user satisfaction by implementing a better user interface using a Haskell library such as Threepenny-gui [30] [31], which uses a web browser as a front. This would provide a more intuitive and visually engaging way for users to interact with the project compared to the terminal.

This would also open up the project to a wider audience by catering to users who prefer graphical interfaces or may have limited experience with command line environments. This inclusivity can attract a broader user base and increase the project's impact and reach. Furthermore, it would be easier for users to "drag and drop" polyominoes onto the board when using the hint feature, rather than entering individual coordinates. Also, presenting any solution boards using colours to indicate different shapes would be easier to visualise than the current number system that is in place.

5.3.2 Hint System Development

As aforementioned, a hint system could be fully added to the implementation in order to open it up for more advanced user engagement. The method described in Section 3.5 is sound for a small puzzle such as the Genius Square, because it can be solved intuitively in most cases anyway. However, it is worth considering that this implementation could be extended into solving bigger problems, which would be harder for a human to solve. In these cases, the hint system could be optimised to give a user their best chance for solving it.

For instance, rather than generating just one possible solution board for a user input, the system could generate all of the possible solutions, ask the user which polyomino to tile, and then parse through each solution to find the most common placement of that polyomino, then combine it with the input board and return it to the user. In this way, the user would have the most choice in terms of future polyomino placements, and therefore would be more likely to solve the board quicker.

Alternatively, for the purposes of enjoyment the user could instead choose for the system to return a placement that is "harder" to solve. This would accommodate users of varying skill levels and prolong user interaction with the puzzle. Such a feature is an example of a potential extension talked about in Section 5.2.

While this is a better method than described in Section 3.6.1, it also depends on how quickly we can generate large lists of solution boards.

Because we may have thousands of boards to parse through, and because a more complex method isn't necessary for our purposes, the basic hint system is quite sufficient.

5.3.3 Formal Proof

Another area that could be improved within the project is formally proving the correctness of the algorithm. This could confirm two key assumptions that are implied within the implementation:

1. **Completeness:** The solver finds all possible solutions for any given instance of the puzzle.
2. **Soundness:** Every solution provided by the solver is a valid solution to the puzzle instance.

The process of constructing a formal proof often involves scrutinising the algorithm's logic and structure in detail, and doing so can help identify potential flaws, edge cases, or unintended consequences that may not be immediately apparent during informal (not to mention incomplete) testing, especially in the case of extending the design for more advanced purposes. Addressing these issues is important in the development of any application.

On the other hand, proving these assumptions would strengthen confidence in the algorithm's effectiveness and suitability for its intended purpose of solving the Genius Square. This, in turn, would facilitate collaboration with others who might have a use for the algorithm.

6 Conclusion

All in all, this project has succeeded in creating a computational solver for the Genius Square puzzle using Haskell. This demonstrates the practical applicability of functional programming and illustrates its viability in the implementation of polyomino tiling algorithms, thereby highlighting its capability of addressing potentially complex real-world challenges.

The model uses the best techniques available in Haskell to be able to find the possible tilings of any valid Genius Square board, and one solution is able to be found quickly (within 3 seconds). However, the current version of the model does not include features such as a highly interactive user interface or a hint system, making the functional implementation less preferable to a user than existing solvers which use a more dynamic approach (such as the Dancing Links technique used with Algorithm X).

Nonetheless, the model's ability to effectively find *all* solutions to a tiling problem, rather than just one, could make it a worthy candidate for other, more complex problems.

