Using Genetic Algorithms to Explore Optimal Strategies in the Dice Game Pig

Group 7A

014395
008809
023347
017713
018049
011679
031292
036157

University of Exeter

College of Engineering, Mathematics and Physical Sciences

Adviser - Dr Barrie Cooper

December 2016

Contents

1 Abstract

Pig is a two player jeopardy dice game that combines chance and strategy. The aim is to be the first player to reach a specified total score (usually 100). Each player’s turn consists of repeatedly rolling a die until they either hold or roll a 1. If the player rolls a 1, their turn score becomes zero and it is then the opponent's turn. If the player holds then the total turn score, the accumulated score from each roll, is added to the player's total score and it becomes the opponent's turn.

In this paper, we explore how different forms of genetic algorithms can be used to develop successful strategies for the game Pig. We use dynamic programming to recreate the ‘optimal strategy’ proposed by Todd Neller [NP04, p.32] in order to provide a comparison for our own strategies. Adjustments to the method also provide a means to compare strategies on the basis of a probabilistic win rate which presents itself as a natural choice of fitness function.

We implement two tournament based genetic algorithms and test them on a variety of initial populations; some of which were purely random and others containing very simplistic properties. We found that generally our genetic algorithms failed to produce long term improvements to the population of strategies. However, a third method was applied which divides the strategy into regions and we found this to be promising in improving the given strategies.

2 Introduction

2.1 Overview

Pig is a two player jeopardy dice game that combines chance and strategy. The aim is to be the first player to reach a specified total score (usually 100). Each player's turn consists of repeatedly rolling a die. After each roll the player has two options: roll again, or hold (decline to roll again). If the player rolls a 1, the player's turn score becomes zero and it is then the opponent's turn. When any number other than 1 is rolled, the number is added to the player's turn score and the player's turn continues. If the player decides to hold, the turn score points that have been accumulated are added to the player's total score and it becomes the opponent's turn.

There are many variations of the game; Piglet is a simplified version of Pig with the same rules but for a two-sided die (a coin), with a target score of 10 points. Each turn, a player repeatedly flips a coin until either a tail is flipped or the player decides to hold and thereby scores the number of consecutive heads flipped. In this variation, we associate the numbers 1 and 2 with flipping tails and heads respectively.

Consideration of risk and reward is essential when playing Pig as every time you roll you risk losing your accumulated turn score in the event that you roll a 1. However, not only must you consider your own total score and turn score, but it is also crucial to compare these elements to the target score and opponent's score. For example, we review one extreme where your total score is 0 and your opponent's score is 90, with a target score of 100. The likelihood of your opponent winning on the next turn is very high and so the best strategy in this scenario would be to try to obtain a high turn score, hence to roll.

A potential simple strategy for the game Pig is to hold at a certain turn score, similar to how the house holds at 17 in Blackjack. Intuitively, it seems there should exist a number to always 'hold at' in order to maximise your expected score on a particular turn.

Assuming that our aim is simply to maximise the expected outcome from a single turn then the best number to hold at is 20. This can be proved using the following reasoning.

Proof. Let T be the player's current turn score and $E\big[\mathrm{T}_{roll}\big]$ denote the expected value of the turn score after another roll. Given that we are using a fair six-sided dice this becomes,

$$E\big[T_{roll}\big] = \mathrm{T} + \left(-\tfrac{\mathrm{T}}{6} + \tfrac{2+3+4+5+6}{6} \right)$$

simplifying gives,

$$E\left[T_{roll}\right] = \mathrm{T} + \frac{20-\mathrm{T}}{6}$$

and so we expect the current turn score T to increase by a value of $\frac{20-T}{6}$ which is itself a monotonic decreasing function of T. Equating the expected increase to zero we can then solve to find the turn score T_{hold} which maximises the expected values of the new turn score after a roll.

$$\text{Expected Increase} = \frac{20-\mathrm{T}}{6} = 0$$
$$\implies T_{hold} = 20$$

This proves that holding at 20 in all cases will maximise a players expected outcome from a single turn. □

It is important to note however that whilst holding at 20 will maximise a players expected outcome on any given turn it won't necessarily maximise your chances of winning. This is because pig is a race to a given target as opposed to a quest to maximise your score over an unlimited number of turns. As the target score in Pig tends to infinity we could expect that the optimal number of points to hold at (if this was a players strategy) would tend to 20. In the standard version of Pig with a target score of 100 however it is not immediately obvious what the best 'hold at' strategy would be. In his paper, 'Practical Play of the Dice Game Pig', Neller demonstrates that the best number to hold at for a target of 100 is in fact 25.

In any case, while holding at 20 or 25 provides a reasonably good strategy for Pig, it is clear that these strategies are far from optimal as they fail to consider the opponent's score and how this might affect your chances of winning. In actual fact, finding the 'optimal strategy' for this seemingly simple dice game turns out to be exceedingly complex. It is worth mentioning at this point exactly how we define an optimal strategy as this in itself isn't necessarily obvious.

Definition 2.3.1. An optimal strategy is one whose win rate against *any* other strategy will tend to *at least* a half as the number of games played tends to infinity [1].

In this paper, we explore two very different approaches in order to develop successful strategies to the dice game Pig. First, we examine how genetic algorithms, which mimic the process of natural selection, can be used to evolve good strategies from a selection of different starting points. In addition to this, we build upon existing work by Neller [NP04] who presents a set of equations which he claims describe the optimal solution to the game. We employ dynamic programming techniques in order to solve this set of equations using an alternative method before examining the resulting strategy and comparing it to Neller's.

[1]Note that this is *not* equivalent to saying an optimal strategy is the best possible against all other strategies, but is rather a general purpose optimal.

2.2 Genetic Algorithms

Genetic algorithms attempt to mimic the real world process of evolution and are based on Charles Darwin's theory of evolution by natural selection where individuals that perform well are more likely to survive and thus have greater opportunity to reproduce and pass on their genetic material to future generations. [Mit98, p.4].

The basic processes performed by a genetic algorithm involve starting with an initial population and then applying three operators to produce the next generation of strategies. The three operators applied during genetic algorithms are:

1. Selection
2. Breeding
3. Mutation.

Selection:
The selection operator refers to a process by which better strategies are given greater opportunity to survive and reproduce and is intended to improve the overall quality of the population over time [BT95, p.9]. The majority of genetic algorithms make use of a fitness function in order to objectively measure the quality of each individual, while others employ a tournament system to select the best individuals in a given generation. (See step 1 in Figure 1)

Breeding:
Breeding is achieved via a crossover operator in which individuals exchange information and reproduce in a method that is similar to the way natural organisms sexually reproduce [Col99, p.11]. In this way, genetic material from the parents which contributes to their success will be passed on to the offspring which form the next generation. (See step 2 in Figure 1)

Mutation:
The mutation operator is used to randomly alter a small proportion of the genetic material for each of the individual offspring. The purpose of a mutation operator is to maintain a reasonable level of diversity amongst the population and prevent the algorithm from converging to a local optimum. (See step 3 in Figure 1)

One of the two overarching aims in this research is to develop a genetic algorithm which can successfully evolve an initially random population of strategies into coherent and successful strategies. In addition to this, we wish to see if our algorithms can improve upon initially 'good' strategies and approach an optimal strategy.

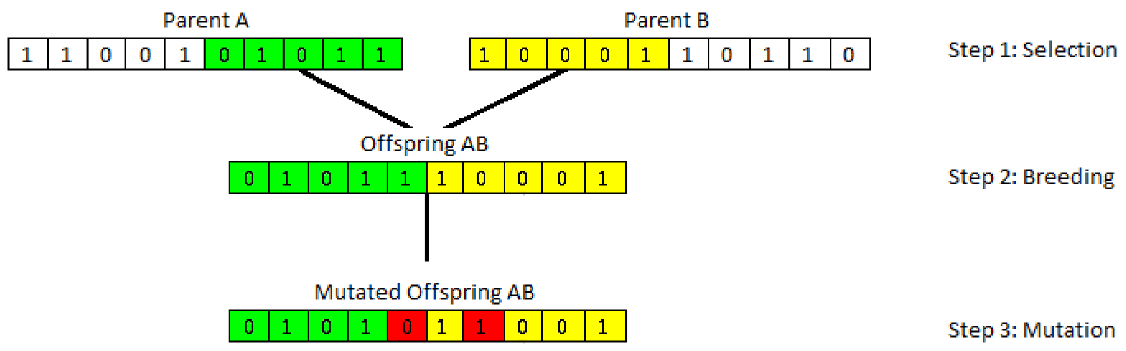


Figure 1: Binary code highlighting the selection, breeding and mutation operators in genetic algorithms.

We decided to concentrate on genetic algorithms as they have the ability to search large spaces where there may be many local optima [Col99, p.4]. Alternative methods include random or exhaustive searches. However, we decided these were not viable options due to the sheer number of possible strategies, $2^{1,000,000}$. Genetic algorithms are free of human preconceptions or bias and are, as a result, capable of producing solutions that are comparable to or better than equivalent human solutions.

However, genetic algorithms are not without their limitations. For one, there is no guarantee that our genetic algorithms will successfully find a global maximum and getting stuck within local maxima is a possible issue. Another potential limitation of using genetic algorithms is that it is hard to find a suitable combination of parameters, such as mutation rate.

2.3 Neller's Optimal Strategy for Pig

In his paper 'Optimal Play of the Dice Game Pig' [NP04], Neller presents a set of equations for the probability of a player winning from each game state given their strategy. These equations are derived in the following manner:

Let $P_{i,j,k}$ denote the player's probability of winning where i is the player's score, j is the opponent's score and k is the turn score. Due to the target score of 100, we have the following limitations for i, j and k: $0 \leq i, j < 100$ and $k < 100 - i$.

