University of Dublin

TRINITY COLLEGE

Functional Programming: An Untapped Tool for Linear Algebra?

Diarmuid McDonnell
M.A.I. Engineering
Supervisor: Dr. Hugh Gibbons

Submitted to the University of Dublin, Trinity College, May, 2018

I, Diarmuid McDonnell, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

Signature: ______________________________ Date: ___________________________

Summary

This research paper investigates whether or not functional programming would be a good tool for solving linear algebra problems. This is carried out through building an application in the functional programming language Haskell to solve Lights Out. The paper first identifies key aspects of functional programming that may be of benefit when solving linear algebra problems. Past research is discussed which shows how functional programming has been used to benefit other areas of mathematics. Some mathematical software are also identified which are typically used for solving these problems, along with linear algebra functions that should be addressed in any proposed linear algebra toolkit. The Lights Out game is discussed in order to establish its suitability as a linear algebra problem. The equation for solving Lights Out is proven to be a matrix product, thus solving it using functional programming will demonstrate its ability to solve linear algebra problems. The main challenge is identified as executing a matrix inverse on a non-invertible matrix to produce a pseudo inverse matrix.

An algorithm for executing the matrix inverse operation is designed and implemented in Haskell which returns the desired result and allows for the solving of Lights Out. This paper identifies the key characteristics of functional programming that benefit the application for solving Lights Out and compares them to imperative languages. It is found that functional programming is able to not only invert a matrix but find a pseudo inverse for a non-invertible matrix, that it is capable of solving linear algebra problems. However, some

further research may be necessary in order to investigate a larger range of linear algebra problems to determine just how capable a tool functional programming is.

Abstract

Functional Programming: An Untapped Tool for Linear Algebra? – Diarmuid McDonnell

This research proposes the idea that functional programming is a suitable tool for solving linear algebra problems. The Lights Out game is used to explore this idea. This paper shows how Lights Out is a non-trivial linear algebra problem suitable for testing the capabilities of functional programming as a tool for computing linear algebra functions. An application is built to solve Lights Out using the functional programming language Haskell. The results presented in this paper show that functional programming is well suited for solving linear algebra problems and could rival the use of imperative languages. Some future work is also proposed to improve upon the work carried during this research.

Acknowledgements

I would like to thank my supervisor Dr. Hugh Gibbons for his help throughout the year. His advice during our weekly meetings was extremely beneficial.

I would also like to thank my family and girlfriend for the support they’ve given me throughout the entirety of my university career.

Table of Contents

1 Introduction

This project investigates whether or not functional programming is a good tool for solving linear algebra problems. The electronic board game Lights Out will be used as a method of demonstrating this.

Figure 1.1

Throughout this paper the characteristics of functional programming are outlined that could potentially make it a useful tool for anyone working on problems in linear algebra. As well as that, some past research is discussed that hints at functional programming being a capable tool solving linear algebra problems.

The background of linear algebra is also discussed, with some examples of common linear algebra functions that a program to solve linear algebra problems would have to address. There has also been research carried out in the field of linear algebra with regards to improving the efficiency in computation when carrying out linear algebra operations. Some of this research is discussed as it shows the ever-evolving state of linear algebra in computing as well as the continued need for finding new and more efficient ways of carrying out linear algebra operations. The mathematical software that is typically used to perform these linear algebra operations are identified and discussed, as well as the potential for these software to be built with a functional programming language in the back end.

An overview of Lights Out is given. A lot of the information gathered for this research is taken from Jamie Mulholland's web page[1] and lecture 24 of his module "Permutation Puzzles: A Mathematical Perspective"[2] as well as Jaap Scherpuis' puzzle page[3]. Both Mulholland and Scherpius heavily base their work on a paper by Anderson and Feil (1998)[4]. In this overview, the rules and objective of the game is outlined. The maths behind solving the puzzle is then discussed which demonstrates why Lights Out is a suitable tool for demonstrating that functional programming can solve linear algebra problems. This is done by showing how to get a solution for the puzzle and then optimising that solution to take the least number of moves. A working example is walked through in order to further explain the process of solving Lights Out and back up the theory discussed.

With all the background to the project addressed, the method by which Lights Out is solved is outlined. This involves outlining the algorithms behind calculating the pseudo inverse of the Lights Out matrix, checking whether or not a given configuration is solvable, and calculating a solution for a given configuration. The method for calculating the optimal solution of a given configuration is also addressed.

With the application built using the algorithms outlined in the method, the results of the algorithms are identified and discussed. The advantages of functional programming for solving linear algebra problems identified through the process of implementing the application are compared to more typically used, imperative languages such as C and Java. Some future work for improving the application are suggested.

2 Background

This project can be broken up into three parts:

1. Functional Programming
2. Linear Algebra
3. Lights Out

In this section the current start of the art for these three areas will be discussed. The areas highlighted will be those that pertain to this project and help show why functional programming has the potential to be a good tool for solving linear algebra problems.

2.1 Functional Programming

There are a number of characteristics that make a programming language a functional programming language. Below are just some of those characteristics:

Declarative:	Functional programming languages are declarative, meaning that they describe what the program must do. An imperative programming language on the other hand would describe how a program would do something.
Pure Functions:	Functions in functional programming languages are pure functions. If a function is called with the same inputs, it will always produce the same output. This is due to the fact that

	functions in functional programming languages have no side effects.
Referential Transparency:	Once defined, a variable in a functional programming language cannot be changed. This means that at any point, a variable can be replaced with its actual value without affecting the end result.
Higher Order Functions:	Functional programming languages make use of higher order functions. This means that a function can take another function as an input, return a function as an output, or both.
Lazy Evaluation:	Some functional programming language functions use lazy or non-strict evaluation. This means that a function is only executed once its result is required by another function. It also allows some functions to be partially outputted even if one of the values in the output cannot be evaluated. In this case the functions output will include everything up to the point of the failing evaluation.
Recursion:	Recursion is used in most programming languages but is much more prevalent in functional programming languages. This is due to the fact that it is the main technique used in iteration. This is done by defining a ‘base case’ that marks the end point of the iteration.

Other Characteristics	Other defining characteristics of functional programming languages include the fact that there is no shared state, mutable data or side effects in these languages.

There have been a number of arguments for why functional programming is beneficial in a wide variety of areas. This shows that functional programming can be useful for many different use cases and is worthy of being investigated for its use in solve linear algebra problems. One example of this is Ball (1999)[5] who argued that functional programming is very useful for avoiding inexecutable paths compared to imperative languages. More specifically, functional programming allows programmers to avoid destructive updates and unnecessary sequencing.

Another supporter of functional programming, Hughes (1995)[6], argued that functional programming languages are very good at supporting software reuse. He wanted to demonstrate this by highlighting how higher order functions allow the body of a loop to be more easily implemented. This was carried out through a study of a pretty-printing library. This study showed that formal specifications and the use of the algebraic properties of higher order functions can be used to design and implement a library of common programming idioms in the application.

There have even been some studies in the area of using functional programming for testing software. Thompson (1993)[7] explored this idea by using Haskell to try and prove functional programs correct. However, he found that some of the difficulties in this area outweighed any benefits gained. There has been further research in the area since however. Claessen and Hughes (2000)[8] discuss QuickCheck, a testing framework written in Haskell for Haskell. This study discusses the fact that despite some pitfalls, the framework works successfully over a large range of use cases.

There have also been a number of studies suggesting that functional programming is a valuable tool in the field of mathematics. Page (2001)[9] states that although functional programming still has some barriers it has yet to overcome (performance issues and programmers resistance to the use of non-standard languages), it is a very effective teaching tool. Page states that teaching the application of math logic and reasoning in the software development process draws benefits from the use of functional programming. As well as that, Doets (2012)[10] argues that Haskell is a marvellous demonstration tool for logic and maths. Doets also states that Haskell can be viewed as an elegant implementation of lambda calculus. These papers hint towards the idea that functional programming has its place in the field of mathematics and could potentially benefit linear algebra.

This point is further strengthened by a number of studies which suggest that functional programming would indeed be a good tool for Linear Algebra. Dolan (2013)[11] defined a type-class for describing closed semerings in Haskell. In doing so, he also implemented a few functions for manipulating matrices which show functional programming's ability to work on matrices. Eriksson and Jansson (2016)[12] further showed this by defining a block based matrix in the functional programming language Agda. They verify that algorithms can be implemented using the closure operation of a semi-ring by lifting algebraic structures to matrices. Both of these papers show previous use of functional programming for solving linear algebra problems. This suggests that the current research can indeed find that functional programming is a good tool for solving linear algebra problems.