A Appendix

A.1 Polyomino Implementation

```
--P1: MONOMINO  
p1 = fromList 1 1 [1]  
p1all = [p1]
```

```
--P2: DOMINO  
p2i1 = fromList 1 2 [1,1]  
p2i2 = M.transpose p2i1  
p2all = [p2i1,p2i2]
```

```
--P3: STRAIGHT TROMINO  
p3i1 = fromList 1 3 [1,1,1]  
p3i2 = M.transpose p3i1  
p3all = [p3i1,p3i2]
```

```
--P4: RIGHT TROMINO  
p4i1 = fromLists [[1,1],[1,0]]  
p4i2 = switchCols 1 2 p4i1  
p4i3 = switchRows 1 2 p4i2  
p4i4 = switchCols 1 2 p4i3  
p4all = [p4i1,p4i2,p4i3,p4i4]
```

```
--P5: STRAIGHT TETROMINO  
p5i1 = fromList 1 4 [1,1,1,1]  
p5i2 = M.transpose p5i1  
p5all = [p5i1,p5i2]
```

```
--P6: SQUARE TETROMINO  
p6 = fromLists [[1,1],[1,1]]  
p6all = [p6]
```

```
--P7: T TETROMINO
p7i1 = fromLists [[0,1,0],[1,1,1]]
p7i2 = M.transpose p7i1
p7i3 = switchRows 1 2 p7i1
p7i4 = M.transpose p7i3
p7all = [p7i1,p7i2,p7i3,p7i4]
```

```
--P8: L TETROMINO
p8i1 = fromLists [[1,1,1],[1,0,0]]
p8i2 = switchCols 1 3 p8i1
p8i3 = switchRows 1 2 p8i1
p8i4 = switchCols 1 3 p8i3
p8i5 = M.transpose p8i1
p8i6 = switchRows 1 3 p8i5
p8i7 = switchCols 1 2 p8i5
p8i8 = switchRows 1 3 p8i7
p8all = [p8i1,p8i2,p8i3,p8i4,p8i5,p8i6,p8i7,p8i8]
```

```
--P9: SKEW TETROMINO
p9i1 = fromLists [[0,1,1],[1,1,0]]
p9i2 = switchRows 1 2 p9i1
p9i3 = M.transpose p9i1
p9i4 = M.transpose p9i2
p9all = [p9i1,p9i2,p9i3,p9i4]
```

```
unusedPi = [p1all,p2all,p3all,p4all,p5all,p6all,p7all,
             p8all,p9all]
```

A.2 Labelled Matrix Validity Checker

```
snc :: (Num a, Eq a) => Int -> Matrix a -> Bool
snc h mat = (all (/= fromIntegral(h+3)) (concat(toLists
    mat)))
    && (all (/= fromIntegral(h+4)) (concat(toLists
    mat)))
    && (all (/= fromIntegral(h+5)) (concat(toLists
    mat)))
    && (all (/= fromIntegral(h+6)) (concat(toLists
    mat)))
    && (all (/= fromIntegral(h+7)) (concat(toLists
    mat)))
```

```

        && (all (/= fromIntegral(h+8)) (concat(toLists
mat)))
        && (all (/= fromIntegral(h+9)) (concat(toLists
mat)))
        && (all (/= fromIntegral(h+10)) (concat(toLists
mat)))

```

A.3 User Interface

```

pickChoice :: (Num a) => IO(Matrix a)
pickChoice = do
    putStrLn showBoard
    putStrLn "Pick your choice for the first die:\n1.
(4,e)\n2. (5,d)\n3. (5,e)\n4. (5,f)\n5. (6,d)\n6. (6,
e) "
    dc1 <- getLine
    putStrLn "Pick your choice for the second die:\n1.
(1,b)\n2. (1,c)\n3. (2,a)\n4. (2,b)\n5. (2,c)\n6. (3,
b) "
    dc2 <- getLine
    putStrLn "Pick your choice for the third die:\n1.
(1,a)\n2. (3,a)\n3. (4,a)\n4. (4,b)\n5. (5,b)\n6. (6,
c) "
    dc3 <- getLine
    putStrLn "Pick your choice for the fourth die:\n1.
(1,f)\n2. (6,a) "
    dc4 <- getLine
    putStrLn "Pick your choice for the fifth die:\n1.
(2,d)\n2. (3,c)\n3. (3,d)\n4. (4,c)\n5. (4,d)\n6. (5,
c) "
    dc5 <- getLine
    putStrLn "Pick your choice for the sixth die:\n1.
(1,d)\n2. (2,e)\n3. (3,e)\n4. (3,f)\n5. (4,f)\n6. (6,
f) "
    dc6 <- getLine
    putStrLn "Pick your choice for the seventh die:\n1.
(1,e)\n2. (2,f)\n3. (5,a)\n4. (6,b) "
    dc7 <- getLine
    return (matrix 6 6 $ \(i,j) -> if (i,j) `elem` [(
die1 !! (read dc1 - 1)), (die2 !! (read dc2 - 1)), (
die3 !! (read dc3 - 1)), (die4 !! (read dc4 - 1)), (
die5 !! (read dc5 - 1)), (die6 !! (read dc6 - 1)), (
die7 !! (read dc7 - 1))] then 1 else 0)