The probability of a player winning is denoted by

$$P_{i,j,k} = max[P_{i,j,k,roll}, P_{i,j,k,hold}]$$

[Eq 2.31]

where $P_{i,j,k,roll}$ and $P_{i,j,k,hold}$ are the probabilities of winning dependent on whether you roll or hold respectively, represented by:

$$P_{i,j,k,roll} = \tfrac{1}{6}[1 - P_{j,i,0}] + \tfrac{1}{6}\sum_{r=2}^{6} P_{i,j,k+r}$$

[Eq 2.32]

$$P_{i,j,k,hold} = 1 - P_{j,i+k,0}$$

[Eq 2.33]

[NP04, p.26]

In order to then find the optimal strategy from these equations one must now calculate the values for $P_{i,j,k,roll}$ and $P_{i,j,k,hold}$ for all possible game states. In each game state, the decision (roll or hold) which gives the higher probability of winning is then used within the optimal strategy. Neller achieved this using the method of value iteration. Value iteration involves assigning estimates and iteratively improving them until they converge to a desired tolerance [How60, p.30].

Neller assigned estimates to the probabilities in equations 2.32 and 2.33 and repeatedly iterated until the probability values settled within a desired tolerance. Due to the way in which different game states are related, it makes sense to break the problem down into levels and solve one by one. A level is defined by the sum of the two players' current scores $(i+j)$. In the standard game of Pig with a target of 100, this means the levels range between 0 and 198. The probability of winning at any game state will only depend upon probabilities for game states in its own level or higher. For this reason, Neller solves for all game states in level 198 before proceeding backwards until all levels are complete. This system speeds up the convergence, as previously calculated values will be used to compute later levels, resulting in an efficient method for creating the optimal strategy.

Given that this is the best existing attempt in the literature to describe optimal play, our second fundamental aim was to recreate this strategy for ourselves. The primary reason for doing so was to have a single ‘best strategy’ to compare all others strategies we develop to by their average win rate. We go on to show how techniques from dynamic programming can be used to provide an alternative means to value iteration in calculating this optimal strategy. In addition to this, an adaptation of the dynamic programming approach enables us to compare any two strategies on the basis of a probabilistic win rate. We explore how comparing strategies to the optimal in this manner might be used as a natural choice of fitness function for our genetic algorithms.

3 Methodology

3.1 Overview

We created our own version of Pig in MATLAB; enabling us to play any two given strategies against each other and record their number of wins over a specified number of games. We then implemented two tournament based genetic algorithms and tested them on a selection of initial populations of strategies. Using the approach of dynamic programming, we recreated the optimal strategy proposed by Neller and explored how it can be used as a fitness function for our genetic algorithms.

3.2 Programming with MATLAB

MATLAB was an appropriate choice of software as it specialises in matrix operations which are prominent within our project. This computer programming platform was also one which we were all reasonably familiar with already. It's user-friendly nature allowed us to focus on the design of our programmes and algorithms without having to learn a significant amount of new syntax.

3.3 Creating Pig

As the basis of our project, it was necessary to create a version of Pig in MATLAB that enables any two strategies to play a single game against each other. Once we had this we made an additional function to repeatedly play the game a given number of times and output the total number of wins for each strategy. Our programme was easily adjustable in terms of the target score, number of sides of the die and the number of games to be played.

All input strategies are represented by a three-dimensional binary array, in which each element represents one of the $2^{1,000,000}$ possible game states in Pig. Three dimensions, all of length 101 [2], correspond to the player's score, the opponent's score and the current turn score. A '1' at a given position indicates that the player's strategy calls for a roll for that particular game state, whereas a '0' corresponds to a hold. We are able to represent any strategy for the game in this manner.

In order to prevent the possibility of a game reaching stalemate, with neither player deciding to roll, it is necessary to impose the restriction that each strategy must roll at least once at the start of a new turn (i.e. when the turn score is equal to zero).

[2]Note: We were forced to use dimensions of length 101 as opposed to 100 due to the inability to access element (0,0,0) given MATLAB's indexing system.

3.4 Genetic Algorithms

Before we had successfully recreated Neller's optimal strategy there was no obvious choice for a fitness function we might use. We decided, therefore, to initially implement tournament based strategies as our research indicated these could act as a viable alternative in our case.

Numerous tournament structures could have been used for our genetic algorithms. However, we settled on two different types of tournament to be used: a round robin tournament and a knockout tournament. Tournament selection was implemented in favour of alternative methods of selection as research suggested it would be simple to code yet efficient and would allow the degree to which better strategies are favoured to be adjusted. The latter is referred to as the *'selection pressure'* and its adjustment is important because it is correlated with the convergence rate of the genetic algorithm [MG95, p.195].

3.4.1 Round Robin Tournament

In our round robin tournament system, each strategy plays every other strategy a specified number of times. The strategies are then ranked in order of total cumulative wins and only the best half of these survive and breed to form the next generation. The benefit of a round robin system is that since it records cumulative wins there is a significantly lower probability that some of our best strategies will be eliminated due to the random variation in win rate. One disadvantage of this system, however, is that since all pair of strategies in the population must play, it can require a significant amount of time per iteration when using even a reasonably low number of games played per pair. The population size we use is particularly restricted by this tournament structure as the number of total pairings increases by an order of N^2.

3.4.2 Knockout Tournament

The knockout tournament system randomly pairs strategies together and plays them against each other a given number of times. The strategy with the most wins from each pair would then survive to reproduce whilst the losers are eliminated. This method has the advantage of being more time efficient than the round robin method in the sense that it would take significantly less time to perform a given number of generations. However, it does come with the disadvantage that there is a higher probability that a 'good' strategy would be eliminated due to random variation in the win rate. Meaning a worse strategy may progress and this could have knock-on implications to the rest of our generations.

3.4.3 Crossover and Mutation

We bred and mutated the successful strategies from each type of tournament using the same crossover and mutation operators. We randomly paired our surviving strategies at each generation and bred them using a simple crossover operator. Our crossover operator worked as followed. Using our two parent strategies, at any game state where they agreed within the matrix, their offspring would also retain this value (either a '1' or a '0'). In positions where the parent strategies did not agree, the position was randomly assigned either a '1' or a '0'. The mutation operator then randomly selects a specified proportion of the game states and 'flips' the value which currently existed in that position i.e. changing a '1' to a '0' and vice versa.

Our mutation operator was originally implemented using nested for loops. A faster method using matrix arithmetic was adopted to make use of MATLABs efficient handling of matrices. The percentage of game states which are mutated is easily altered and we will further explore how the chosen percentage affects our results later in the report. There are many variations of the crossover and mutation operators that we could have used, but we decided to perform them in the manner we did because it was the natural choice. When the two strategies being bred agree in their strategy for a given game state there seemed no reason not to maintain this in their offspring. Equally, in places where they disagree there is no indication as to whether a roll of hold would be more beneficial and so assigning equal probability to each possibility seemed appropriate.

3.4.4 The Cube System

Initial testing of our two genetic algorithms suggested that they were struggling to significantly improve upon the initial population of strategies in the long term. We suspected that a major cause of this might simply be that the decision whether to roll or hold at each individual game state had such a minuscule impact on the overall success of a strategy. This for two reasons. Firstly, given that there are one million different game states, the decision taken at each literally accounts for such a small percentage of the overall strategy. In addition to this, in a single game of Pig the vast majority of game states are never reached and so have no impact whatsoever on the result.

This second point is particularly pressing. After all, the logic behind breeding two successful individuals is that the parts of each strategy which caused them to succeed will be selected and passed onto the next generation (or are at least more likely to). When we can say with certainty that the vast majority of a strategy had no impact whatsoever on its success, we can quickly see how the breeding process might fail to induce significant improvements in the long-term quality of strategies. This results in an algorithm more similar to a random search in all practicality.

In an attempt to rectify this, we developed a novel addition to our existing genetic algorithms which we term the ‘cube system’. This system works by breaking down the three dimensional strategy matrix into a number of cuboidal regions. We then imposed the restriction that at any point in time, all game states existing within a particular region must share the same strategy i.e. they are all roll or all hold. This restriction is pictured in Figure 2.

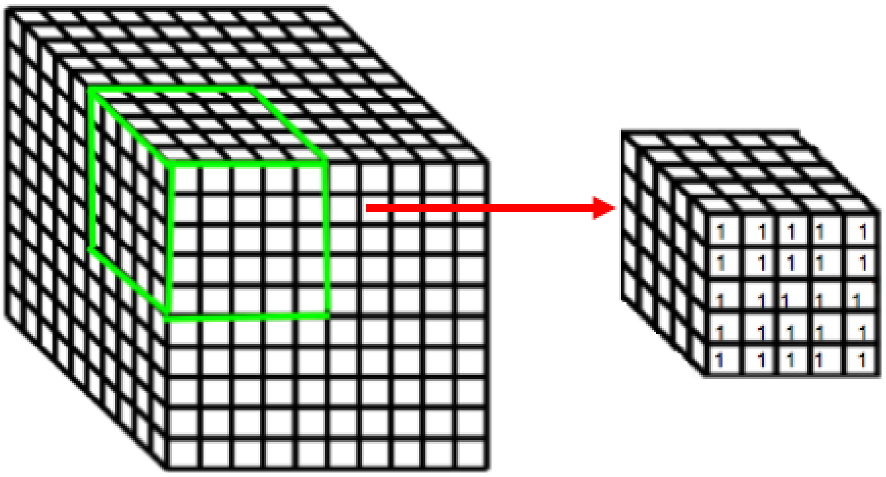


Figure 2: Diagram illustrating the cube system.

The idea behind the system is that as a single decision now governs a larger region of the strategy matrix, it will have a more significant impact on its success. With a small number of large regions, it also becomes increasingly likely that at least one game state within the region will be reached within a given game. In theory, this should allow our existing genetic algorithms to arrive more efficiently at good strategies.

Of course, there exists a trade-off here in relation to the size of region used. Using a small number of large regions means that each region has a significant impact on the success of the strategy whilst simultaneously reducing the size of the search space; both of which we view as positive effects. The drawback to using large regions, however, is that it could potentially restrict the quality of strategies which are able to develop in the long run. This trade-off is analogous to selecting image quality when streaming video online. Using a lower bandwidth will allow you to arrive at your solution more efficiently (i.e. being able to watch without frequent buffering) but will inevitably restrict the final image quality.

