Developing an AI Solution to Backgammon

By Samuel Gartside

BSc Computer Science

Supervised By: Dr David McLean

I acknowledge that this project is my own. No part of this project has been previously submitted as part of any other degree submission, apart from the areas containing citations and references.

Samuel Gartside

Contents:

1.Terms of Reference:

This is the terms of reference for my proposed backgammon artificial intelligence project. This will show my intended aim and objective. It will also contain some background information about the project and some of the potential problems I might encounter.

Learning outcomes:

To use knowledge, abilities and skills for further study and for a range of employment in areas related to scientific and technical computing;

Interpret legislation appropriate to computer professionals and be aware of relevant ethical issues and the role of professional bodies;

Analyse, design, and implement algorithms using a range of appropriate languages and/ or methodologies;

Apply the principles and operation of languages, compilers and interpreters;

Demonstrate effective communication, decision making and creative problem-solving skills, and identify appropriate practices within a professional, legal and ethical framework;

Critically appraise and apply suitable artificial intelligence techniques for a variety of software systems.

Background:

Board games have existed throughout human history. They were often played for entertainment or as a test of wits and cunning. Some of these board games can be traced back to the 36th century BCE to ancient Egypt, the Sumerian civilisation and ancient Persia. Most of these games involve moving pieces around the board often as a means of controlling sections 'or territories' of the board. Because of this, these games often incorporate a level of strategy. One of these games is Backgammon. Backgammon can be traced back approximately five thousand years to Shahr-e Sukhteh in present day Iran, and is a game that requires both skill and luck. As a result, the most skilled player doesn't always win every game, however multiple games are often played before deciding a winner. As such the most skilled player will win the overall set. Also, the game has a number of strategies due to the level of randomness and the number of counters on the board at any one time. This means that a higher degree of complexity is required to develop an automated solving algorithm than a game with no element of luck, such as chess, which has more restricted movements.

To play Backgammon a player needs to get his or her fifteen counters from one side of the board to the other. This is achieved by moving along a series of points on the board. The number of moves a player can make is determined by rolling two dice. Counters can only be moved to an open point, this is any point that isn't already occupied by two or more opposing counters. The player can choose between adding the numbers on the dice together for one counter or moving two counters for the number represented on the two dice. If a player rolls a double then the dice are doubled e.g. If a player rolls two fives, then the effect is like rolling four fives. Again, the player gets the choice of move less counters further, or moving more counters a shorter distance.

Counters can be moved passed occupied points but cannot land on them. A point that contains one opposing counter is called a blot. Moving one of your counters onto your opponent's blot is called a hit. Doing this removes their counter from the board and onto the bar along the centre of the board. On their next move they must first move any counters on the bar. Any counters moved this way must start on the opposing players home board, meaning that they must travel to full length of the board to complete the game. However, a player can block their opponents moves by moving two or more checkers onto a point preventing them from hitting it and occupying that point. If this blocks a player's re-entry from the bar, then that player's move is forfeited.

Once a player has all their counters on their home board they must start bearing off. This is the process of moving counters off the board and completing the game. This is done by rolling the dice. The number you get will allow you to move the counters on the corresponding point off the board. If there are no counters on this point then the player must make some other legal move. The game is completed once one player is successful in bearing off all fifteen counters.

It is possible for a simulation to play perfectly using a table lookup method like that used in chess AI, when playing only partial games. However, it becomes impossible to play a full game using this method, due to the complexity caused by the probability of the dice rolls. The game currently has a working AI solution called TD-Gammon. In my initial research I have found out that this uses a type of temporal-difference learning. TD-Gammon currently works by looking at the possible moves of the next two turns. Then each turn it will update the weighting to reflect the most likely of these moves. The final build of TD-Gammon was capable of playing and defeating backgammon masters. TD-Gammon was designed intended to explore issues with reinforcement learning and not for mastering backgammon, Tesauro(1994). As such I intend to research and test alternative algorithms for AI with the goal of developing an AI also capable of playing backgammon.

Aims:

To develop an AI capable of playing backgammon

To gain an insight into AI techniques such as neural networks and genetic algorithms

To develop a piece of software, of a professional standard, from initial inception to final compilation

Objectives:

Objective 1: Investigate the AI methodologies available. I will also need to assess each one on its merits and determine which will be most appropriate for developing a backgammon solving AI.