```

A Appendix

```
randomChoice :: (Num a) => IO (Matrix a)
randomChoice = do
    putStrLn "Pick a number between 1 and 62208"
    boardNumber <- getLine
    return (allBoards allDice !! (read boardNumber - 1))
```

```
main = do
    putStrLn "Hello! Here is the general layout of the
    Genius Square board:"
    putStrLn showBoard
    putStrLn "Would you like to:\n1. Generate a random
    board\n2. Pick your own coordinates"
    choice <- getLine
    let bn1 = if choice == "1" then randomChoice else
    pickChoice
    bn <- bn1
    putStrLn "Your board looks like this:"
    print bn
    putStrLn "The solution boards are labelled as
    follows:\n0: BLOCKER\n1: MONOMINO\n2: DOMINO\n3:
    STRAIGHT TROMINO\n4: RIGHT TROMINO\n5: STRAIGHT
    TETROMIN\n6: SQUARE TETROMINO\n7: T TETROMINO\n8: L
    TETROMINO\n9: SKEW TETROMINO"
    timeIt $ print $ together bn unusedPi [1..9]
    print $ length((together bn unusedPi [1..9])!!0)
```

A.4 Using the User Interface

```
elzgor@Tux:~$ cabal run gensqu
Hello! Here is the general layout of the Genius Square
board:

|a|b|c|d|e|f|
-----
1| | | | | | |
-----
2| | | | | | |
-----
3| | | | | | |
-----
4| | | | | | |
-----
5| | | | | | |
```



```
-----
6| | | | | | |
-----
```

Would you like to:
 Generate a random board
 Pick your own coordinates
 1
 Pick a number between 1 and 62208
 542
 Your board looks like this:

```

0 1 0 0 0 0
0 0 0 0 0 1
1 0 0 0 0 1
0 0 0 1 1 0
0 0 0 0 0 0
1 0 0 0 0 0
```

The solution boards are labelled as follows:

0: BLOCKER
 1: MONOMINO
 2: DOMINO
 3: STRAIGHT TROMINO
 4: RIGHT TROMINO
 5: STRAIGHT TETROMIN
 6: SQUARE TETROMINO
 7: T TETROMINO
 8: L TETROMINO
 9: SKEW TETROMINO

```
[[
1 0 2 2 9 9
3 3 3 9 9 0
0 5 5 5 5 0
6 6 8 0 0 7
6 6 8 4 7 7
0 8 8 4 4 7
```

```

1 0 2 2 9 9
3 3 3 9 9 0
0 5 5 5 5 0
8 8 8 0 0 7
8 6 6 4 7 7
0 6 6 4 4 7
```

```

1 0 2 2 8 8
5 5 5 5 8 0
0 7 7 7 8 0
```