It was not immediately clear what size region would lead to the best results [3] and so we programmed this as an adjustable parameter in order to test a variety of cases. We also implemented a version which would perform three successive waves of iterations with increasingly smaller regions. The logic behind this was that by using the larger regions to begin with we could quickly arrive at 'good' strategies and then decrease the cube size to allow for more precise (and ultimately better) strategies to develop.

Note that this 'cube system' merely provides the means to simplify the task for our existing genetic algorithms, of which it can be applied to either. The cube system is not a genetic algorithm in itself.

3.4.5 Experiments

After preliminary testing, three main experiments were performed for each of our two tournament based genetic algorithms. Each experiment began using a different set of starting strategies, each designed to test our algorithms in a different way. We decided to keep the population size, number of generations and number of games per pairing consistent across all experiments to provide the most reliable comparison across the experiments. At any point where two strategies played a match, 100 games were played in the knockout tournament and cube system and 25 games were played in the round robin tournament (due to limitations on time constraints [4]); whereby the winner of the match was determined by the majority of games won. A thousand iterations were performed for every experiment. The three sets of starting strategies we used are outlined below.

Random Strategies

We were keen to test whether our genetic algorithms could take a random population of strategies and evolve them into coherent and successful strategies. Starting with total randomness is perhaps the ultimate test for any genetic algorithm of this kind. Success here would mimic how simple organisms were able to evolve in to the complex life forms we see on our planet. A collection of 20 random strategies were used here. These were used consistently within this investigation to test the different genetic algorithms in a fair way.

[3]Though we are restricted to those which divide 100.

[4]The round robin algorithm takes much longer to run than both the knockout and cube system algorithms. Therefore, in order to overcome this issue, we assigned the round robin algorithm to play 25 games rather than 100. This should not affect the reliability because the strategies are ranked by cumulative wins.

'Hold at' Strategies

As well as starting with randomness, we wanted to see if our genetic algorithms could successfully build upon already good 'hold at' strategies and improve them. In order to do this four copies of strategies were created which hold when the following turn scores are reached: 10, 15, 20, 25 and 30, giving us a total of 20 starting strategies.

'Score' Strategies

In addition to the 'hold at' strategies we decided to test an equally simple set of strategies we developed as an extension to the basic 'hold at' version. We will refer to these strategies as the 'score' strategies.

If the sum of the player's turn score and their total score exceeds their opponent's score then a 'score' strategy will roll up until a certain value, as with the 'hold at' strategies. However, if a 'score' strategy is behind it will roll until the sum of the player's turn score and total score exceeds their opponent's score. Even though you are more likely to roll a 1 when there is a large gap between the player's score and the opponent's score, this maximises your probability of winning as your opponent would otherwise be likely to win within their next few turns.

We will test a mixture of 'score' strategies using both our round robin and knockout genetic algorithms in order to see if they produce better results than the 'hold at' strategies. We would expect to see a slight improvement in the strategies produced as the 'score' strategies will be better than the 'hold at' strategies in game states where the probability of winning has not been maximised.

3.5 Strategy Iteration for Pig

While Neller's method of value iteration is effective in solving the equations he presented, it requires the user to assign a suitable value for the tolerance. Since Neller did not report the value he used for his tolerance we would therefore have to decide on a suitable value ourselves in order to replicate his method. Considering that we do not have the means to compare our resulting optimal strategy precisely to Neller's, we would have no way of knowing whether the value we used leads to a sufficiently accurate solution. For this reason we decided to use an alternative dynamic programming method which we will term 'strategy iteration'.

Dynamic programming is a multi-stage decision process [Bel72, p.7] that converts potentially very complex problems into a set of simpler problems, therefore making them easier to solve. It is necessary to note that dynamic programming itself covers a very broad system of optimisation approaches to solving problems and that our method for creating a solution to Pig is not necessarily the only possible way. [SL77, p.320]

With the strategy iteration method you can assign an initial guess strategy (either a roll or a hold) to each game state. This causes Neller's equations to become a system of linear equations and therefore easily solvable by matrix algebra. It then becomes easier to solve for the optimal strategy, especially in MATLAB due to its specialism in matrix operations.

If we denote each level in Pig by the value of $i + j$,where i and j are the player's score and the opponent's score respectively, it is then feasible to evaluate the probabilities of winning at each game state by starting with the highest possible level and working backwards. This is due to the fact that the probabilities of a win from a game state in a given level are independent of all probabilities of game states in lower levels.
As an example, the highest level in Pig would be when $i + j = 198$ (here $i = 99$,$j = 99$). If we input this into Neller's equations for the probabilities of winning for a roll or hold (Eq 2.32 and Eq 2.33) we get:

$$P_{98,98,0,roll} = \tfrac{1}{6}[1 - P_{98,98,0}] + \tfrac{5}{6}$$

[Eq 3.51]

$$P_{98,98,0,hold} = 1 - P_{98,98,0}$$

[Eq 3.52]

It is clear that all of the probabilities in these equations are of level 198 and therefore independent of probabilities for game states at all lower levels. Now, unlike with value iteration, strategy iteration consists of applying the decision of a roll or hold to each of these probabilities. In order to avoid the risk of the algorithm not converging, we make the decision to start with all of these strategies as a roll: [5]

$$P_{98,98,0,roll} = \tfrac{1}{6}[1 - P_{98,98,0,roll}] + \tfrac{5}{6}$$

[Eq 3.53]

[5]There is a risk that if you start with a strategy that cannot possibly win e.g. a strategy that always holds, the algorithm will continue to run and may never converge. Therefore, in order to address this problem, we start by applying a roll to every single game state.

$$P_{98,98,0,hold} = 1 - P_{98,98,0,roll}$$

[Eq 3.54]

These equations can then be represented in matrix form where $AX = B$ as below:

$$\begin{bmatrix} \frac{7}{6} & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \mathrm{P}_{98,98,0,roll} \\ \mathrm{P}_{98,98,0,hold} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

[Eq 3.55]

Matrix 'A' represents the coefficients of each of the probabilities in Eq 3.53 and Eq 3.54, 'B' holds the constants in each of these equations and 'X' denotes the probabilities of roll and hold for all possible game states for a particular level. We can then solve this matrix equation for the probabilities in 'X' and reassign the strategy at game states whereby the probability matrix informs us that our current guess is wrong. We can then repeatedly iterate in this manner until the strategy does not change. This is then repeated for each successive level in the game, resulting in a matrix of '1's (rolls) and '0's (holds) that represents the decision that should be made at every single game state in Pig in order to maximise your probability of winning, thereby representing Neller's optimal strategy.

For the initial testing of our strategy iteration method we found the optimal strategy for Piglet with a turn score of two (as Neller did with value iteration, see [NP04, p.28]). With a version of the game this simple we were able to check our results by hand to ensure our method was working as we expected. We then adjusted our code for it to work for the full version of Pig. There are some limitations to how accurately we can test whether we have recreated Neller's optimal strategy exactly, since we do not have access to the data he used to create it.

Once we had convinced ourselves that our strategy iteration was working as we expected, we decided to adapt the method to evaluate the probabilities of any two given strategies winning from every given game state. In this way, the probability that each will win from the starting game state (and therefore their probability of winning in general), provides a means to assess their win rate on a probabilistic basis. The main reason we chose to make this adjustment was so that we could compare two strategies without having to worry about random variation affecting win rates. Comparing a strategy to our optimal solution using this method presents itself as a means for a fitness function which we explore later on.

3.6 Team Approach

Within our group of eight, different tasks were delegated to two separate subgroups. A team of four focused on carrying out background research; specifically working towards analysing Neller's report and his equations and primary research on genetic algorithms. The other sub team designed and manufactured a running simulation for Pig, in MATLAB, that could be altered to become Piglet or other variations of the game.

We further assigned two people from each subgroup to swap and form the next pair of subgroups; ensuring that we had a mixture of members familiar with both computing and the relevant background research. When working in these teams, utilising communication and explanation skills was essential.

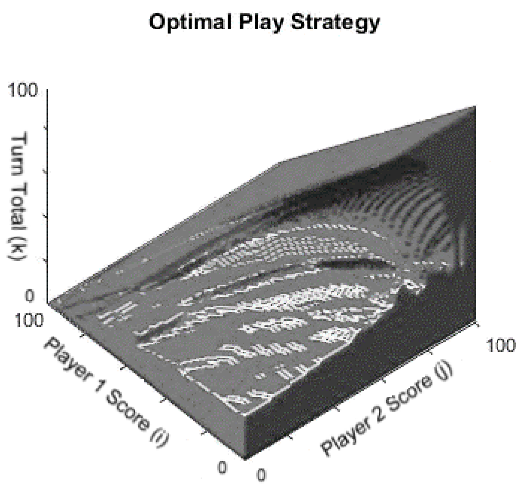
Writing up the report was an ongoing task throughout all stages of the project and was completed by all members of the group. Members wrote about the aspects of the project that they understood and had specialised in during the preliminary stages in order to play to their strengths. Time management and organisation was fundamental in the written communication of our findings due to the time constraints on this project.

Competence in computer software such as MATLAB, R and LaTeX is now prevalent within our group and will aid all members in the future.

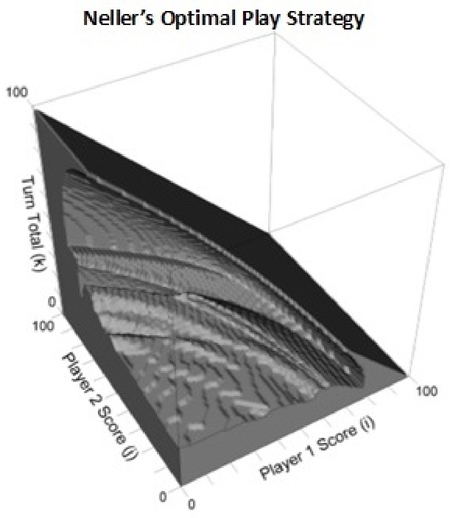
As well as utilising the above attributes, we further developed our problem-solving skills whilst analysing our findings and explaining certain data discrepancies.