There has even been research to suggest that functional programming can challenge the already established tools in the field of mathematics. Eaton (2006)[13] argues that because Matlab and other popular matrix languages are dynamically typed, type errors are only caught at runtime. Eaton states that by exposing object dimensions to the type system, a much wider range of common errors can be detected at compile time. Using Haskell, Eaton writes what he calls a “strongly typed linear algebra” prototype based on a Haskell library by Alberto Ruiz[14]. This is a Haskell library that provides a fully functional interface for linear algebra and other numerical algorithms. Eaton’s prototype is also uses techniques from Kiselyov and Shan (2004)[15], a paper that uses widely implemented language features such as the type-class system to solve configuration problems in Haskell. As well as Eaton, Ghitza and Westerholt-Raum (2016)[16] have compared Haskell to tools commonly used to solve linear algebra problems. They present an implementation of the PLE decomposition of matrices over division rings and discovered that a relevant number of cases performed faster than C-based implementations.

Looking at all of this past research certainly shows that the use functional programming as a tool for solving problems in linear algebra should be researched further. The current research aims to do this and provide some insight into how it can be done.

2.2 Linear Algebra

There are a number of constantly occurring developments in the field of linear algebra regarding the use of computers for solving problem. Buttari, Langou, Kurzak and Dongarra (2008)[17] addressed the fact that as computing power grows, linear algebra algorithms have to be reformatted. In this paper, they present a new algorithm for QR factorisation (the decomposition of a matrix into a product of an orthogonal and an upper triangular matrix) to make use of this greater computing power. Another paper that shows the continuous updating of computing algorithms in linear algebra is Quintana-Ortí, Quintana-Ortí, van de Geijn, van Zee and Chan (2009)[18]. This research highlights that continued performance improvements have come about due to growth in thread level parallelism. They argue that it is not viable to evolve legacy libraries for dense and banded linear algebra. This is because of the constraints imposed by the early design of the library. This means that in some cases an upgrade in linear algebra algorithms may require starting from scratch in order to get the best results in terms of performance.

It can even be shown that common matrices such as matrix multiplication and matrix inversion can be updated in order to make them perform more efficiently. Quintana, Quintana, Sun and van de Geijn (2000)[19] show this by presenting a new one-sweep parallel algorithm for matrix inversion. A paper that puts forward a new algorithm for matrix multiplication is Gunnels, Henry, van de Geijn (2001). The aim of this paper was to employ maths to determine a locally optimal method for blocking matrices. The resulting algorithm, combined with a highly optimised inner-kernel, produces a higher performing matrix multiplication compared to that of algorithms using an automatically tuned kernels. A later paper, Goto and van de Geijn (2008), present the basic principles that underline the high performance of matrix multiplication. They present a simple but effective algorithm for executing matrix multiplication. When implemented over a large range of architecture, the algorithm met near-peak performance. These studies show that even the more common matrix operations can be optimised. The current study is looking at optimising some of these linear algebra operations by use of functional programming and these studies show that it is a worthwhile endeavour.

2.2.1 Common Linear Algebra Functions

There are a number of common linear algebra functions. If a functional programme were to fully support linear algebra, it would need to be able to execute these functions on any viable input. Below are some of these equations, some of which will be addressed by the application produced by the current study.

Dot Product:	Also known as the scalar product, the dot product can be performed on two vectors of equal length. The result is the sum of the products of the corresponding entries in the vector. e.g $(v_1, v_2, v_3) \cdot (u_1, u_2, u_3) = v_1u_1 + v_2u_2 + v_3u_3$
Cross Product:	Also known as vector product, the cross product is used to find a vector perpendicular to two given vectors in 3D space. The equation for cross product is $a \times b = \|a\|\|b\| \sin\theta$
Rank:	The rank of a matrix A is determined by the row and column ranks of that matrix. The row rank is the number of linearly independent rows in A while the column matrix is the number of linearly independent columns in A. The rank of matrix A is the lowest of these two values.

Determinant: A determinant can only be calculated for a square matrix. The equation for the determinant of a 2x2 square matrix is

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$$

For square matrices larger than that the determinant is the sum of each number in the first row multiplied by the determinant of the square matrix of the remaining rows, not including its column. Every second value of the sum is negative.

Gaussian Elimination: This is an algorithm used in linear algebra to convert a matrix to row echelon form using row operations. A matrix is in row echelon form when every non-zero row has a leading coefficient is to the right of the leading coefficient of the row above it. The row operations used to reduce the matrix to this form are:

- Swapping row positions
- Multiplying a row by a non-zero scalar
- Adding one row to a multiple of another

Gauss-Jordan Elimination: This is an algorithm that continues on from Gaussian elimination. In this case, the goal is to convert the matrix to reduced row echelon form. A matrix is in reduced row echelon form if further to being in row echelon form, the leading coefficients are all one, and all other values in a column with a leading one are zero.

Matrix Inverse: The inverse of a given square matrix can be found if the given matrix can be converted to reduced row echelon form. In order to find the inverse of a square matrix, Gauss-Jordan elimination is performed on it. Each operation performed on the matrix is also performed on an identity matrix of the same size. Once the original matrix has been reduced to reduced row echelon form, the result of the operations on the identity matrix is the inverse.

2.2.2 Mathematical Software

There are a number of mathematical software used to solve linear algebra problems. In this section some of these software will be discussed, along with the programming languages used to create them, and why functional programming could potentially do a better job at solving linear algebra problems.

2.2.2.1 MATLAB

MATLAB, developed by MathWorks, is a proprietary programming language written predominately in C, C++ and Java. It can be used for plotting functions and other data, and also for manipulating matrices.

2.2.2.2 Mathematica

Mathematica is a software developed by Wolfram Research. It is written in C, C++ and Java, but predominately Wolfram Language.

2.2.2.3 SageMath

SageMath is a computer algebra system initially created to provide an open source alternative to other mathematical software such as MATLAB and Mathematica. It is written in Python and Cython and covers many areas of mathematics including linear algebra.

These software make use of imperative languages to carry out their processes. The current study looks at the possibility of using a declarative functional programming language in place of these imperative languages to solve linear algebra problems that might need to be solved by users of the above software.

2.3 Lights Out

In 1995, Tiger Electronics released a game called Lights Out. The game consists of a 5x5 board of light up buttons. Each game starts with a random configuration of lights turned on and the objective of the game is to turn all the lights off. What makes the game difficult is that when a light is pressed, the lights immediately above, below, left and right of the pressed light as well as the pressed light itself, is toggled. People can come up with solutions to this game, but it is difficult to do. What makes this game even more challenging is attempting to solve an initial configuration in the minimum number of moves necessary. An example of a lights out board is shown in figure 2.1.

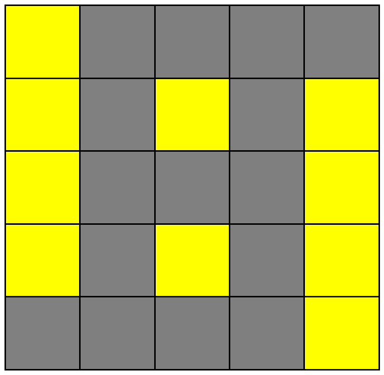


Figure 2.1

2.3.1 Solving the Puzzle

We can represent a Lights Out board using a 5x5 matrix. We can demonstrate the state of lights, the buttons being pressed or the result of a button press. The below matrix represents the above configuration of the board. 1 represents a lit light and 0 represents an unlit light.

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

The following two matrices represent a button being pressed and the result of that button being pressed. For the matrix on the left, the * represents the action of pressing a particular button. For the matrix on the right the 1 represents which lights on the board will change state as a result of that button press. This matrix is known as a toggle matrix. This demonstrates that if you press a single button twice the overall result of the board is unchanged. In the below case we can see that light (2,2) is pressed. As a result, lights (1,2), (2,1), (2,2), (2,3) and (3,2) are toggled. By combining a number of these actions, we can create a button pattern in order to get our desired result. The result of that button pattern will be the combination of the corresponding toggle matrices.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad + \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

What this would look like on a board is shown below, where green represents the button being pressed (figure 2.2), and red represents the lights that are toggled as a result of that button press (figure 2.3).