Objective 2: Investigate into the solutions used in the pre-existing TD-Gammon.

Objective 3: Develop a Backgammon game which will allow human players to play a game of Backgammon using a language for rapid prototyping. This will also need to be capable of integration with an AI.

Objective 4: Design and test a series of AI algorithms with the backgammon program to determine the more suitable AI solutions.

Objective 5: Document results of algorithm testing.

Objective 6: Maintain a professional level of programming, using correct techniques and providing comments.

Objective 7: Run the chosen AI algorithm through the necessary learning phase.

Objective 8: Test the finished product with a bench mark, or human player.

Objective 9: Document the final results.

Problems:

With this project problems could occur at any stage of the development. However, from what I know of the proposed task I have assessed several stages which could be the most damaging to the overall project, if a problem was to occur. I have also considered possible ways to get around these stages if a problem does occur.

The first two objectives are purely research and will not cause any problems.

The third objective shouldn't cause too many problems as I have previously developed similar board games designed for two human players, and I have a complete understanding of how backgammon is played.

A problem in the fourth objective shouldn't cause too much difficulty to overcome. However, if a major problem was to occur I could scale the project back to using simpler algorithms such as playing against a greedy algorithm. This will only result in a very basic AI, but it could still play a game.

The sixth objective isn't necessary for a working solution. This objective was included as part of developing a piece of software to a professional standard. If I am behind schedule I will miss this objective out as the program will still be functional.

In order to complete a working AI, I will need to run it through learning phase. As such a problem at this stage would cause serious issues, but as this stage could take some considerable amount of time, I intend on getting to this stage as soon as I can. This will give me plenty of time to find a solution to any problems and still run the necessary algorithms through the learning phase.

Requirement List:

For this project I will not need any extra hardware or specialist software. I will need access to a computer with the processing power required for the software I will be running. For this, I have the option of using the computers available in the university or my home computer. I will also need access to the software for writing and compiling Java. Again, I have access to this in university and at home. I will also need access to a journal regarding the design and the algorithms used in TD-Backgammon. I will need access to further information on AI. For this I can use the university library and some material that is available online. I will also be attending a course about AI next year and I can use resources from this in my project development.

2. Literature review:

In this section I will be looking at and summarising different sources and we will be addressing several objectives. The first objective is to establish a basic understanding of the terminology that I will be using. the second objective is to get a background understanding of the programming techniques I will be using in the development of a Backgammon solving AI. The third objective is to look at similar backgammon solving AI, the techniques they used and how they will be related to my work. The final objective will look at addressing any ethical and social issues with my proposed development.

2.1. Genetic algorithms:

As there is no current agreed upon definition for genetic algorithms, this section will attempt to create a definition for the development of this project. A genetic algorithm or GA can be seen as any software solution that uses a model of natural selection to solve a problem based around optimisation. Genetic algorithms were originally designed to address problems that were on such a large scale that a brute force approach would not be possible or feasible. Another way of identifying a genetic algorithm, according to Mitchell (1999), is by looking at the four major components present in all genetic algorithms. These four are: a population of chromosomes, a fitness function that will select the most suitable chromosomes for survival and disregard the rest, a crossover from the suitable chromosomes to create new offspring and finally, a random mutation in the offspring. The Chromosomes are simply a way of describing a set of possible solutions. The fitness function is used to provide a number, usually between 0 and 1, to help identify the best solution in any given set. This is then used to select the best chromosomes to be reproduced. The crossover is when elements from two randomly selected chromosomes are crossed over to create two new chromosomes, each having elements of both parents. The standard crossover is when a position is selected in the genome, this determines where the genes from the first parent end and the second parent begins. E.g. the chromosomes 10000100 and 11111111 could create the resulting chromosomes 11100100 and 10011111, as described by Shiffman (2012). Other crossover methods include ordered crossover, in which all the values are used (this is used for permutations) and uniform crossover, in which genes are selected at random for crossover, as described by Mitchell (1997) . A random mutation is also needed in a GA. This is done by selecting a random value from a random chromosome and altering its value. This is often done at a lower rate as too much randomness can cause the loss of some of the better attributes.