4 Findings

4.1 Optimal Strategy from Strategy Iteration

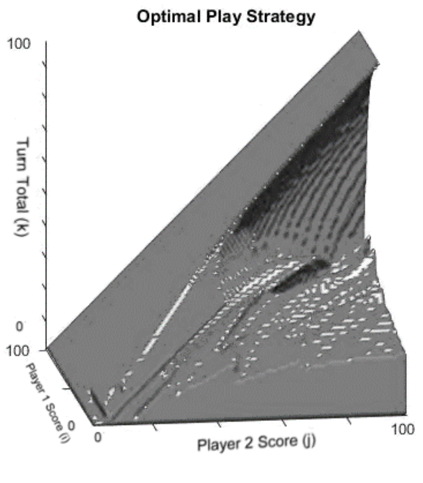


(a) Optimal Strategy from Strategy Iteration

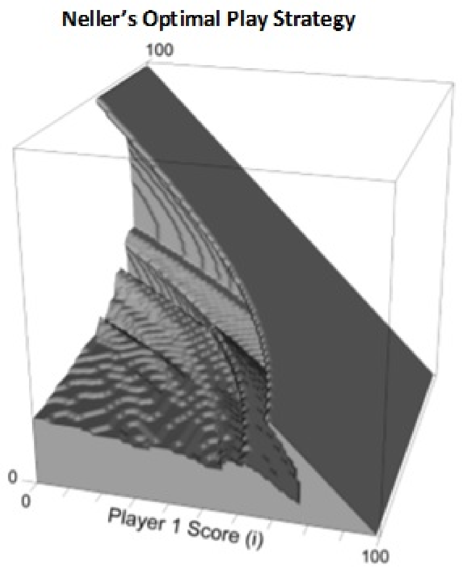


(b) Neller's Optimal Strategy

Figure 3: Comparing our Optimal Strategy to that of Neller's.



(a) Optimal Strategy from Strategy Iteration



(b) Neller's Optimal Strategy

Figure 4: Comparing our Optimal Strategy to that of Neller's.

Here we present the results from our strategy iteration method for finding the optimal strategy. This is graphically represented in Figures 3 and 4 which show both ours and Neller's optimal strategies from two different angles.

The graphs demonstrate the roll/hold policies for every game state in Pig with a target score of 100 and a standard six sided die. The game terminates after the set target is reached; explaining the triangular shaped graphs whereby the white space represents the unreachable states after the player's banked score and turn score combined exceeds the target score, 100. The axes are i (Player 1's score), j (Player 2's score), and k (the turn total). The visualisation of our results (Figure 3a and 4a) are a mirror images of Neller's optimal strategy. This is due to the axes being the opposite to Neller's for our findings. With this in mind, it is still clear that our version of the optimal strategy resembles Neller's findings; though we cannot be 100% sure how exact we are due to the fact that we do not have an exact copy of Neller's strategy used to create his graph.

The surface of the graphs show where a player should roll. The shaded area represents regions at which it is optimal for the player to roll whereas the white space above the surface represents when it is optimal to hold.Overall, we see that the ‘hold at’ 20 or 25 policies only serve as a good approximation to optimal play when both players have low scores. When either player has a high score, the ‘hold at’ methods do not maximise the probability of winning and so it is advisable on each turn to try to win. When the optimal player has a significant advantage over the opponent, the optimal player maximises their expected probability of a win by holding at turn totals significantly lower than average. This can be highlighted in Figures 3 and 4. It is important to note, however, that this optimal strategy is not a practical method for human play in Pig, unlike the ‘hold at’ strategies.

4.2 Testing the Mutation Rate

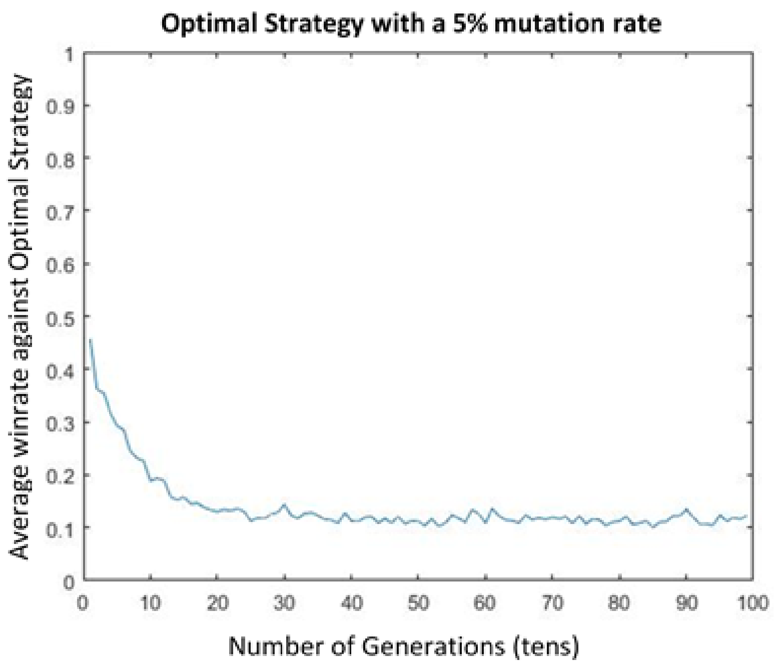


Figure 5: Testing a 5% mutation rate for the optimal strategy

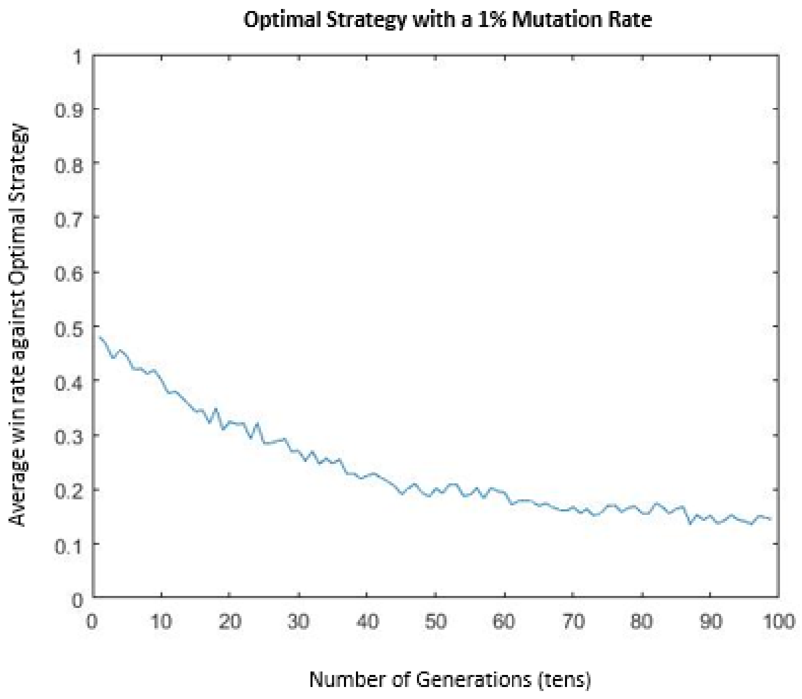


Figure 6: Testing a 1% mutation rate for the optimal strategy

Figures 5 and 6 were produced when running twenty identical versions of our recreation of Neller's optimal strategy through the knockout genetic algorithm. The only difference between the two tests we ran was the percentage of values that were mutated in each individual offspring generated. These different mutation rates of 5% and 1% are displayed in Figures 5 and 6 respectively.

The aim of this particular test was to observe the effect the mutation operator was having on inputted strategies. We, therefore, ran the algorithms for 200 iterations and took results every two generations as opposed to the usual 1000 iterations and 100 interval results we perform for other tests. Ultimately, we were interested in the rate at which the win rate decreased as each new generation was played against our recreation of Neller's optimal solution. As a result, we deemed 200 iterations to be sufficient as any significant decrease in win rate should occur during the primary generations and therefore running further iterations would not tell us anything more and only result in the tests taking longer to complete. Any change in a strategy considered optimal, such as Neller's, should in theory only result in the said strategy gradually getting worse. As we can see from both Figures 5 and 6 this is, in fact, the case. In Figure 5, we notice that even after only the second iteration, the new generation of strategies dropped to a win rate just above 0.45. After twenty iterations, the win rate had decreased dramatically to just below 0.20 and after forty the new generation of strategies plateaued; winning on average around 0.11 of all games played against Neller's optimal strategy.

Comparatively, we can see from Figure 6 that the decrease in win rate is much more gradual than that of Figure 5. The results after two iterations had dropped to a win rate just below 0.5 which is better than that for Figure 5. The test running a 1% mutation, Figure 6, took almost until the 160^{th} iteration to plateau like the results in Figure 5. Moreover, Figure 6 also shows that the results for a 1% mutation plateaued at a win rate of just below 0.15 which is higher than the plateaued win rate of a 5% mutation of 0.11.

Based on the results of these basic tests, we opted to run our genetic algorithms with a mutation rate of 1% only. We felt any higher mutation rate would have too large of an effect on any offspring produced and may result in us moving away from the beginnings of potentially successful strategies.