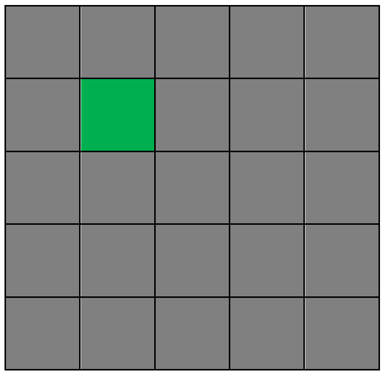


Figure 2.2

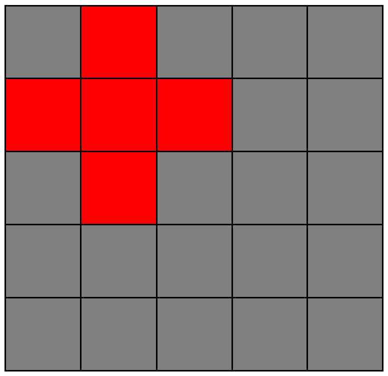


Figure 2.3

The act of pressing a button is performing an addition operation on the configuration matrix. Taking the above button press, this can be shown by adding the toggle matrix to a configuration of entirely unlit lights, or an all zero matrix.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} => \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

By combining multiple button presses we can find the resulting toggle matrix by combining the toggle matrices of each of those buttons. Below is an example of this. There are three button presses on the left-hand side with the resulting toggle matrix on the right-hand side.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ * & 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

In order to solve the puzzle, a button combination must be found that returns a toggle matrix that matches the initial configuration of the lights out board. This is shown in the below diagrams where the left-hand board (figure 2.4) represents the initial configuration and the right-hand board (figure 2.5) represents the toggle matrix required to make the board blank:

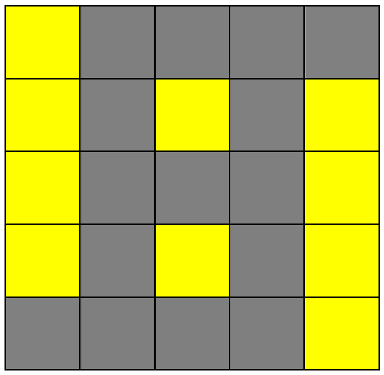


Figure 2.4

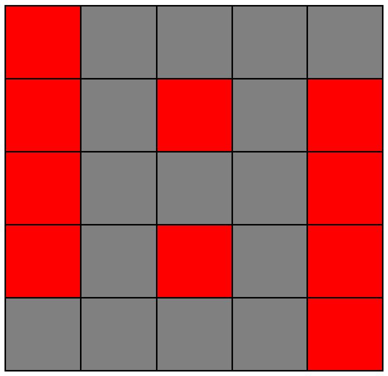


Figure 2.5

The resulting toggle matrix for each individual button press is known. In order to solve the puzzle, a combination of these toggle matrices is required that, when added together, matches the initial configuration as shown above. The buttons that are pressed to return these toggle matrices are the buttons required to complete the game. Therefore, the following equation can be used to solve Lights Out:

$$\sum_{\substack{1 \le i \le 5 \\ 1 \le j \le 5}} x_{i,j} T_{i,j} = B$$

Where, $x_{i,j}$ is a button on the board, $T_{i,j}$ is the toggle matrix corresponding to that button, and B is the initial configuration matrix. The solution to the game is the $x_{i,j}$'s required to return $T_{i,j}$'s, that when combined, result in a matrix that matches B. If this equation is expanded to show the individual elements of the summation, this can be seen more clearly.

$$x_{1,1}\begin{bmatrix}1&1&0&0&0\\1&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\end{bmatrix} + x_{1,2}\begin{bmatrix}1&1&1&0&0\\0&1&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\end{bmatrix} + \cdots + x_{5,5}\begin{bmatrix}0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&1\\0&0&0&1&1\end{bmatrix} = \mathrm{B}$$

The toggle matrices required to create a toggle matrix that matches the initial configuration B should have a coefficient of one and the toggle matrices that are not used should have a coefficient of zero. These coefficients are represented by the $x_{i,j}$ variable. Finding all of the $x_{i,j}$ values provides the solution, where a one value for a button $x_{i,j}$ represents a button that should be pressed, and a zero represents a button that should not be pressed.

Each member of the summation equation corresponds to a linear equation when represented as vectors rather than matrices. For example, entry (2, 2) in the above equation is:

$$x_{1,2} + x_{2,1} + x_{2,2} + x_{2,3} + x_{3,2} = b_{2,2}$$

These are the only toggle matrices that toggle the button (2, 2). We can represent each $b_{i,j}$ with the vector representation of B.

$$b = (b_{1,1},\ b_{1,2},\ b_{1,3}, \dots, b_{5,4}, b_{5,5})$$

The toggle matrices $T_{i,j}$ can be written as toggle vectors $t_{i,j}$. These toggle vectors can used as columns in a 25x25 matrix A

$$A = \left[t_{1,1}\middle|t_{1,2}\middle| \dots \middle|t_{5,5}\right]$$

This means that the above summation can be written as a matrix product

$$A\vec{x} = \vec{b}$$

Where $\vec{x}$ represents the coefficients of the toggle matrices in vector form. This vector shows the buttons to press in order to solve the puzzle. To find $\vec{x}$, the matrix A will have to be inverted.

This is where the problem becomes a linear algebra problem. In order to solve the above equation we first convert the button press matrix for a single button press to a binary matrix. This is shown below accompanied by its corresponding toggle matrix.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

By computing the above two matrices for every button on the board and converting each matrix into a vector of length 25, we can produce the two 25x25 matrices shown below. The matrix on the right is made up of 25 columns, each of which represent a single button being pressed on the Lights Out board. Each column of the matrix on the left represents the toggle vector resulting from the button presses of the corresponding column on the right-hand matrix. This shows us that the result of each button press is independent from all other button presses. Therefore, no button needs be pressed more than once in order to complete the game.

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

As it has been shown that we can use linear algebra to represent certain aspects of the Lights Out game we can extend that further. We can use the equation $\vec{r} = \vec{p} + A\vec{x}$ to represent a game of Lights Out. $\vec{r}$ is the end vector, or the desired result. In the case of the Lights Out game this is always a vector of all zeroes. $\vec{p}$ is the starting vector, or the initial configuration on the Lights Out board. A represents the 25x25 matrix whose columns represent the result of each button press. $\vec{x}$ represents the button pattern required to get from $\vec{p}$ to $\vec{r}$.

By rearranging this equation, it can be written as $A\vec{x} = \vec{r} - \vec{p} = \vec{b}$. $\vec{b}$ is the difference between the starting and ending configurations. The desired ending configuration ($\vec{r}$) is a vector of all zeroes, therefore $\vec{b} = -\vec{p}$. As this problem is a binary arithmetic problem, if you add a matrix to itself the result is a matrix of all zeroes. For this reason, $-\vec{p}$ and $\vec{p}$ are the same ($\vec{p} - \vec{p} = 0$ and $\vec{p} + \vec{p} = 0$). This is because if you apply a button pattern twice it is the same as not applying it at all. Therefore $\vec{b} = \vec{p}$ which means $\vec{b}$ is represents the initial configuration of the Lights Out board. By finding the inverse of A (A^{-1}) a solution to the game can be found by multiplying $\vec{b}$ by A^{-1}. The reason this works is because A^{-1} shows us the button pattern to turn off each of the first 23 lights individually. When we multiply A^{-1} by the initial configuration we are essentially finding the button patterns to turn off each light in the initial configuration and performing addition in modulo 2 (or the XOR operation) on those patterns.

Unfortunately, when the Gaussian elimination is performed on the matrix A it is found that the matrix cannot be inverted. However, by performing a Gauss-Jordan elimination on the matrix alongside the 25x25 identity matrix we get the following result:

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

The result of the Gauss-Jordan elimination is a pseudo-inverse matrix which can be seen on the left. An identity for a 23x23 matrix can be seen within the 25x25 matrix. This shows us that the matrix has a rank of 23. The rank of an m x n matrix can be found by first finding the row rank and the column rank for the matrix. Whichever rank is the lowest of these two is the rank of the matrix. The row rank is the maximum number of linearly independent rows in the matrix while, the column rank is the maximum number of linearly independent columns in the matrix. In the case of the above matrix the row rank is 23 while the column rank is 25. Therefore the rank of the matrix is 23.

In the context of the Lights Out game, this means that we can turn any of the first 23 lights on or off by pressing the corresponding button pattern from the matrix on the right. It also means that these 23 buttons can be used to complete the Lights Out game, regardless of the state of the last 2. Therefore, if we reduce the initial configuration to a vector of length 23 ($\overrightarrow{b_r}$) and reduce the matrix A to a 23x23 matrix (A_r), a solution to the puzzle ($\overrightarrow{x_r}$) can be found by multiplying the initial configuration $\overrightarrow{b_r}$ by A_r^{-1}. $\overrightarrow{x_r}$ can then be extended back to a vector of length 25 by appending two zeroes to it.