GAs have multiple selections methods. A selection method is the method in which the best fitness is selected. One selection is a roulette wheel, in this method, the chromosome with a higher fitness will have a higher chance of being selected to added to the next generation Chambers (2001). Roulette wheels can also be used in the inverse, removing the chromosomes with the lowest fitness. Another selection method is tournament selection, in this a sample of the population is selected. Then the one with the highest fitness passes and the others are removed. An element of random chance is present in the sampling of the population. This could affect the performance of the GA as chromosomes with a high fitness may be removed if they are in a sample with a chromosome with a higher fitness.

2.2. MiniMax:

MiniMax is an algorithm that can often be used when modelling two player behaviour in zero sum games, such as chess. The goal is to make a decision based on the idea that the player will always try to minimise the AIs maximum output. So, the decision is not about trying to get to the strongest possible position, but to make sure that the opponent does not put the AI in the worst position. This could be seen as a way of avoiding making risky moves. A MiniMax algorithm could also be implemented in Backgammon decision making to minimise potential losses. However, Minimax is better implemented in decision trees according to Hauk, Buro and Schaeffer (2004).

2.3. Neural Networks:

An artificial neural network (ANN) is a computational model based on the workings of neurons in the brain, with the aim of solving problems. It works by modelling a collection of nodes or units. These are sometime referred to as neurons and should help with the understanding of how the ANN functions. These neurons take a series of inputs and will multiply them with a weight. In this case let's assume it is a random weight. The sum of the weighted input values will then be passed to the neuron's activation function (Shiffman 2012). This activation function acts as a threshold. If the sum of the (inputs * weights) is greater than the activation function, then an output is made.

A neural network will need a learning strategy. The main ones are supervised learning, unsupervised learning and reinforcement learning. Supervised learning is when a teacher is present (Shiffman 2012). This teacher is normally another algorithm, will need to know more about the subject than the neural network. The network will make a guess which the teacher already knows the answer to. The teacher will then correct the neural network. The network will then compare its guess with the correct answer, and the network will adjust its weighting accordingly. An example could be a Backgammon master instructing a neural network in which moves are most beneficial at each turn of the game. This was the practice for Backgammon AI before TD-Gammon. Unsupervised learning is used in pattern recognition when no current pattern is known. Reinforcement learning is when positive or negative feedback is given to the neural network so that it can learn for next time. This practice is more common in robotics but has also been used in Backgammon in the TD-Gammon program in which reinforcement learning was chosen over supervised learning (Tesauro, 1994).

2.4. TD-Gammon:

TD-Gammon is a computation backgammon AI. It was developed by Gerald Tesauro in 1992 and built off the Neurogammon program also developed by Tesauro. TD-Gammon arose from a study into reinforcement learning and how it can be used in non-linear function approximations. TD-Gammon was intended to be compared with its predecessor Neurogammon, a neural network that used supervised learning and backpropagation learning. TD-Gammon is a self-teaching neural network. The program trains by playing a number of games of backgammon against itself then learning from the outcome. This is an example of the reinforcement learning discussed in the previous section. The neural network has a multilayer perceptron architecture and implements a form of temporal difference learning. This means that the values of the perceptron's weights for each turn are based on the temporally successful predictions. TD-Gammon version one, was trained using 300,000 games and with this alone managed to surpass Neurogammons capabilities, with its level of play being that of an intermediate Backgammon player. Eventually it became possible to train TD-Gammon for longer and provide a greater depth of search. TD-Gammon version 2.1 trains for 1.5 million games and has a 2ply lookahead algorithm. This version managed to achieve expert level play and could compete with the top human players in the world, Tesauro (1994), Tesauro(2002). As a result, a lot of research into the program and how it plays has been conducted by top Backgammon players. This has changed and improved a lot of the strategies implemented by champion players.

TD-Gammon's performance is an interesting find, as I will be intending to develop a program that utilises a self-learning method. The testing of TD-Gammon showed a large amount of learning took place in the first few thousand games. During this stage is when the basics are being developed, concepts like attacking the opponent and not leaving blots vulnerable. The more advanced tactics are developed later on in its learning. The final stage shows a steady gradient, slowly increasing its level of play, before reaching a peak at 50,000 games. At this stage it was considerably better than its benchmark test subject. The testing also showed that the results increased as the network was scaled up. So as the number of hidden nodes in the network increased the resulting performance increased.