4.3 Testing Random Strategies

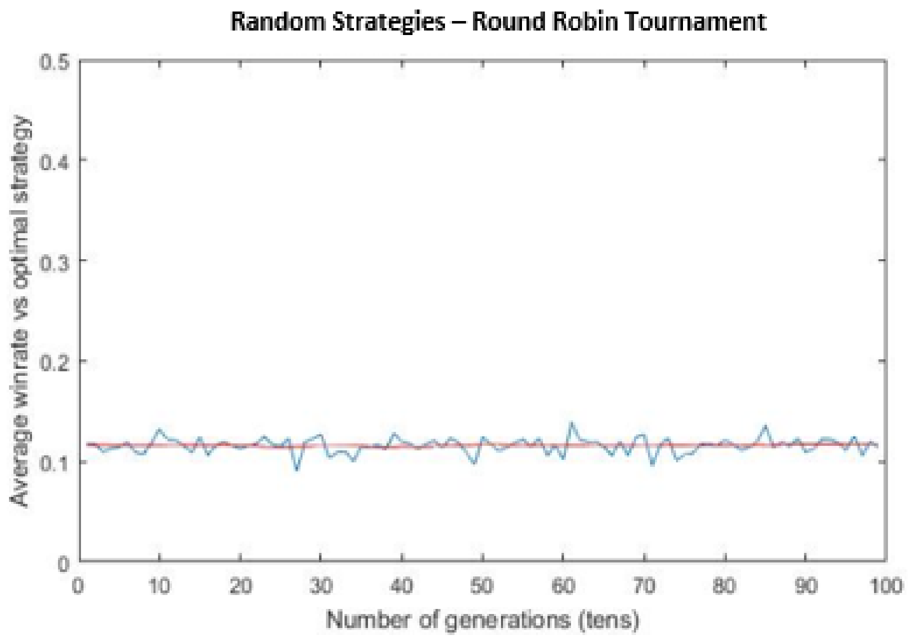


Figure 7: Results of the round robin genetic algorithm for 20 random strategies.

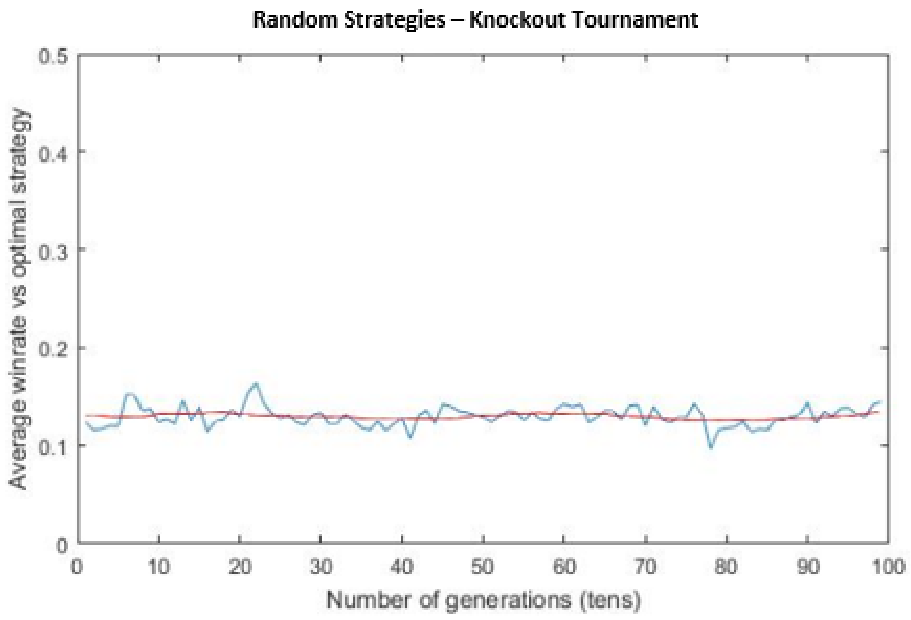


Figure 8: Results of the knock out genetic algorithm for 20 random strategies.

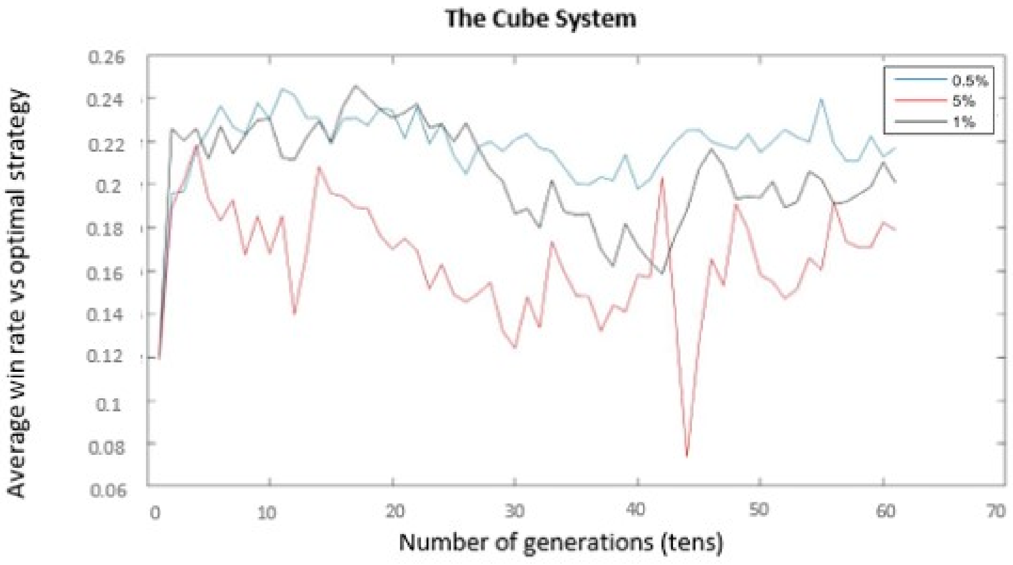


Figure 9: Results of the cube system using 20 random strategies and three different mutation rates represented in different colours (see key).

Figures 7 and 8 display the round robin and knockout results respectively for tests starting with the same 20 random strategies and ran for 1000 iterations. The only difference between the two tests was that the round robin algorithm played 25 games, whereas the knockout algorithm played 100; this is due to the fact that the round robin algorithm is more time-consuming to run. Each data point on the respective graph shows the average win rate for the all strategies in a generating when played against the optimal strategy. Data points only recorded every ten generations in order to save time. .

We can see that Figures 7 and 8 both follow a similar trend with the win rate only fluctuating by a small amount in either direciton. This is to be expected due to the fact that we are starting with completely random strategies that have no structure. Since we are playing these strategies against our optimal strategy, it is very difficult for the future mutated strategies to improve significantly. As a result, the moving average (visualised using the red line on both Figures 7 and 8) shows no real change over the course of 1000 generations.

However, the cube system, shown in Figure 9, seems to illustrate a slightly better performance than both the round robin and knockout algorithms. Here, we use the knockout tournament as the genetic algorithm, playing 100 games with 100 iterations per change in the size of cubes. The three coloured lines represent the different mutation rates of 0.5%, 1% and 5%. We found that changing the mutation rate for the round robin and knockout methods did not make any significant difference to the results and so we only included the 1% rate of mutation for these algorithms.

However, for the cube system, we see much more varied and interesting results when the mutation rate is altered. All three mutation rates display a high initial peak in win rate for the first 10 generations; this is most likely due to the fact that with large cubes, one change in a significant place can have a large impact. With a lower mutation rate, we can see that the results fluctuate much less, giving a more constant and overall better win rate. The reason for this is that if the mutation rate is too large, you run the risk of varying the strategies too much with each mutation and therefore having a greater potential of losing successful strategies.

4.3.1 Comparing Our Results to a Random Search

As discussed in our outline of the 'cube system', the difficulty we have with our tournament based genetic algorithms is that they assume the decision to roll or hold at every game state has an equal influence on the success of the strategy. However, given that such a tiny proportion of game states are reached in a single game of Pig, this is far from the case.

Of course, by playing two strategies against each other repeatedly we are able to increase the proportion of the game states which contribute to the result. In reality, however, the number of games that would be required for a reasonable proportion of game states to be reached is exceedingly large. Whilst we are unable to calculate this number, we might estimate that it would take 1000 games where each strategy consults an average of 50 game states in a single game in order for just 5% (50,000) of game states to be reached. This is under the assumption that the same game state would not be consulted twice. This is clearly not the case as the initial conditions (0, 0, 0) will be consulted at least once a game. With our current implementation of the algorithms and the computing power available to us, this number of games is simply far too large to achieve a reasonable run time.

The worry is, therefore, that our genetic algorithms are in essence more of a random search. In order to test this theory, we decided to compare the results from our round robin and knockout genetic algorithms for the random starting strategies to those achieved using a true random search.

The random search algorithm we used for comparison was brutally simplistic. We created 20 random strategies and recorded their average win rate against optimal solution. We repeated this process 300 times, each time for a new random set of 20 strategies.

In order to compare the results of our genetic algorithms to this random search, we wanted to find the probability that this random algorithm would, at any point, encounter a set of strategies that are at least as good as the best strategies found using our genetic algorithms. If we found there to be a reasonable likelihood that a random search would find better results than our genetic algorithm, then we could conclude that we failed to improve our strategies beyond the level which could be achieved using a totally undirected search.

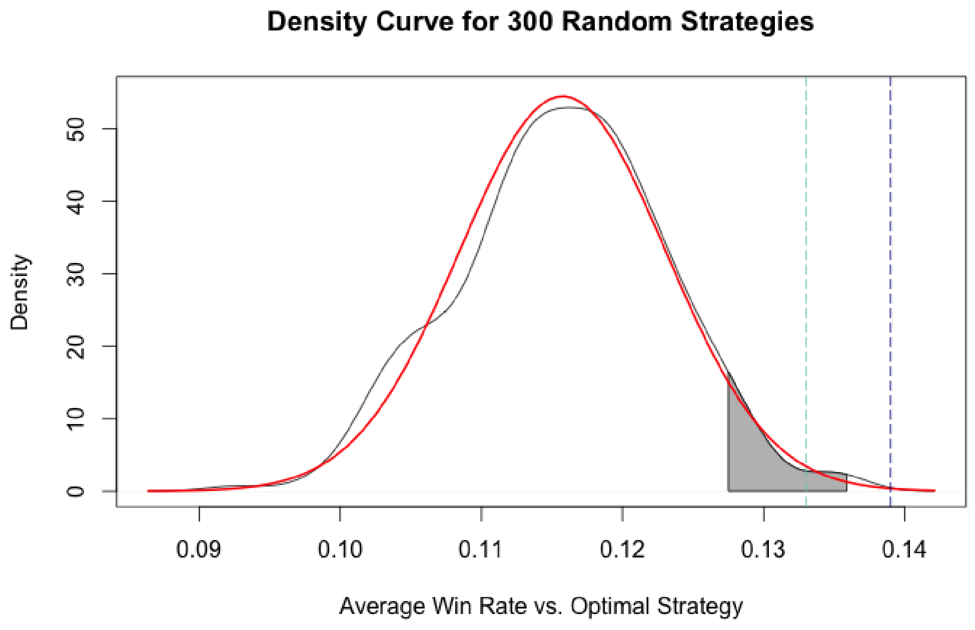
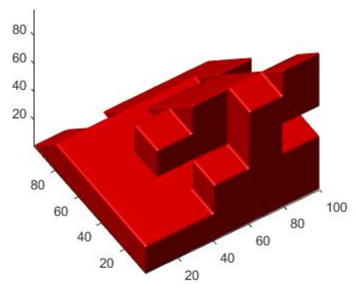


Figure 10: Graph representing the density curves of the average win rate of 300 random strategies against the optimal strategy (black line) and the normal distribution corresponding to this data (i.e with the same mean and standard deviation - red line). The shaded region is the upper 5% quantile and the green and blue dotted lines represent the best win rates associated to the knockout and round robin genetic algorithms respectively.