As there are only 23 buttons required to solve the puzzle, there are 2^{23} solvable configurations. The total number of possible configurations is 2^{25}. This means that for a random initial configuration there is a 1 in 4 chance of it being solvable.

2.3.2 Solving the Puzzle with the Optimal Solution

As well as showing how to find a solution to the Lights Out Game, the matrix also provides other useful information. It can be seen that the two bottom lines of the matrix are entirely 0. This means that if the corresponding button patterns from the matrix on the right are pressed, the board would be in the same configuration as before the button pattern was started. These button patterns are known as the quiet patterns. They are:

$$(0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0)$$

$$(1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1)$$

There is a third quiet pattern which is calculated by combining the original two quiet patterns using the XOR operation, or addition in modulo 2.

$$(1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1)$$

The reason this is also a quiet pattern is because if you apply the original two quiet patterns to a given initial configuration the result on the board will be the same as the initial configuration. Applying both of these quiet patterns is the same as applying the combination of the two patterns which gives us our third quiet pattern.

These quiet patterns are shown in figures 2.6-2.8 below as they would be shown on a Lights Out board.

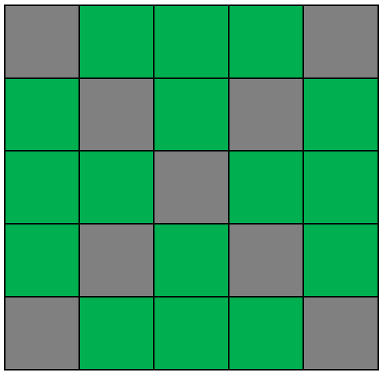


Figure 2.6

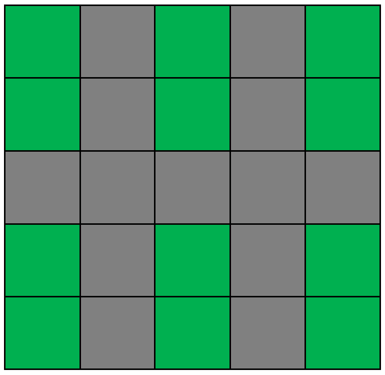


Figure 2.7

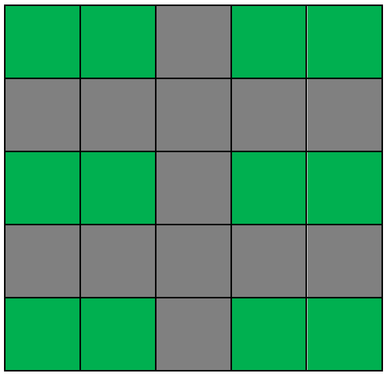


Figure 2.8

By applying each of these quiet patterns to the pattern $\vec{x}$ calculated previously, four patterns that solve the Lights Out game can be obtained. Of these four patterns, the one that requires the least number of moves (i.e. has the least number of 1s in the vector) is the solution to the original configuration with the least number of moves.

These quiet patterns can also be used to test whether an initial configuration is solvable. In a symmetric game a light pattern is solvable if, and only if, the number of lights in common with each quiet button pattern is even. This is because each button will affect an even number of lights in a quiet pattern. Therefore, an even number of number of lights must be switched on in each quiet pattern for an initial configuration to be solvable.

The quiet patterns can also show the highest number of moves possible for an optimised solution. This is done by representing the quiet patterns as shown in the figure 2.9.

c	b	a	b	c
a	d	a	d	a
b	b	d	b	b
a	d	a	d	a
c	b	a	b	c

Figure 2.9

The three quiet patterns are given by squares a and b, a and c, and b and c. The squares marked with d are not part of any quiet pattern. As it is known that no button need be pressed twice in order to solve a given configuration, wit can be said that the number of moves required to solve a given configuration can be represented as:

$$A + B + C + D$$

A, B, C, and D represent the number of buttons pressed of the sets a, b, c, and d respectively. The ranges for these variables are therefore:

$$0 \leq A, B \leq 8$$

$$0 \leq C \leq 4$$

$$0 \leq D \leq 5$$

By applying the quiet pattern made up of a and b, and assuming that this provides an optimal solution, the equation can now be written as:

$$(8 - A) + (8 - B) + C + D$$

As it has been assumed that this is the optimal solution, the following statement can be made:

$$(8 - A) + (8 - B) \geq A + B$$

$$A + B \leq 8$$

By repeating this process for the other two quiet patterns the following two statements can also be made:

$$A + C \leq 6$$

$$B + C \leq 6$$

By maximising the original equation, the variables A,B, and C are 4, 4 and 2 respectively. As D is not part pf a quiet pattern in the case of maximising the equation it is assumed that all the buttons are pressed making $D = 5$. This means that the result of the equation when all the variables are maximised is 15. Therefore, the maximum amount of moves an optimised solution can require is 15.

2.3.3 Working Example

In order to best illustrate the process of using the $A\vec{x} = \vec{b}$ function to find the solution to a Lights Out configuration, a working example will be outlined from beginning to end. Take the following initial configuration shown in figure 2.10.

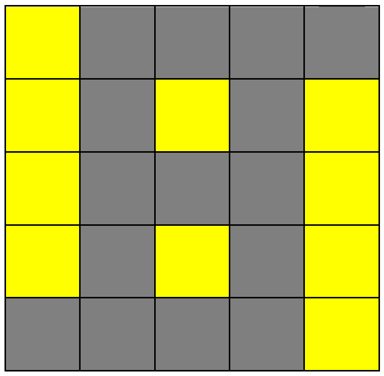


Figure 2.10

This can be represented as a matrix where 1 represents a light that is lit and 0 represents a light that is not:

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

This configuration is represented as the matrix $\vec{b}$ in the equation and is therefore converted into a vector. This vector along with the inverse of matrix A calculated previously are reduced down to a vector of length 23($\overrightarrow{b_r}$) and a 23x23 ($A^{-1}{}_r$) matrix respectively. They are the multiplied together to find the reduced solution

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

The resulting vector when the above matrix and vector are multiplied is:

(0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0)

After appending two zeroes to this vector it can be converted back to a 5x5 matrix

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

This provides us with one of the possible solutions to the game. In order to find the solution that requires the least number of moves, the above solution is combined with each of the quiet patterns by means of the XOR operation. This is done because we know the above button pattern will turn all the lights of the initial configuration off and we know that applying a quiet pattern will not affect the configuration of the board in any way. Therefore, a combination of the two will result in all the lights of the initial configuration being turned off. By combining the two button patterns for each of the quiet patterns, it is possible to find a solution that requires less moves then the above matrix. When we carry out this process we get the following solutions:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Comparing these four button patterns and by counting the number of moves (number of 1s) required to complete the game, it can be seen that the original matrix and the central matrix above both only require 8 moves to solve the puzzle. Either of these button patterns can be outputted as a solution to the game.

By converting these matrices into button patterns, the following two patterns (figures 2.11 and 2.12) are found to be the solutions with the least number of moves.

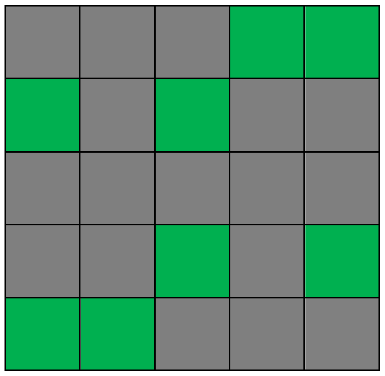


Figure 2.11

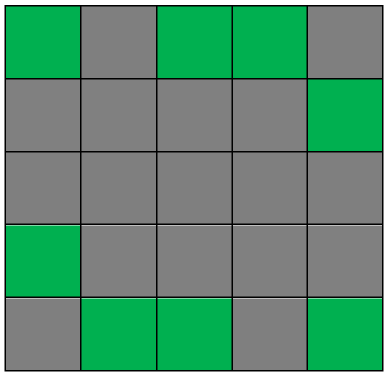


Figure 2.12

3 Method

A Haskell console application was created in order to show that Functional Programming is a good tool for solving these types of Linear Algebra problems. In order to do this the goal of the program was to solve Lights Out, taking in any configuration, checking whether or not it is solvable, and if it was, returning the solution that requires the least number of moves to complete the game. The process for carrying this out can be split up into four parts:

1. Performing Gauss-Jordan
2. Checking for Solvability
3. Calculating the Solution
4. Calculating the Optimal Solution

3.1 Performing Gauss-Jordan

The pseudo inverse matrix is calculated by performing a Gauss-Jordan elimination on the Lights Out matrix. This can be broken up into two sections, converting the matrix to row echelon form and converting the matrix to reduced row echelon form. A number of algorithms for carrying this out were attempted. One of these algorithms involved converting the matrix to row echelon form using a rowEchelon function and then rotating the outputted matrix 180 degrees and calling the rowEchelon function a second time with the new matrix as the input. The outputted matrix would then be in reduced row echelon form. This method was successful at finding the inverse of a matrix but was unsuccessful at finding the pseudo inverse of matrix.