2.5. Backgammon, Minimax and Heuristics:

As discussed previously, Minimax algorithms are used to model opponent behaviour in two player games. Minimax is used to minimise the player losses. The first ever Backgammon AI agent, BKG by Hans Berline operated using a Minimax algorithm. This was also used in TD-Gammon. Backgammon is a game that also incorporates a random chance element in the form of dice rolls. To deal with this in a Minimax algorithm, chance nodes also have to be used between each MIN and MAX layers. This way the 21 possible dice combinations can be modelled and an expected result can be calculated. A heuristic evaluation function will also be used to determine the value of the terminal nodes.

Heuristic functions in Backgammon AI software are used to evaluate whether a move is a good move or a bad move. In the case of a MiniMax algorithm the heuristic function will produce a value rating the moves. These values will be calculated based on a range of features that each move in Backgammon will have. For example, a move that will move a checker closer to the end point can be seen as a good move, however if this move will leave another checker vulnerable to attack, this move will also have negative side effects. As such the heuristic evaluation will have a series of weightings for each value. In a genetic algorithm these will be the elements that will be changing in each generation. In the case of a Backgammon AI using only a MiniMax algorithm, these values will remain fixed. This will explain why BKG had difficulty playing against players with very little skill. In Felice and Ciolfi Felice (2012) the heuristic function provides both basic and advanced strategies using temporal context sensitivity. The AI also calculates the heuristics based on the play (the combination of moves) as opposed to individual moves. Depending on the stage of the game, the MiniMax algorithm operates at different breadths and depths. This is to account for the large amounts of possible moves with low stakes in the early game, in contrast with the late game stages. Also, the heuristics are used differently at different stages of the game. They are categorised into three groups, bear off, re-entry and basic. This will be partly due to the rules of Backgammon that will prevent a play that does not re-enter a checker when one is on the bar. Likewise, the bear off stage of the game will make many of the basic heuristic values redundant e.g. the formation of a prime becomes irrelevant at this stage.

2.6. GP-Gammon: Genetically Programming Backgammon:

This paper covers the development of a GA that plays Backgammon. The majority of Backgammon AI is based around neural networks. It has been stated in this paper that neural networks have a fixed topology and must be designed before any testing can be done to decide on the best topology to use. Azaria and Sipper (2005), have decided to use Genetic Programming in their Backgammon AI solution, as it allows for the structure of the program to change. The genetic programming is achieved using LISP sub expressions. These sub expressions are made up of terminals and functions, which receive multiple inputs, compute them and then output the result. The terminals are zero-argument functions that act as constraints as well as providing a way to query the backgammon board state. The searching algorithm works by making the dice roll and then evaluating each board state from the collection of possible board states. It then uses the one with the highest score.

Because Backgammon has different stages, each of which will require a different style of play, GP-Gammon has split the game into two main stages. These are the contact stage and the race stage. These stages can be seen as the main game and the bear off stage of the game. In the first, tactics such as blocking the opponent using a prime and not leaving a blot open are encouraged. The race stage is in the late game when the program will start to bear checkers off the board and the aforementioned tactics will become irrelevant. Both these stages are represented using a tree data structure. GP-Gammon operates using a genomic structure of individuals. These individuals will contain a contact tree and a race tree. During the evaluation stage the program checks if the game is in the contact or race stage. If it is in the contact stage it will complete the evaluation using the contact tree, and if it is in the race stage it will complete the evaluation in the race tree. It is worth noting that GP-Gammon trains both of these trees together in the same game to develop a complete individual as opposed to generating a pure race game board for them to train on.

GP-Gammon uses four different operations to create the next generation these are, identity, sub tree crossover, point mutation and MutateERC. For the selection function GP-Gammon would select a small section of the population then find the individual with the highest fitness, based on the programs fitness function. The program would run for 500 generations and every 5 generation would play the top four individuals against Pubeval, a public domain board evaluator developed by Gerald Tesauro. This practice of playing against an external opponent was thought to have reduced the generalisation of the program, making it suited only for playing against Pubeval. Because of this the program was then entered into a self-learning phase in which the program would play against itself, using the population of individuals that it has generated. The results of this was a much stronger Backgammon player than the training on external opponents. This can be seen when the training using self-learning has a maximum benchmark performance against Pubeval of over 50% whereas the externally trained program has a benchmark of less than 50%.