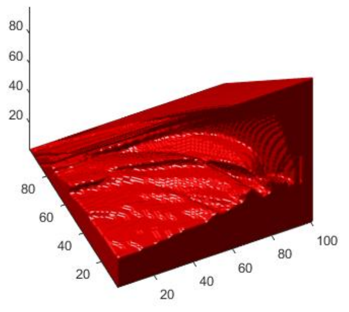
Once we had our results from the random search algorithm, we calculated the mean and standard deviation of the data. Figure 10 shows a density plot of the data (black line) alongside the corresponding normal distribution (red line) using these parameters. It is a fair to assume that our random search data is normally distributed s the black line closely follows the red line in Figure 10. Furthermore, with 300 data points our parameter values should be sufficiently reliable for the purposes of this comparison. We then calculated the 95%, 99% and 99.9% quantiles of the normal distribution and compared these values to the peak performance from each genetic algorithm. For the round robin tournament algorithm, we recorded a peak performance of 13.9% win rate against optimal; placing it in the top 0.1% of solutions we could expect to find with our random search. In the case of the knockout tournament we found a maximum win rate of 13.3% which places it in the top 1% of strategies we could expect from the random search. However, due to the large number of generations that were created and played against the optimal, we cannot conclude that these are better than we would expect from a random search.

The results of the cube system, however did demonstrate an improvement. In particular with a 0.5% mutation rates the cube system exceeded a win rate of 0.24. From the statistics given above this is clearly much better than a random search. When using a lower mutation rate the graphs seem to fluctuate less. However, we cannot say this is certainly the case as we only ran the genetic algorithm with the system once. We also see a huge initial improvement in the population. This is due to the fact that there are some crucial 20x20x20 areas within a strategy. For example, an optimal player will almost always roll up to 20. Using the cube method we will reach a strategy close to this at much faster than the other systems. Once these areas have their appropriate roll or hold value, they are very unlikely to be bred out as they are so advantageous. Using the cube method as we progress through the generations the cubes become smaller. At generation 21, the cubes go from 20x20x20 to 10x10x10 and then at wave 41 go to 5x5x5. Interestingly whatever the mutation percentage the strategies became notably worse when moving from 20x20x20 cubes to 10x10x10 cubes. This could be due to the fact that if the mutation causes an important 10x10x10 block to be removed then the strategy is more likely to survive as this will not have as drastic an impact as an important 20x20x20 block being removed. The strategies appear to improve during the 5x5x5 wave, however the experiment was not run enough times to be able to conclude this with any certainty.

One of our aims was to create a strategy resembling optimal. Below we provide a visual comparison between one of the final strategies produced by the cube system (Figure 11a) to the optimal strategy (Figure 11b). Whilst the finer details of the optimal strategy cannot possibly be achieved using cubes of this size there is a clear resemblance in terms of their general structure.



(a) Final strategy from the 'cube system'



(b) Our Optimal Strategy

Figure 11: Visual comparison between a strategy from our 'cube system' and the Optimal Strategy

4.4 Testing ‘Hold at’ Strategies

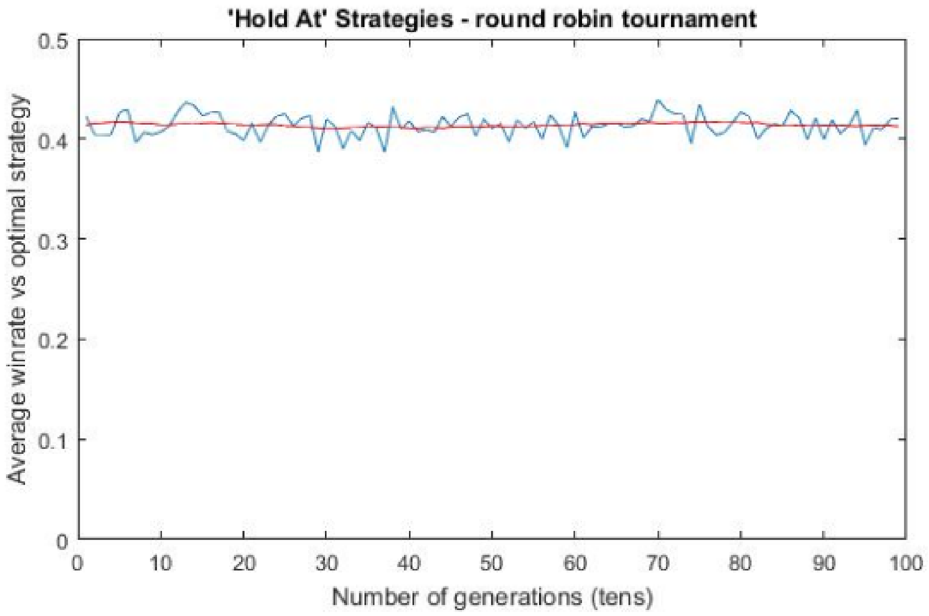


Figure 12: Results of the round robin genetic algorithm for 20 mixed ‘hold at’ strategies

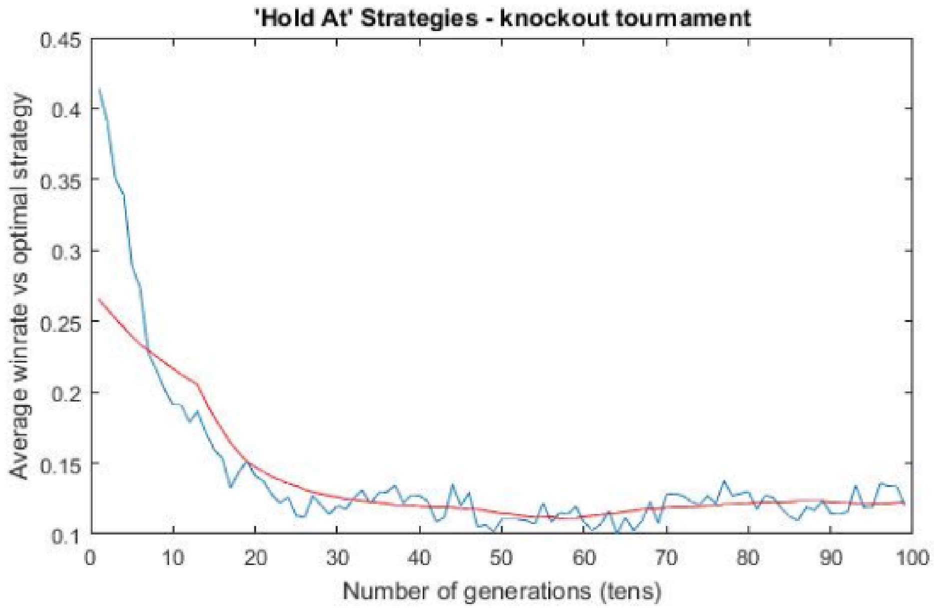


Figure 13: Results of the knockout genetic algorithm for 20 mixed ‘hold at’ strategies

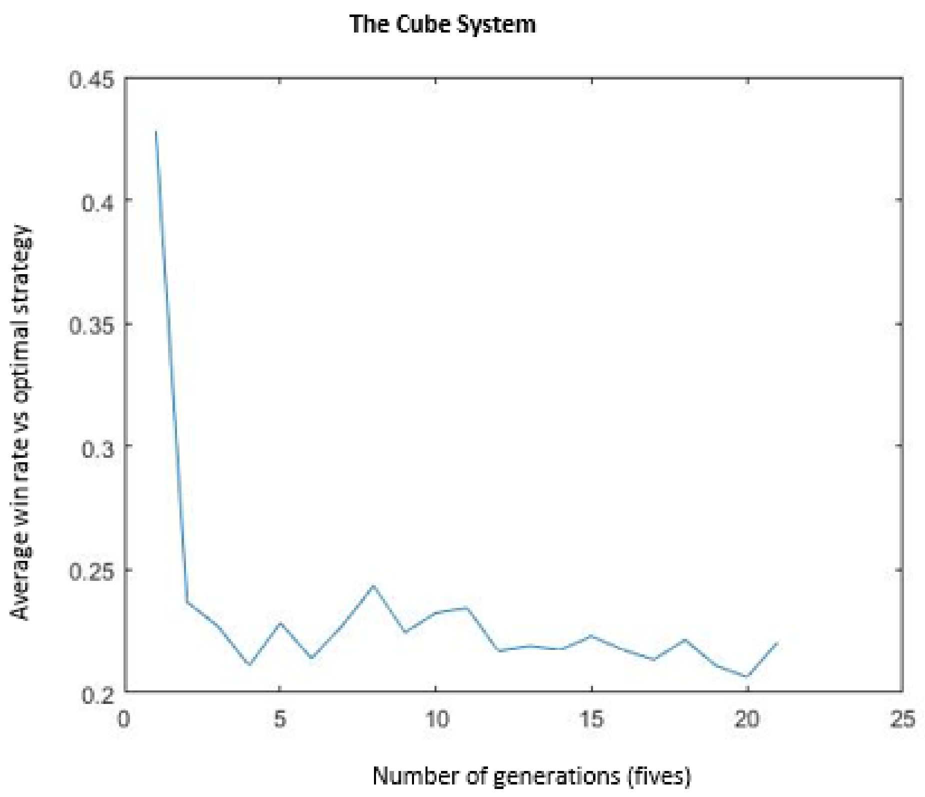


Figure 14: Results of the cube system for 20 mixed ‘hold at’ strategies

Figures 12 and 13 display the round robin and knockout algorithms respectively, starting with a combination of hold at strategies for the following values: 10, 15, 20, 25 and 30. The starting population included four identical versions of each of these ‘hold at’ strategies, running through 1000 iterations, with the round robin algorithm playing 25 games and the knockout method playing 100. As with the random strategies, each data point represents the result for when the strategies played the optimal strategy every 10^{th} generation with a mutation rate of 1%.