Therefore a new method for performing Gauss-Jordan was devised.

When executing this function the Lights Out Matrix and the 25x25 identity matrix are inputted as a tuple in the first (left-hand matrix) and second (right-hand matrix) positions respectively. The outputted tuple should then have the identity (or pseudo identity) as the first item of the tuple, and the inverted matrix as the second position of the tuple.

In order to convert the matrix to row echelon form the following code snippet was implemented:

```
rowEchelon :: (Matrix, Matrix) -> (Matrix, Matrix)

rowEchelon ([], _) = ([], [])

rowEchelon (_, []) = ([], [])

rowEchelon (m1, m2) = ([head newM1] ++ newM1Tail, [head newM2] ++ newM2Tail)

    where    (newM1Tail, newM2Tail) = rowEchelon (removeFirstColumn $ tail newM1, tail
newM2)

             (newM1, newM2) = xorWithFirstRow (rowStartWithOne (m1, m2))
```

This function is a recursive function. It first checks to see if the first row of the left-hand matrix begins with a one. If this is not the case, then that row is moved to the bottom of the matrix and the next row is checked. This is repeated until a row beginning with a one is found. This row is then added (in modulo 2) to all the other rows that also begin with one. All operations that are performed on the left-hand matrix are performed on the corresponding rows of the right-hand matrix.

The function is then recursively called, with the input being a tuple of the left-hand matrix with the first row and first column removed, and the right-hand matrix with the first row removed. The result of this function call is added to the first rows of both matrices calculated in this instance of the function call. The recursive call is ended once all the rows have been calculated.

As this use case requires a pseudo inverted matrix, the end points that would normally be reached will not be reached. The desired result has two all zero rows. for this reason, the function that finds a row with a leading row and puts it at the top of the matrix, also ensures that there is a row that starts with one present. If not, then the matrix is returned as is. This ensures that the pseudo row echelon form can be calculated in the case of non-invertible matrices.

In order to convert the matrix to reduced row echelon form the following code snippet was implemented:

```
reducedRowEchelon :: (Matrix, Matrix) -> (Matrix, Matrix)

reducedRowEchelon ([], _) = ([], [])

reducedRowEchelon (_, []) = ([], [])

reducedRowEchelon (m1, m2) = if numAllZeroRowsEqualsNumColumns m1 == True || matrixIsEmpty 
m1 then (m1, m2) else (addColumnToMatrixLeft m1Column1 m1Part3, m2Part3)

    where    (m1Part3, m2Part3) = reducedRowEchelon (m1LessColumn1, m2Rejoin)

             m1Column1 = getFirstColumn m1Part2

             m1LessColumn1 = removeFirstColumn m1Rejoin

             m1Rejoin = m1Part2 ++ m1Part1Bottom

             m2Rejoin = m2Part2 ++ m2Part1Bottom

             (m1Part2, m2Part2) = xorWithLastRow (m1Part1Top, m2Part1Top)

             (m1Part1Top, m2Part1Top) = getRowUpToLastLeadingOne (m1, m2)

             (m1Part1Bottom, m2Part1Bottom) = getRowAfterLastLeadingOne (m1, m2)
```

This function is also a recursive function, and takes in a tuple as an input where the first item is the left-hand matrix and the second is the right-hand matrix. This function finds the last row that begins with a one in the left-hand matrix and adds it to all the other rows that also begin with a one. Any operation on left-hand matrix is performed on the corresponding rows of the right-hand matrix. The function then recursively calls itself with the input being the left-hand matrix with the first column removed, and the newly calculated right-hand matrix as the elements of the tuple. The result of the function is a tuple of the left-hand matrix returned from the recursive call added to the previously removed first column, and the resulting right-hand matrix from the recursive call.

The end point of the recursive call occurs when the left-hand matrix is empty. However, as this use case requires the calculation of pseudo inverse matrices, there is another check done in order to ensure the desired result. When the function is called, it initially checks if the number rows in the left-hand matrix that are entirely zeros is the same as the number of columns remaining. If this is the case, the inputted tuple is outputted as the result of the function. This acts as an end point for the cases in which the matrix is not invertible.

Now that both the row echelon form and the reduced echelon form functions have been implemented, the matrix inverse function can be implemented. The following code snippet shows how this was carried out:

```
matrixInverse :: (Matrix, Matrix) -> (Matrix, Matrix)
matrixInverse ([], _) = ([], [])
matrixInverse (_, []) = ([], [])
matrixInverse (m1, m2) = (m1Step3, m2Step5)
    where  m2Step5 = xorMatrixRowsWithVectorsWithSecondLastOne (last (init m2Step4)) (init
(init m2Step4)) ++ [last (init m2Step4)] ++ [last m2Step4]
           m2Step4 = xorMatrixRowsWithVectorsWithLastOne (last m2Step3) (init m2Step3) ++
[last m2Step3]
           (m1Step3, m2Step3) = reducedRowEchelon (m1Step2, m2Step1)
           m1Step2 = addZeroesToMatrix m1Step1 (length m1)
           (m1Step1, m2Step1) = rowEchelon (m1, m2)
```

The first step is to get the matrix in row echelon form using the previously implemented rowEchelon function. Once this is done zeroes must be added to the left-hand matrix before the first leading one as they were removed in the process of calculating the row echelon form. The reducedRowEchelon function can then be called. The outputted right-hand matrix should be the inverted (pseudo inverted) matrix.

This is enough for matrices that are fully invertible, however the use case for solving Lights Out requires a few more steps. The calculated pseudo inverse at this point does not have two columns of all zeroes to the right of the 23x23 matrix. In order to rectify this, without affecting the pseudo identity in the left-hand matrix, the bottom row is added to all rows ending in a one. The second to last row is then added to all rows whose second to last number is a one. This returns the desired matrix.

3.2 Checking for Solvability

Now that the pseudo inverse of the Lights Out matrix has been calculated, the quiet patterns can be obtained. These can then be used to test if a given configuration is solvable or not.

The below code snippet shows the function used to acquire the two quiet patterns from the inverted Lights Out matrix:

```haskell
getQuietPatterns :: (Vector, Vector)

getQuietPatterns = (last (init aInverse), last aInverse)

    where (pseudoIdentity, aInverse) = matrixInverse (lightsOutMatrix, identity 25)
```

This function calls the matrixInverse function to calculate the inverse matrix. It takes the two bottom rows of the right-hand matrix and returns them both in a tuple.

This function is used in the below code snippet to detect if a given configuration is solvable:

```haskell
solvable :: Vector -> Bool

solvable initialConfig = if dot1 == 0 && dot2 == 0 then True else False

    where   dot1 = dotProduct initialConfig qp1

            dot2 = dotProduct initialConfig qp2

            (qp1, qp2) = getQuietPatterns
```

Dot product is performed twice, once for each quiet pattern. The dot product is carried out on the initial configuration and one of the quiet patterns. If the result of both of these operations is zero, then the configuration is solvable.

3.3 Calculating the Solution

In order to calculate the solution, the following code snippet is used:

```
solve :: Matrix -> Maybe Matrix

solve m = if solvable v == False

    then Nothing

    else  Just  (vectorToMatrix  $  getLeastMovesSolution  $  solutionsAndMoveCount  $
createSolutionsList  $  appendTwoZeroes  $  multiplyByVector  (reduceMatrixTo23  aInverse)
(reduceVectorTo23 v))

    where   (pseudoIdentity, aInverse) = matrixInverse (lightsOutMatrix, identity 25)

            v = matrixToVector m
```

The function first converts the matrix to a vector and tests whether or not it is solvable. If it is not solvable, the function returns ‘Nothing’. If it is solvable the vector is reduced to a vector of length 23 and multiplied with the inverse matrix function calculated by the matrix inverse function. This matrix is reduced from a 25x25 matrix to a 23x23. The result of this multiplication then has two zeros appended to it. This gives a solution to the inputted configuration in vector form. The function then continues to find the optimal solution which is described below. The result is then converted back to a matrix and returned.