A Appendix

```
6 6 7 0 0 4
6 6 9 9 4 4
0 9 9 3 3 3
...
```

```
elzgor@Tux:~$ cabal run gensqu
Hello! Here is the general layout of the Genius Square
board:
```

```
 |a|b|c|d|e|f|
-----
1| | | | | | |
-----
2| | | | | | |
-----
3| | | | | | |
-----
4| | | | | | |
-----
5| | | | | | |
-----
6| | | | | | |
-----
```

```
Would you like to:
Generate a random board
Pick your own coordinates
2
```

```
 |a|b|c|d|e|f|
-----
1| | | | | | |
-----
2| | | | | | |
-----
3| | | | | | |
-----
4| | | | | | |
-----
5| | | | | | |
-----
6| | | | | | |
-----
```

```
Pick your choice for the first die:
(4,e)
(5,d)
(5,e)
```

A Appendix

```
(5,f)
(6,d)
(6,e)
4
Pick your choice for the second die:
(1,b)
(1,c)
(2,a)
(2,b)
(2,c)
(3,b)
3
Pick your choice for the third die:
(1,a)
(3,a)
(4,a)
(4,b)
(5,b)
(6,c)
6
Pick your choice for the fourth die:
(1,f)
(6,a)
1
Pick your choice for the fifth die:
(2,d)
(3,c)
(3,d)
(4,c)
(4,d)
(5,c)
3
Pick your choice for the sixth die:
(1,d)
(2,e)
(3,e)
(3,f)
(4,f)
(6,f)
5
Pick your choice for the seventh die:
(1,e)
(2,f)
(5,a)
(6,b)
1
Your board looks like this:
```

```

0 0 0 0 1 1
1 0 0 0 0 0
0 0 0 1 0 0
0 0 0 0 0 1
0 0 0 0 0 1
0 0 1 0 0 0

```

The solution boards are labelled as follows:

```

0: BLOCKER
1: MONOMINO
2: DOMINO
3: STRAIGHT TROMINO
4: RIGHT TROMINO
5: STRAIGHT TETROMIN
6: SQUARE TETROMINO
7: T TETROMINO
8: L TETROMINO
9: SKEW TETROMINO

```

```

[[
  1 2 2 4 0 0
  0 6 6 4 4 9
  5 6 6 0 9 9
  5 3 3 3 9 0
  5 8 8 8 7 0
  5 8 0 7 7 7
],
[
  1 2 2 9 0 0
  0 4 9 9 6 6
  4 4 9 0 6 6
  8 5 5 5 5 0
  8 3 3 3 7 0
  8 8 0 7 7 7
],
[
  1 2 2 8 0 0
  0 6 6 8 8 8
  5 6 6 0 4 4
  5 7 9 9 4 0
  5 7 7 9 9 0
  5 7 0 3 3 3
],
...

```

A.5 Placement Function

```
placePlaced :: a -> Int -> Matrix a -> IO(Matrix a)
placePlaced placed 0 mat = return mat
placePlaced placed x mat = do
    putStrLn "Which position in the above matrix is it
    occupied by?"
    pos1 <- getLine
    let pos2 = read pos1
    let temp = setElem placed ((fst (divMod pos2 6)) +
    1, snd (divMod pos2 6)) mat
    --print temp
    placePlaced placed (x-1) temp
```

A.6 Direct Implementation

```
getr :: (Fractional a, Eq a) => Matrix a -> Int -> Int
getr m x
    | getElem x x m /= 1.0 = x
    | otherwise = getr m (x+1)
```

```
backSub :: (Num a) => [[a]] -> [a]
backSub mat = foldr nxt [final (final matrix)] (first
    matrix)
    where
        nxt row found = let
            part = first $ drop (length mat - length found)
        row
            sol = final row - sum (zipWith (*) found part)
            in sol : found
```

```
toUppTri :: (Num a, Eq a, Fractional a) =>
    Matrix a -> [Int] -> (Matrix a, [Int])
toUppTri mat lst
    | ncols mat == nrows mat = (mat, lst)
    | otherwise = toUppTri
        (fromLists ((take ((getr mat 1) - 1) (ms mat))
        ++ (ms (nextRow (getr mat 1)))
        ++ (drop ((getr mat 1) - 1) (ms mat)))
        (lst ++ [((getr mat 1) - 1)]))
    where
        nextRow r = matrix 1 (ncols mat)
            $ \(i,j) -> if (i,j) == (1,r) then 1 else 0
        ms mat = toLists mat
```