Unlike with our results from the random strategies, where both the round robin and knockout tournaments illustrated a similar trend to each other, it is clear from Figures 12 and 13 that the two different methods of genetic algorithms give contrasting results for our combination of ‘hold at’ strategies. Figure 13, referring to the knockout method, displays a sudden downfall in the win rate which never increases again. This is likely due to the fact that there is a high possibility that the known good strategies (hold at 20 and hold at 25) will be eliminated from a generation after 100 games due to chance. If this is the case, the following generations will struggle to improve as there is a chance that there will no longer be good competitive strategies in the mutating population and, as a result, the graph quickly becomes similar to results that we obtained with the random strategies. In comparison, with the round robin tournament (Figure 12), it is highly likely that since we record cumulative wins that only the better strategies are reproducing, not due to chance, but due to them actually performing well consistently. Therefore, we have similar, already successful strategies attempting to mutate and reproduce which actually results in there being little room for further improvement. This is demonstrated by the red line on the graph which represents the moving average for these results. We can see that it remains constant throughout the generations but at a fairly high win rate. Although the round robin tournament has not decreased the win rate of the strategies, it is not necessarily an ideal method when starting with already good strategies as the initial population.

As previously mentioned the cube system is not a genetic algorithm in its own right but is rather an addition to the knockout genetic algorithm. For this reason, when testing 'hold at' strategies the win rate will have an initial rapid decrease (similar to that in Figure 13), which is visualised in Figure 14. Therefore, we decided not to persist with fully running the cube system for the 'hold at' strategies. Note that the cube system could be re-written to incorporate the round robin algorithm, which would potentially give us better results, but due to limits of time constraints and the fact that our 'hold at' strategies do not seem to perform well as an initial population, we do not cover this in our project.

4.5 Testing 'Score' Strategies

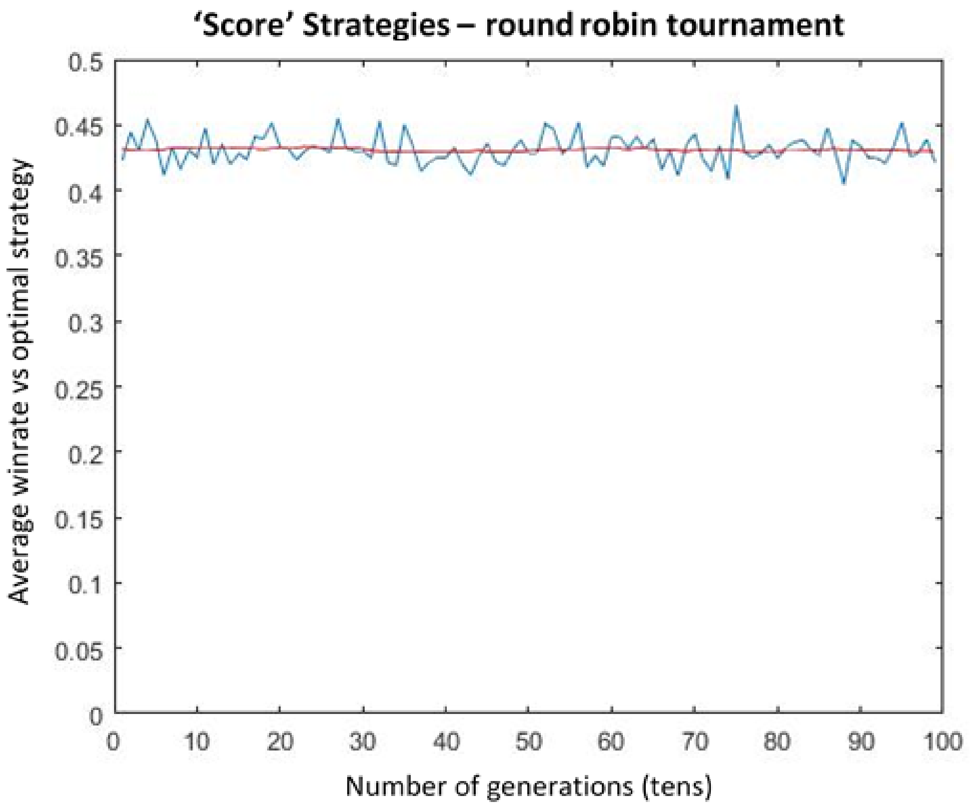


Figure 15: Results of the round robin tournament for 20 mixed 'score' strategies

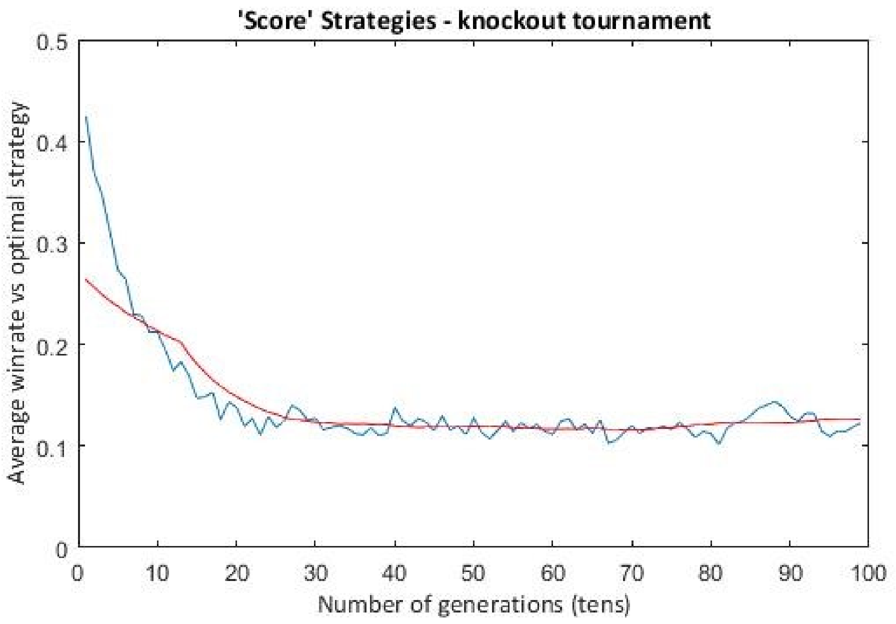


Figure 16: Results of the knockout tournament for 20 mixed 'score' strategies

Figures 15 and 16 display the results when a combination of 'score' strategies (four identical versions of each of 10, 15, 20, 25 and 30) played 25 games using our round robin algorithm and

100 games using our knockout algorithm, with 1000 iterations. As with our other tests, each data point represents the result when the strategies were played against the optimal strategy every 10^{th} generation, with a mutation rate of 1%.

Due to the fact that ‘score’ strategies are an adaptation of the ‘hold at’ strategies, we see very similar patterns to those in Figures 12 and 13. The data remains fairly constant in Figure 15, illustrated using the mean average, as the better strategies will tend to win due to the cumulative scores. This means that only the good strategies will be responsible for reproducing the next generation, leaving a very small margin for improvement. In Figure 16, we see that as before there is a steep initial drop in the win rate of the strategies. As with the ‘hold at’ strategies, this is because it is quite likely that better strategies could be eliminated early on, making it much better for the mutating strategies to improve significantly.

However, as we would expect, the average win rate is slightly higher for the ‘score’ strategies than the ‘hold at’ for both algorithms. This is because the ‘score’ strategies account for situations where holding at a certain score would not necessarily maximise the probability of winning.

5 Discussion

5.1 Conclusion

Our project aims were to explore the use of genetic algorithms in improving a variation of starting strategy populations; namely 'hold at' strategies, 'score' strategies and random strategies. Our findings highlight some interesting insights as to how effective genetic algorithms are in optimising the play of the dice game Pig.

Overall, we found there to be no notable movement towards developing a coherent result when starting with random strategies for either style of tournament; round robin or knockout. However, when starting with 'hold at' or 'score' strategies, we observed that the offspring strategies either maintained the same win rate against the optimal strategy (round robin) or tended to random strategies (knockout).

The cube system allows for the introduction of structure when it is absent, such as with random strategies. This allows the win rate of random strategies to significantly improve in the first few iterations, though following this the improvement of win rate tends to stagnate. Although the win rate stops increasing, the initial tests show that this method can be improved on in the future and has the potential to locate successful strategies. In comparison to the use of the cube system on 'hold at' and 'score' strategies where it results in the alteration of a clear structure, often it is this structure (such as holding at a value) that is the foundation of a successful strategy.

Comparing all three methods (displayed in Figures 7, 8 and 9), we conclude that the cube system seems to demonstrate positive results for developing strategies using genetic algorithms. The fact that we are playing our offspring strategies against the optimal makes it difficult to assess how good the outcomes are. However, the cube method still displays a higher positive change in win rate, reinforcing that it is a positive development of our genetic algorithms.