3.4 Calculating the Optimal Solution

Now that a solution has been found, the quiet patterns can be used to find the optimal solution. Looking at the code snippet in the previous section, a number of functions were used to find the optimal solution. These functions are:

- createSolutionsList
- solutionsAndMoveCount
- getLeastMovesSolution

The code snippet below shows the implementation of the createSolutionsList function:

```
createSolutionsList :: Vector -> [Vector]

createSolutionsList v = [v, (v `xorVectors` qp1), (v `xorVectors` qp2), (v `xorVectors`
qp3)]

    where   (qp1, qp2) = getQuietPatterns

            qp3 = qp1 `xorVectors` qp2
```

This function creates a list of four solutions. This is done by taking the previously calculated solution as an input, and adding each of the three quiet patterns to it. These three patterns are then added to a list along with the original solution and returned.

The code snippet below shows the implementation of the solutionsAndMoveCount function:

```
solutionsAndMoveCount :: [Vector] -> [(Vector, Int)]

solutionsAndMoveCount [] = []

solutionsAndMoveCount (v:vs) = (v, countMoves v) : solutionsAndMoveCount vs
```

This function takes in a list of solutions and calculates the moves required to solve each of them. This is done by calling the countMoves function which counts the number of ones in each solution vector. A list of tuples is then created where the first element is a solution and the second element is the number of moves required to complete that solution. This list is then returned as the result of the function.

The code snippet below shows the implementation of the getLeastMovesSolution function:

```
getLeastMovesSolution :: [(Vector, Int)] -> Vector

getLeastMovesSolution list =  fst $ head $ sortBy (compare `on` snd) list
```

This function takes in a list the previously calculated list of tuples. It then returns the solution that has the least number of moves associated with it. If the least number of moves is associated with more than one solution, the first solution in the list with that number of moves will be returned.

4 Result

Using the functions described in the method section, the program is now able to solve Lights Out. To show this we take the solvable configuration shown in figure 4.1.

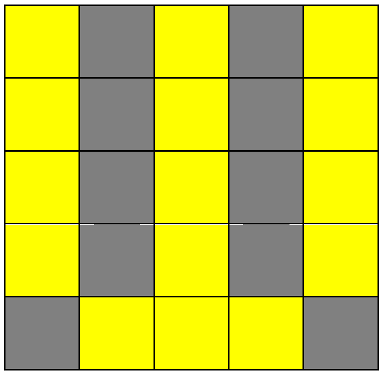


Figure 4.1

This configuration is passed as an input to the 'solve' function as shown below

```
ghci> solve [[1,0,1,0,1],[1,0,1,0,1],[1,0,1,0,1],[1,0,1,0,1],[0,1,1,1,0]]
```

The result if this operation is

```
Just [[1,0,0,0,1],[0,1,1,1,0],[1,0,0,0,1],[0,0,0,0,0],[0,0,1,0,0]]
```

This equates to using the button pattern shown in figure 4.2 to solve the initial configuration.

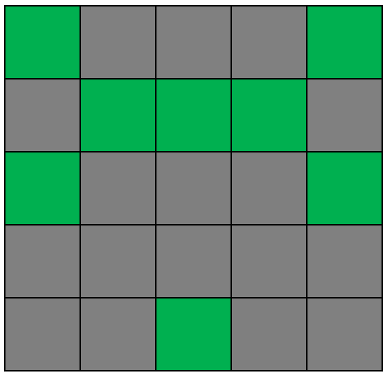


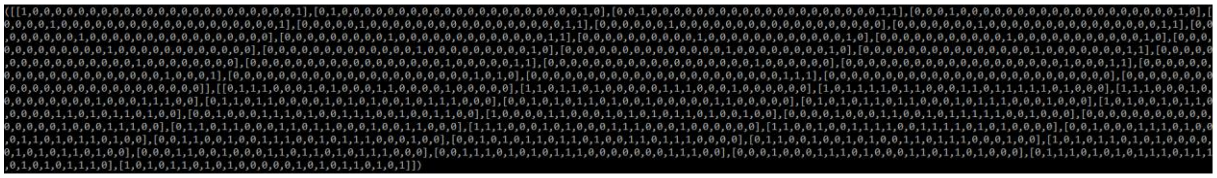
Figure 4.2

This pattern requires 8 moves to solve the puzzle. If the results of each of the functions described in the method section is investigated for this use case, it can be seen how this result was reached.

4.1 Matrix Inverse

When finding solutions to the classic 5x5 Lights Out game, the solution to this function is always the same. This is because the same Lights Out matrix is used as the input.

The out put of this function is shown below:

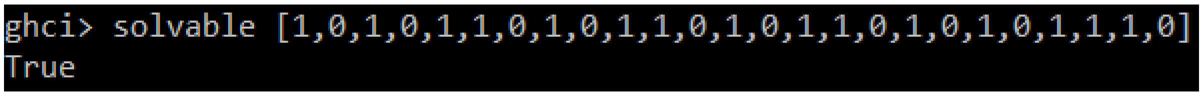


```
([[1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],[0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0],[0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],[0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0],[
0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],[0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],[0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],[0,0
,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],[0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0],[0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,0],[0,0,0,
0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,1],[0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,1],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,1],[0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1],[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]],[[0,1,1,1,0,0,0,1,0,1,0,0,0,1,1,0,0,0,0,1,0,0,0,0,0],[1,1,0,1,1,0,1,0,0,0,0,0,1,1,1,0,0,0,1,0,0,0,0,0,0],[1,0,1,1,1,1,0,1,1,0,0,0,1,1,0,1,1,1,1,1,0,1,0,0,0],[1,1,1,0,0,0,1,0,
0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,0,0],[0,1,1,0,1,1,0,0,0,0,1,0,1,0,1,0,0,1,0,1,1,1,0,0,0],[0,0,1,0,1,0,1,1,0,1,0,0,1,0,0,0,0,0,1,1,0,0,0,0,0],[0,1,0,1,0,1,1,0,1,1,0,0,0,1,0,1,1,1,0,0,0,1,0,0,0],[1,0,1,0,0,1,0,1,1,0
,0,0,0,0,1,1,0,1,0,1,1,0,1,0,0],[0,0,1,0,0,0,1,1,1,0,1,0,0,1,1,1,0,0,1,0,0,1,1,0,0],[1,0,0,0,0,1,1,0,0,0,1,0,1,0,1,0,1,1,0,1,0,0,1,0,0],[0,0,0,0,1,0,0,0,1,1,0,0,1,0,1,1,1,1,1,0,0,1,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,1,0,0,0,1,1,1,0,0],[0,1,1,0,1,1,0,0,0,1,1,0,1,1,0,0,0,1,0,0,1,1,0,0,0],[1,1,1,0,0,0,1,0,1,0,0,0,1,1,1,0,0,0,1,0,0,0,0,0,0],[1,1,0,0,1,0,0,1,1,1,1,0,0,1,1,1,1,0,1,0,1,0,0,0,0],[0,0,1,0,0,0,1,1,1,0,1,0,0
,0,1,1,0,1,0,1,1,0,1,0,0],[0,0,1,1,0,0,1,0,0,1,1,1,0,0,1,0,1,1,1,0,0,0,1,0,0],[0,0,1,0,1,0,1,1,0,1,1,0,1,0,0,1,1,0,1,1,1,0,0,0,0],[0,1,1,0,0,1,0,0,1,0,1,0,0,1,1,0,1,1,1,0,0,0,1,0,0],[1,0,1,0,1,1,0,1,0,1,0,0,0,0,
0,1,0,1,0,1,1,0,1,0,0],[0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,1,0,1,1,1,0,0,0],[0,0,1,1,1,0,1,0,1,0,1,1,1,0,0,0,0,0,0,0,1,1,1,0,0],[0,0,0,1,0,0,0,1,1,1,0,1,0,0,0,1,1,0,1,1,0,1,0,0,0],[0,1,1,1,0,1,0,1,0,1,1,1,0,1,1,1
,0,1,0,1,0,1,1,1,0],[1,0,1,0,1,1,0,1,0,1,0,0,0,0,0,1,0,1,0,1,1,0,1,0,1]])
```

This output is the two matrices expected from this equation. The reduced row echelon matrix in the first element of the tuple and the inverted matrix in the second element of the tuple. The inverted matrix can be reduced to a 23x23 matrix in order to solve Lights Out.

4.2 Checking for Solvability

In order to check if this configuration is solvable, it is inputted into the 'solvable' function as shown:



```
ghci> solvable [1,0,1,0,1,1,0,1,0,1,1,0,1,0,1,1,0,1,0,1,0,1,1,1,0]
True
```

In this case it returns 'True', but in the case of an unsolvable function it would return false.