A.7 Linear Optimiser Implementation

```

-- From Math.LinearEquationSolver
solveIntegerLinearEqsAll :: Solver          -- ^ SMT
  Solver to use
  --> Int                                -- ^
  Maximum number of solutions to return, in case
  infinite
  --> [[Integer]]                        -- ^
  Coefficient matrix (A)
  --> [Integer]                          -- ^ Result
  vector (b)
  --> IO [[Integer]]                     -- ^ All
  solutions to @Ax = b@
solveIntegerLinearEqsAll s maxNo coeffs res =
  extractModels `fmap` allSatWith cfg cs
  where cs = buildConstraints "solveIntegerLinearEqsAll"
        coeffs res
        cfg = (defaultSolverConfig s) {
          allSatMaxModelCount = Just maxNo}

```

```

-- From Math.LinearEquationSolver
-- Build the constraints as given by the coefficient
  matrix and the resulting vector
buildConstraints :: (Ord a, Num a, SymVal a) => String
  -> [[a]] -> [a] -> Symbolic SBool
buildConstraints f coeffs res
  | m == 0 || any (/= n) ns || m /= length res
  = error $ f ++ ": received ill-formed input."
  | True
  = do xs <- Data.SBV.mkFreeVars n
        let rowEq row r = sum (zipWith (*) xs row) .== r
        solve $ zipWith rowEq (map (map literal) coeffs)
          (map literal res)
  where m      = length coeffs
        n:ns  = map length coeffs

```

```

listBinary :: [Integer] -> Bool
listBinary xs = all (\x -> x == 0 || x == 1) xs

```

A Appendix

```
solveIt mat vect = do
  candidates <- solveIntegerLinearEqsAll Z3 8000 mat
  vect
  print(head (filter listBinary candidates))
```

Bibliography

- [1] [Online]. Available: https://tetris.com/play-tetris/?utm_source=top_nav_link&utm_medium=webnav&utm_campaign=playNow_btm_tst&utm_content=text_play_now.
- [2] [Online]. Available: <https://www.happypuzzle.co.uk/family-puzzles-and-games/genius-square>.
- [3] P. Hudak and J. H. Fasel, 'A gentle introduction to haskell,' *ACM Sigplan Notices*, vol. 27, no. 5, pp. 1–52, 1992.
- [4] [Online]. Available: <https://www.haskell.org/>.
- [5] F. Hayat, A. U. Rehman, K. S. Arif, K. Wahab and M. Abbas, 'The influence of agile methodology (scrum) on software project management,' in *2019 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2019, pp. 145–149. DOI: 10.1109/SNPD.2019.8935813.
- [6] M. Coram and S. Bohner, 'The impact of agile methods on software project management,' in *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, 2005, pp. 363–370. DOI: 10.1109/ECBS.2005.68.
- [7] D. J. Fernandez and J. D. Fernandez, 'Agile project management—agilism versus traditional approaches,' *Journal of Computer Information Systems*, vol. 49, no. 2, pp. 10–17, 2008. DOI: 10.1080/08874417.2009.11646044. eprint: <https://www.tandfonline.com/doi/pdf/10.1080/08874417.2009.11646044>. [Online]. Available: <https://www.tandfonline.com/doi/abs/10.1080/08874417.2009.11646044>.
- [8] *A Modular Design Approach to Support Sustainable Design*, vol. Volume 3d: 8th Design for Manufacturing Conference, International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, 2004, pp. 909–918. DOI: 10.1115/DETC2004-57775. eprint: https://asmedigitalcollection.asme.org/IDETC-CIE/proceedings-pdf/IDETC-CIE2004/46962d/909/2617413/909_1.pdf. [Online]. Available: <https://doi.org/10.1115/DETC2004-57775>.