2.7. Final thoughts:

In this literature survey I have explored some ideas on how the Backgammon problem has been addressed previously. One of the key elements that I have found is that a self-learning algorithm can often exceed people's expectations and can outperform a lot of other Backgammon software. This is presented by both papers discussing TD-Gammon and the one paper discussing GP-Gammon, both of these programs were developed around the idea of self-learning, with the TD-Gammon going on to achieve an expert level of play at the Backgammon world cup tournament in 1992. This survey has also given me some themes which are present in each of these backgammon programs, and which will help me in the development of my own program. The discussion in GP-Gammon (2010) and the, Teaching Algorithms: Backgammon Resolution Using A MiniMax Strategy (2012), both made it clear that the program will benefit from distinguishing stage of the game it is in. Either the contact stage or the race stage. It has also shown how a variation on the MiniMax function that takes into account random chance could be a viable tool for evaluation in my program. I have also looked into the neural networks and genetic algorithm approaches to solving Backgammon games. For my project I will be making the decision to develop it using genetic algorithms. As shown from the TD-Gammon, a very large dataset will be required to properly train it even to an intermediate level. And as shown by the GP-Gammon, which operates using very similar principles to genetic algorithms and was trained using self-learning, a self-learning GA could have some real potential at playing Backgammon.

On the topic of ethical and social issues, I believe that my intended plans, based on the previous work in this field, will not have any major social or ethical implications. All these programs can be seen as self-contained. They do not store personal information about people and they are not used for decision making in any life or death situations. Most of these programs were developed to see how different AI techniques can be applied to Backgammon and my program will be no different.

3. Design:

In this section I will be discussing the design of the Backgammon AI solution. I will be using a hybrid of prototyping development methodology and the waterfall model for this project. I have chosen this because I am developing this project alone, from my own specification. As such it would not be appropriate to use methodologies such as agile, which is more suited for teams. Prototyping will also be beneficial as this is a novel project to me. This will be both my first backgammon game, as well as my first genetic algorithm. The project will be split into three sections. The first is a two-player backgammon game. The second is a heuristic algorithm which will be used to evaluate each movement. The third section will be a genetic algorithm which will be used to evolve the weights in the heuristic algorithm. This modularisation of the development will lend itself to the waterfall model. Each stage in the waterfall will be a section of the project. Each stage will be prototyped. The first stage will be the setting up of the two-player backgammon game. prototyping here will be slowly adding new features until the game follows all the rules of backgammon. Once this is complete I can then move onto the heuristic algorithm and start the prototyping in that section, and so forth.

3.1. Two Player Backgammon Game:

The two player Backgammon game will be set up as a platform to develop the AI Backgammon solution. This game will be using the following objects, Counter, which will be used to represent the counters on the backgammon board, Point, which will be a stack of counters, and a GameBoard object, which will extend the JPanel class. The GameBoard will have an ArrayList of Points. The position in the ArrayList will correspond with the point on the board, with 0 being the “Home” for player 2 and 25 being the “Home” for player 1. This will provide a way of visually representing the backgammon game. At the start of a game, the roll button will be pressed generating two random numbers between 1 and 6. The highest of the two numbers will set which player has the first turn. These two numbers will also be used to determine the possible movements. This will be done by the player selecting a point which they control, then all available points will be highlighted and set as available. This will be done based on the original points position in the ArrayList, adding or subtracting the dice values will find the available points. Selecting a point which the player controls will also pop a counter from the top of the counter stack. This will be the selectedCounter. After selecting one of the available points to move to, the selectedCounter will be pushed into the counter stack. The dice values will be adjusted for the movement selected by the player. Once no more movements are available, either because of lack of dice values, or because points are occupied by the opponent, the roll button will be pressed again, generating new values and changing player control.

Pseudo code available Fig.1, class diagrams Fig.2, flow chart Fig.3, in section 3 in the appendix.