4.3 Calculating the Optimal Solution

In order to show that the solution returned by the ‘solve’ function is indeed the most optimal, it can be passed into the ‘createSolutionsList’ function in order to see all 4 potential optimal solutions. The solution is passed in as a vector as shown:

```
ghci> createSolutionsList [1,0,0,0,1,0,1,1,1,0,1,0,0,0,1,0,0,0,0,0,0,0,1,0,0]
```

This returns the following output:

```
[[1,0,0,0,1,0,1,1,1,0,1,0,0,0,1,0,0,0,0,0,0,0,1,0,0],

[1,1,1,1,1,1,1,0,1,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0],

[0,0,1,0,0,1,1,0,1,1,1,0,0,0,1,1,0,1,0,1,1,0,0,0,1],

[0,1,0,1,0,0,1,1,1,0,0,1,0,1,0,0,0,0,0,0,1,1,1,1,1]]
```

These equate to the following four solutions shown in figures 4.3-4.6.

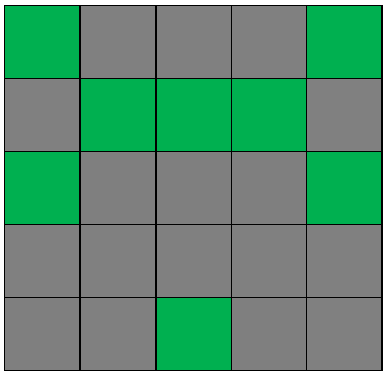


Figure 4.3

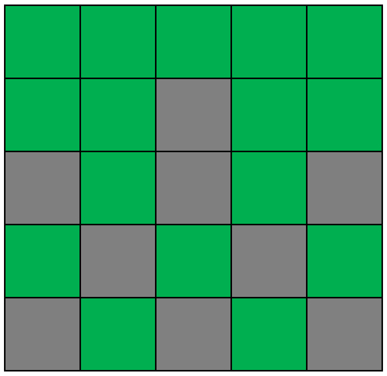


Figure 4.4

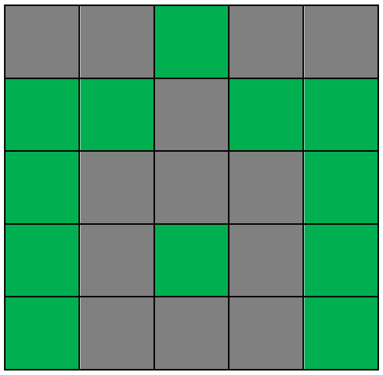


Figure 4.5

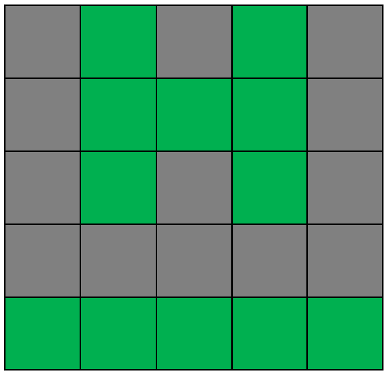


Figure 4.6

When these four button patterns are all applied to the initial configuration they all solve the puzzle. When these four solutions are compared it can be seen that the solution originally returned by the 'solve' function is indeed the most optimal solution. This solution only requires 8 moves to solve the puzzle whereas the other three require 16, 12 and 12 moves respectively.

4.4 Unsolvable Configurations

In the case of an unsolvable configuration (such as the one shown in figure 4.7), the 'solvable' function will return false and the 'solve' function will return 'Nothing'

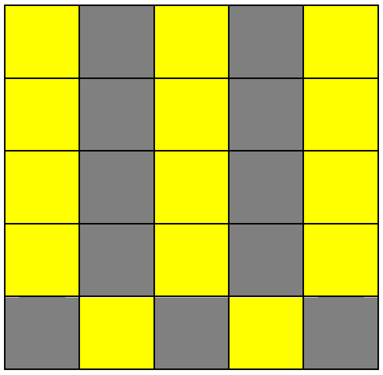
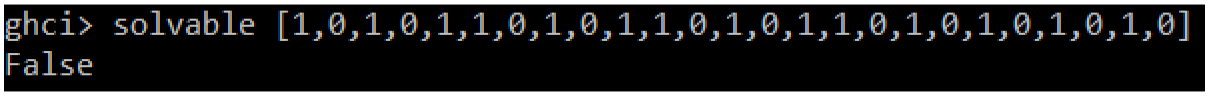
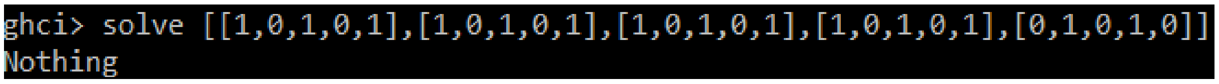


Figure 4.7

The outputs of the 'solvable' and 'solve' function for the above configuration are:



```
ghci> solvable [1,0,1,0,1,1,0,1,0,1,1,0,1,0,1,1,0,1,0,1,0,1,0,1,0]
False
```



```
ghci> solve [[1,0,1,0,1],[1,0,1,0,1],[1,0,1,0,1],[1,0,1,0,1],[0,1,0,1,0]]
Nothing
```

5 Discussion

From the results drawn from this research, there are a few things to discuss. This discussion encompasses both the advantages that functional programming brings to solving linear algebra problems, and where to next take this research.

5.1 Advantages of Functional Programming

During the implementation of the Lights Out solver application a number of advantages brought by using a functional programming language was noted. Most of these advantages are as a result of the characteristics of functional programming discussed in Chapter 2.

Much like functional programming functions, linear algebra functions (and most mathematic functions) are pure functions. The same input will always result in the same output. This shows the benefits of functional programming for solving linear algebra problems as they both have this characteristic. A similar point can be made about the functional programming having referential transparency. At any point in a linear algebra equation or a functional programming function, a variable can be replaced by its actual value without affecting the final result.

Another advantage that functional programming provides is the manner in which it handles lists. In this project, vectors were represented by a list of integers, and matrices were represented by a list of vectors (or a list of lists of integers). This method of representing vectors and matrices allowed for the use of recursion to handle the end points of different vector and matrix operations. One example of using Haskell's list methods in this way would be when the zipWith function is used to add each element of two vectors together in the xorVectors function (Appendix A1.1). Another example that uses recursion would be the checkForAllLeadingZeroes function (Appendix A1.2). This function checks the first vector of a matrix for a leading zero and then calls itself with the rest of the matrix as an input. The recursive calling ends once the input matrix is empty, with 'True' as the return value. The AND operation is performed on all the returned values of the function, giving the desired result of 'True' if the matrix contains vectors that all start with zero and 'False' otherwise. One of the limitations of this, however, is that on some occasions the entire list will need to be iterated through even if the desired case is found on the first iteration. If this were the case in an imperative language, the result would be returned as soon as the desired case was found.

The use of higher order functions also provides an advantage to using functional programming when solving problems in linear algebra. This allows for the use of mapping functions which help to implement elegant solutions to problems. One example of this is the removeFirstColumn function (Appendix A1.3) which takes in a matrix and returns the matrix with the first column removed. This is done by mapping the tail function onto each row of the matrix. The ability to pass the tail function as an argument of a function provides a very elegant implementation of the function.

Possibly one of the biggest advantages functional programming provides is lazy evaluation. The use of sortBy in the getLeastMovesSolution function (Addressed in chapter 3) is a good example of this in this project. As only the list is sorted based on the first element, only the elements that will appear before the first element of the function need to be evaluated, thus reducing the overall computation time of the program.

It can be seen from the above example that functional programming offers many advantages to solving linear algebra problems that traditional, imperative languages do not. Although these languages do offer their own advantages, it is fair to say that there is a case to be made for the use of functional programming.

5.2 Future Works

If this project were to be taken further there are a number of next steps that could be taken.

One of the first things to do would be to refactor the code for efficiency. Due to the time limits associated with the current research, the code may not have been implemented in order to produce the best performance. Each function can be looked at to ensure that the algorithm is being carried out in the least amount of operations. As well as that, the memoization library[20] could be used in order to save the result of some repeated functions. For example, the matrixInverse function is called a number of times with the same inputs. This result could be cached using this library in order to reduce overall computation time. Some functions are implemented specifically to solve the classic Lights Out, such as the reduceMatrixTo23 function (Appendix A1.4). These functions could be generalised to work for other use cases, in this case changing the function reduceMatrix and adding a parameter that dictates to what size it is reduced to.