Bibliography

- [9] S. M. Salhie and A. K. Kamrani, 'Modular design,' in *Collaborative Engineering: Theory and Practice*, A. K. Kamrani and E. S. A. Nasr, Eds. Boston, MA: Springer US, 2008, pp. 207–226, ISBN: 978-0-387-47321-5. DOI: 10.1007/978-0-387-47321-5_10. [Online]. Available: https://doi.org/10.1007/978-0-387-47321-5_10.
- [10] 'Quotation: Lecture to courant institute,' 1990. [Online]. Available: <https://mathshistory.st-andrews.ac.uk/Biographies/Gelfand/quotations/>.
- [11] N. Jensen, 'Solving the genius square: Using math and computers to analyze a polyomino tiling game,' *WWU Honors College Senior Projects*, 711, 2023.
- [12] S. W. Goulomb, *Polyominoes: Puzzles, Patterns, Problems and Packings - Revised and expanded second edition*. Princeton Science Library, 1996.
- [13] B. Gwee and M. H. Lim, 'Polyominoes tiling by a genetic algorithm,' *School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore 2263*, 1995.
- [14] M. R. Garvie and J. Burkardt, 'A new mathematical model for tiling finite regions of the plane with polyominoes,' *Contributions to Discrete Mathematics*, Volume 15, Number 2, Pages 95-131, 2019.
- [15] D. E. Knuth, 'Dancing links,' *Stanford University*, 2000. [Online]. Available: <https://doi.org/10.48550/arXiv.cs/0011047>.
- [16] Various, *On the Exact Cover Problem*. Springer and Cham, 2014. [Online]. Available: https://doi.org/10.1007/978-3-319-07956-1_2.
- [17] [Online]. Available: <https://pypi.org/project/exact-cover/#:~:text=The%20exact%20cover%20problem%20is,empty%20intersection%20with%20each%20other..>
- [18] [Online]. Available: <https://cemulate.github.io/polyomino-solver/>.
- [19] A. Ruiz, 'Introduction to hmatrix,' 2012. [Online]. Available: <https://dis.um.es/~alberto/material/hmatrix.pdf>.
- [20] [Online]. Available: <https://hackage.haskell.org/package/hmatrix-0.20.2/docs/Numeric-LinearAlgebra-HMatrix.html>.
- [21] A. Santos and J. N. Oliveira, 'Type your matrices for great good: A haskell library of typed matrices and applications (functional pearl),' *Association for Computing Machinery, New York*, 2020. [Online]. Available: <https://dl.acm.org/doi/pdf/10.1145/3406088.3409019>.
- [22] [Online]. Available: <https://hackage.haskell.org/package/laop>.
- [23] [Online]. Available: <https://hackage.haskell.org/package/matrix-0.3.6.3/docs/Data-Matrix.html>.

Bibliography

- [24] [Online]. Available:
<https://hackage.haskell.org/package/linearEqSolver-2.3/docs/Math-LinearEquationSolver.html>.
- [25] G. Érdi, 'An adventure in symbolic execution,' in *32nd Symposium on Implementation and Application of Functional Languages*, 2020.
- [26] [Online]. Available:
<https://hackage.haskell.org/package/sbv-1.1/docs/Data-SBV.html>.
- [27] L. de Moura and N. Bjørner, 'Z3: An efficient smt solver,' in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78800-3.
- [28] [Online]. Available: <https://github.com/z3prover/z3>.
- [29] [Online]. Available: <https://hackage.haskell.org/package/timeit-2.0/docs/System-Timelt.html>.
- [30] B. Finkbeiner, F. Klein, R. Piskac and M. Santolucito, 'Synthesizing functional reactive programs,' in *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, 2019, pp. 162–175.
- [31] [Online]. Available:
<https://hackage.haskell.org/package/threepenny-gui>.