Another aim we achieved was to recreate Neller's optimal strategy using dynamic programming. We used strategy iteration as our method of dynamic programming, and as this method does not require a tolerance, we have created a more precise optimal strategy from Neller's original equations than if we were to have used value iteration. From our results, we were unsuccessful in finding a strategy that beats Neller's optimal strategy and we can therefore not disagree with the argument that this is the optimal method play - although this is not to say that a better optimal strategy does not exist.

5.2 Evaluation

To evaluate our recreation of Neller's optimal strategy, we compared our findings for the probability of winning, calculated using our fitness function, to results Neller outlines in his papers. He states that his optimal strategy when playing against 'hold at' 20 has a 58.74% probability of winning if the player using the optimal strategy takes the first turn.[NP04, p.34] In comparison, our fitness function gave a probability of 57.19% (as shown in Appendix A) for the same situation of play. At first appearance, the differences in reported win rate might seem a clear indication that the two optimal strategies are not quite the same. We point out however that, as far as we know, Neller calculated these wins rates by repeated sampling. In comparison we use the more accurate method adapted from our strategy iteration which is able to compare strategies on a probabilistic level. The random variation in win rate alone could easily explain why Neller reports slightly different values. In addition to this, the method of value iteration used by Neller involves the used of a tolerance as previously discussed. If the tolerance is set even slightly too high it would leave to an inaccurate solution to the probability equations. Our method of strategy iteration on the other hand provides an exact solution. The differences in the two methods could be another reason for the differences in reported win rates.

Admittedly, trying to ascertain whether ours and Neller's optimal strategies are the same on the basis of reported win rates is not a reliable comparison. Unfortunately however without an exact copy of Neller's solution we have little other way to evaluate our findings. We feel however that the similarities, both visually and in their reported win rates provides a strong indication that we have indeed recreated the optimal strategy as we concluded.

The biggest contribution from this paper to existing literature, is the implementation of genetic algorithms to try to optimise the strategy of the game Pig. Although we may not be able to improve on Neller's optimal solution, it gives us another method of verifying that Neller's solution is in fact optimal.

One of our biggest findings was the development of our cube system and although its process is similar to that of 'messy genetic algorithms' [Mit98, p.121] it is an original concept in the field of genetic algorithms. It is particularly useful in this project as it supports the idea that genetic algorithms could warrant future research and development.

Throughout this project, we met with game theory specialist, Dieter Balkenborg, with the main aim of discussing and verifying the validity of our findings. With his background knowledge on theoretical problems, he indicated that genetic algorithms are very complex and should be tailored to specific cases, and in our study we should have investigated our mutation at a lower rate. When looking at our cube system, he understood the logic behind the method and had high expectations of it succeeding.

We tried to conduct our testing in the fairest manner possible by running a sizable amount of games and iterations for each separate tournament. However, in an ideal set of conditions we would have liked to have ran our testings with both parameters having a substantially large input. If we had a computer with a superior processing speed or a larger time period for testing then this would increase the validity of our findings.

One method that we could have implemented to both reduce time constraints and allow for fairer testing would be to play a set of games of a smaller cumulative amount but where for each individual set, the winning the strategy would gain a point. The overall effect of this is that the tests become fairer due to there being a lower probability of a 'good' strategy losing the majority of a capped amount of sets.

Ideally, we would have liked to use the adapted version of our strategy iteration method as a fitness function. This would allow us to assess the quality of each solution on the basis of its probabilistic win rate against the optimal without error due to random variation. Unfortunately however, at least with the technology and time frame available to us, this method was far to slow to be a viable option. For this reason any selection operator we use will involve repeatedly playing strategies against each other to assess their win rate. We must accept that there will always be a certain variance associated with this method. Therefore, our time-efficient tournament based algorithms also have the potential to remove better strategies from the population. If good strategies are eliminated early in the course of an evolution this can have significant consequences on the quality of later generations and so the reliability of repeatedly playing game is of great importance to our work.

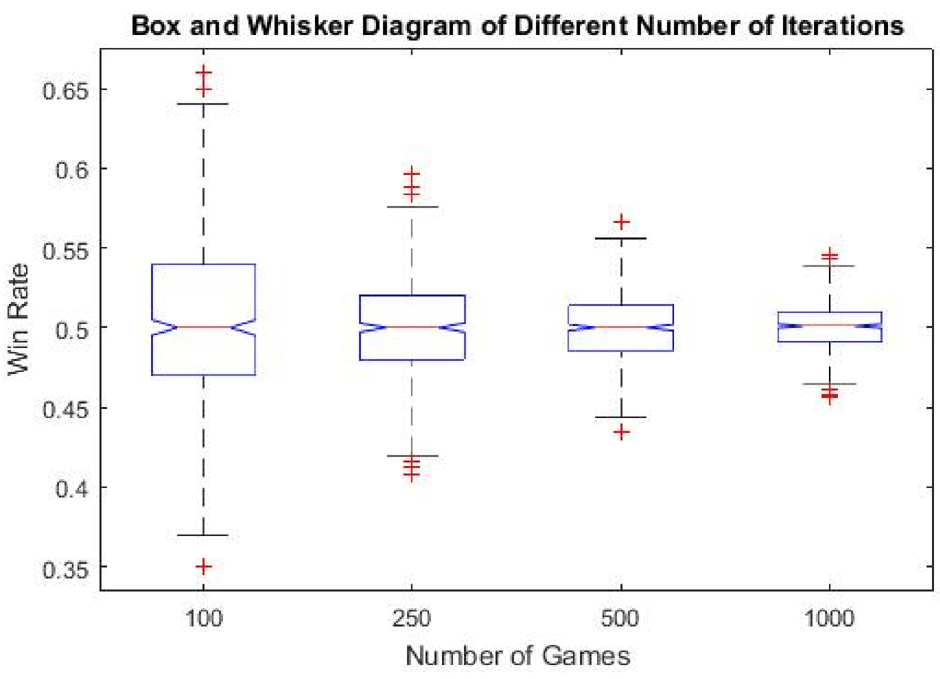


Figure 17: Examining the variation as a result of the number of games played.

In Figure 17, we demonstrate how playing substantially large numbers of games between each pair of strategies can positively affect the reliability of the findings. The above graph illustrates the win rate of two 'hold at' 20 strategies where we played them in a certain numbers of games and recorded the amount of games won. We repeated this process 500 times for each number of games; Figure 17 is a box and whisker plot demonstrating the distribution of our results. Since the strategies are identical and alternating between who takes the first turn, the expected win rate is 50%. We can see that increasing the number of games results in a decrease of the variance. For our knockout tournament and cube system, two strategies were played against each other at random and the winner progressed to the next generation and was allowed to breed. As we can see, a sample of 100 games can provide a wide variety of results, as the maximum and minimum values of win rates recorded were roughly 64% and 37% respectively. This highlights the extreme variation of potential win rates, which could result in a better strategy being eliminated in favour of a weaker one.

In our round robin tournament each game played 475 games (25x19). As we can see from the graph, 500 games gives a much better representation of a strategy. This could explain why during the round robin tournament the 'hold at' strategies did not get any worse, unlike the knockout tournament. However, this could also be down to the fact that during a round robin a strategy is compared to every strategy in the generation, whereas for the knockout tournament, two good strategies could be chosen thereby forcing one of them to be eliminated; a problem that is non-existent in the round robin.

5.3 Further Work

As a natural extension of the work carried out here, we would like to be able to experiment with the use of alternative tournament systems as well as more advanced crossover operators. With a greater amount of time, we would also have liked to carry out more extensive testing into the effects of the mutation rate. Our findings repeatedly demonstrated that, at least in this instance, lower mutation rates lead to a more successful genetic algorithm. With more time we would ideally want to test rates of mutation as low as 0.1% and even 0.01%. We may even want to observe the effect that having no mutation rate has on the performance of our genetic algorithm before gradually introducing it back in.

An interesting development that could be made from our project is research into exploitative strategies. An exploitative strategy is one that is tailor-made to have a higher probability of winning against a certain strategy in comparison to others. For example, although Neller's optimal strategy has an above 50% chance of beating 'hold at' 20 (regardless of whether the player goes first or second - see Appendix A) there may exist a strategy whose win rate is greater than this and which is optimal in the sense that it has a higher win rate against specifically 'hold at' 20.

In order to do this, we would aim to adapt our strategy iteration method so that we could, in theory, input the strategy which we would like to have the biggest advantage against and, as an output, we would find the strategy which has the greatest probability of beating our opponent's strategy (assuming this remains constant).

6 Bibliography

References

[Bel72] Richard Bellman. *Dynamic Programming.* Princeton University Press of Princeton, New Jersey, 1972.

[BT95] T. Blickle and L. Thiele. "A Mathematical Analysis of Tournament Selection". In: *Proceedings of the 6th International Conference on Genetic Algorithms* (1995).

[Col99] David A. Coley. *An introduction to Genetic Algorithms for Scientists and Engineers.* Wspc; Har/Dskt edition, 1999.

[MG95] B. L. Miller and D. E. Goldberg. "Genetic Algorithms, Tournament Selection, and the Effects of Noise". In: *Complex Systems* (1995).

[Mit98] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1998.

[NP04] T. W. Neller and C.G.M. Presser. "Optimal Play of the Dice Game Pig". In: *The UMAP Journal* (2004).

[SL77] Arnoldo C. Hax Stephen P. Bradley and Thomas L.Magnati. *Applied Mathematical Programming.* Addison-Wesley Publishing Company, 1977.

7 Appendix

7.1 Appendix A

Probability of Winning					
	Optimal	Hold at 20	Hold at 25	Score Strategy 20	Score Strategy 25
Optimal	0.5301	0.4914	0.5088	0.4938	0.5046
Hold at 20	0.5719	0.5347	0.5435	0.541	0.5368
Hold at 25	0.5518	0.52	0.5298	0.5243	0.5282
Score Strategy 20	0.5688	0.5272	0.5435	0.5336	0.5341
Score Strategy 25	0.5576	0.5289	0.5322	0.5324	0.5307

Table 1: Table showing the relative probabilities each strategy has of winning against every other strategy (assuming that the strategies labelled in the columns are to have the first turn).