Another valuable addition to this project would be a front end. An application much like that found on Jaap's puzzle page[3] would allow for a better demonstration of the application built during this project. Jaap's application is a Lights Out game built using

JavaScript. A similar application could be built using Haskell where the calculations for the optimal solution can be made using the functions implemented in this project.

As seen in the background research underpinning this project, there are a number of widely used operations in the field of Linear Algebra, many of which are not used in the solving of Lights Out. Two examples are would be cross product and determinant. A more generalised library can be built that also implements these functions as well as those covered in this project. Storage mapping functionality could also be added to this library in order to make it easier to retrieve or edit individual values in a vector or matrix. It should also be noted that the functionality required for solving Lights Out requires operations to be carried out in modulo 2. This means that for a more generalised library the functions implemented in this project will need to be refactored to handle modulo 10 operations. The code could also be refactored to use better naming convention as well has being well documented in order for the library to be more user friendly for anyone who wishes to import it into their projects.

The application built during this project could also be updated to include solutions to different variations of Lights Out. There are a number of variations of Lights Out that are addressed in Jaap's page[3] as well as in David Joyner's book[21]. These include a version of the game that has three states instead of just on and off. Another version has the direct diagonal neighbours of a button toggle when a button is pressed rather than direct vertical and horizontal neighbours. As well as that, the board can be changed to be a different size rather than the classic 5x5 board. In most of these cases, excluding the three state game, the game is solved by changing the A matrix used in the $A\vec{x} = \vec{b}$ equation. The new matrix should math the rules of the new variation, for each button the corresponding column of that matrix should be the toggle vector for that button. This project had planned on implementing functionality to solve a variation of Lights Out whereby the board was boundless. This means that if a button on the left most side of the board is pressed, the button on the right most side on the same column is toggled as shown in figures 5.1 and 5.2 below.

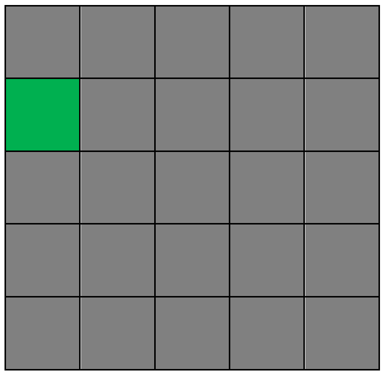


Figure 5.1

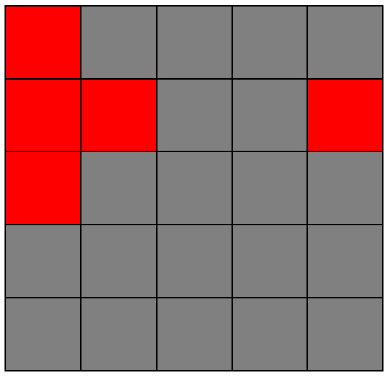


Figure 5.2

Unfortunately given the time constraints, this functionality was not implemented. However, some first steps were made in doing so. The matrixInverse function was executed using the following matrix.

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

Each column of this matrix is the vector representation of the lights that will toggle as a result of each button press, just as in the classic version of Lights Out. The result of the function was promising as it had the desired number if all zero rows. Further research is required in order to fully solve this variation of Lights Out. This will involve ensuring the resulting pseudo inverse is indeed correct, identifying the quiet patterns, establishing a solvability check for a given configuration, solving a configuration and finding the optimal solution for that configuration.

6 Conclusion

The course of this project appears to have substantiated that functional programming certainly has its place when it comes to solving linear algebra problems. This project has shown that functional programming can be used to solve the Lights Out puzzle. In doing this, the program needed to be able to carry out a Gauss-Jordan operation in order to invert a matrix. Not only that, but the program needed to handle the case where a matrix was not invertible, and a pseudo inverse needed to be found. The research carried out by the author has shown that this was indeed within the capabilities of a function programming language, more specifically Haskell. In doing this, it has also been shown that functional programming is capable of solving linear algebra problems and can provide advantages over imperative languages such as C or Java. It could even be a good platform for building mathematical software such as MATLAB, Mathematica and SageMath, which makes functional programming a viable area for further exploration.

7 Bibliography

[1] J. Mulholland. (2016). *Math 302: Lights Out*. Available: http://www.sfu.ca/~jtmulhol/math302/puzzles-lo.html [Accessed: 09 Nov 2017]

[2] J. Mulholland, "Permutation Puzzles: A Mathematical Perspective," *Simon Fraser University,* 2016.

[3] J. Scherphuis. (2014). *Jaap's Puzzle Page: Lights Out*. Available: https://www.jaapsch.net/puzzles/lights.htm [Accessed: 06 May 3018]

[4] M. Anderson and T. Feil, "Turning lights out with linear algebra," *Mathematics Magazine,* vol. 71, no. 4, pp. 300-303, 1998.

[5] T. Ball, "Paths between imperative and functional programming," *SIGPLAN Notices,* vol. 34, no. 2, pp. 21-5, 02/ 1999.

[6] J. Hughes, "The Design of a Pretty-printing Library," presented at the Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text, 1995.

[7] S. Thompson, "Formulating Haskell [functional programming language]," in *Proceedings of the 1992 Glasgow Workshop on Functional Programming, 6-8 July 1992*, Berlin, Germany, 1993, pp. 258-68: Springer-Verlag.

[8] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," presented at the Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, 2000.

[9] R. Page, "Functional programming, and where you can put it," *SIGPLAN Notices,* vol. 36, no. 9, pp. 19-24, 2001.

[10] H. Doets, "The Haskell road to logic, maths and programming," *Texts in Computing,* 2012.

[11] S. Dolan, "Fun with semirings: A functional pearl on the abuse of linear algebra," in *2013 18th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, September 25, 2013 - September 27, 2013*, Boston, MA, United states, 2013, pp. 101-109: Association for Computing Machinery.

[12] A. S. Eriksson and P. Jansson, "An agda formalisation of the transitive closure of block matrices (extended abstract)," presented at the Proceedings of the 1st International Workshop on Type-Driven Development, Nara, Japan, 2016.

[13] F. Eaton, "Statically typed linear algebra in haskell," in *Haskell'06 - ACM SIGPLAN 2006 Haskell Workshop, September 17, 2006 - September 17, 2006*, Protland, OR, United states, 2006, vol. 2006, pp. 120-121: Association for Computing Machinery.

[14] A. Ruiz. (2018). *hmatrix-gsl: Numerical computation*. Available: https://hackage.haskell.org/package/hmatrix-gsl [Accessed: 04 May 2018]

[15] O. Kiselyov and C.-c. Shan, "Functional pearl: implicit configurations--or, type classes reflect the values of types," in *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, 2004, pp. 33-44: ACM.

[16] A. Ghitza and M. Westerholt-Raum, "HLinear: Exact Dense Linear Algebra in Haskell [arXiv]," *arXiv,* p. 12 pp., 05/09 2016.

[17] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Parallel tiled QR factorization for multicore architectures," *Concurr. Comput. : Pract. Exper.,* vol. 20, no. 13, pp. 1573-1590, 2008.

[18] G. Quintana-Ort *et al.*, "Programming matrix algorithms-by-blocks for thread-level parallelism," *ACM Trans. Math. Softw.,* vol. 36, no. 3, pp. 1-26, 2009.

[19] E. S. Quintana, G. Quintana, X. Sun, and R. v. Geijn, "A Note On Parallel Matrix Inversion," *SIAM J. Sci. Comput.,* vol. 22, no. 5, pp. 1762-1771, 2000.

[20] A. Bromage. (2014). *Memoization*. Available: https://wiki.haskell.org/Memoization [Accessed: 08 May 2018]

[21] D. Joyner, *Adventures in group theory: Rubik's Cube, Merlin's machine, and other mathematical toys*. Springer, 2008.

A1. Appendix

Appendix A1.1

```
xorVectors :: Vector -> Vector -> Vector

xorVectors v1 v2 = zipWith xor v1 v2
```

Appendix A1.2

```
checkForAllLeadingZeroes :: Matrix -> Bool

checkForAllLeadingZeroes [] = True

checkForAllLeadingZeroes (m:ms) = hasLeadingZero && checkForAllLeadingZeroes ms

    where hasLeadingZero = if head m == 0 then True else False
```

Appendix A1.3

```
removeFirstColumn :: Matrix-> Matrix

removeFirstColumn [] = []

removeFirstColumn m = map tail m
```

Appendix A1.4

```
reduceMatrixTo23 :: Matrix -> Matrix

reduceMatrixTo23 m = map reduceVectorTo23 newM

    where newM = take 23 m
```