3.2. The Heuristic Greedy Algorithm:

The heuristic algorithm/ greedy algorithm is an algorithm which will generate all possible movements for the dice rolls. It will then perform a series of calculations on each possible movement to find ten parameters which will help identify the best possible movement. This will be done with a class called MovementGenerator (MG). I have identified this as a greedy algorithm as it will be making the simple optimum choice, based on these calculations, at each stage without any kind of machine learning taking place. It will later have the genetic algorithm incorporated into it. The first 5 parameters are positive and result in a better movement, and the other 5 are negative ones which will result in a worse movement. These parameters are:

1)The number of bear offs

2)The number of safe counters

3)If a prime exists

4)How many counters are on the inner board

5)If a move hits an opponent’s counter

6)The total distance from the end

7)If a point has more than five counters on it

8)How many counters are on the home board

9)If a counter moves more than 6 points away from all other counters

10)The distance between the lowest controlled point and the highest controlled point

The positive parameters will be summed up and the negative ones will then be subtracted from this value. This will generate a fitness value (note this is not the same fitness value that will be used in the GA). These values will be assigned to each movement, with the best movement being the one with the highest fitness value. The parameters 2 and 3, encourage a more defensive play. Parameter 2 encourages the MG to select a movement which will keep points defended with two or more counters on them, and parameter 3 will make it more likely to set up 6 safe points in a row creating a prime and blocking the opponent. Parameters 1 and 4 are designed to encourage the MG to get to the end game stage as early as it can. Finally, parameter 5 will encourage the movement to take enemy counters where possible. This supports a more aggressive play. All these parameters can be considered a good play style. The following five parameters are associated with a poor play style. Parameter 6 indicates how far each checker needs to move to win the game. The lower the total distance the higher the fitness value. Parameter 7 is a common indicator of a poor play style in backgammon. Having more than 5 checkers on a single point makes mobility difficult, it also means you are occupying less points, making it less likely to block the opponent. It also makes it easier for you opponent to block you. Parameters 8 and 10, help prevent the movement generator from continuously leaving counters behind and encourages it to move counters up together as a group. This is beneficial as it is more likely to block your opponent and it is less likely you will be blocked. It also makes a prime more likely to form. Parameter 9 discourages moving a single counter out of range of all other counters. This would leave a counter exposed and more likely to be taken by the opponent. It also helps encourage moving the counters up as a group like parameters 8 and 10.

I found many of these attributes in (Felice And Ciolfi Felice, 2012).

Fig.4 in section 3 of the appendix shows the class diagram.

3.3. Genetic Algorithms:

For the genetic algorithm section, I will be exploring two main selection methods. These will be broken down in the next two sections. The genetic algorithm will work with the previous, heuristic/ greedy algorithm. One of the benefits of GAs as mentioned in section 2, is that they are modular and can be easily added into another section of code, in this case the MG. The MG will have an overloaded constructor, which will allow an array of weights to be passed in. These weights will be multiplied by the 10 parameters discussed in the previous section. The evolution will take place on these weights. This will change the prominence of each parameter when evaluating the best movements and so will provide different results. Both selection methods I will be covering will only differ in the selection process, mutation rate and crossover will work in the same way.

The genetic algorithm will be made up of a population object. This object will contain an ArrayList of Individuals. The Individual is an object which will contain the weights. The Individual object will also contain the mutation and crossover methods. This is to keep the main portions of the GA in a self-contained unit. The mutation method will be passed a mutation rate. When the mutation method is called, each weight in the individual will be passed over and a random number between 0 and 1 will be generated. If the number is lower than the mutation rate, then that weight will be changed to a random number. The Individual will also contain the crossover method. This will take in another Individual, called Individual2, and generate a random integer between 0 and 10 (the size of the array of weights). This is the crossover point. All weights before the crossover point stay the same and all the weights after the crossover point, are changed to the corresponding weights in Individual2.

Fig.5 in section 3 shows the class diagrams.

3.3.1. Roulette Wheel:

For the roulette wheel selection, individuals will play against other individuals in the population at random. The distance travelled by all counters is calculated once the game is won. This distance is 375 minus the total distance from end to end for all counters. This will be the fitness of the Individuals. However, (Azaria And Sipper, 2005) use 167, the distance to the end, for all counters, at the start of a new game. For the winner, the distance travelled will be 375. The distance travelled will be the number of slots each Individual will get within the roulette wheel. The roulette wheel selection will then pick random slots in the roulette wheel and add the Individual at that slot to the mating pool. This will be done until the mating pool is full. Crossover will then be done by selecting two Individuals at random out of the mating pool, with the result being added to the new genepool. This will be done until the genepool is full. Once crossover is complete the mutation will take place. This will be done by calling the mutation method on each Individual in the genepool.

3.3.2. Tournament Selection:

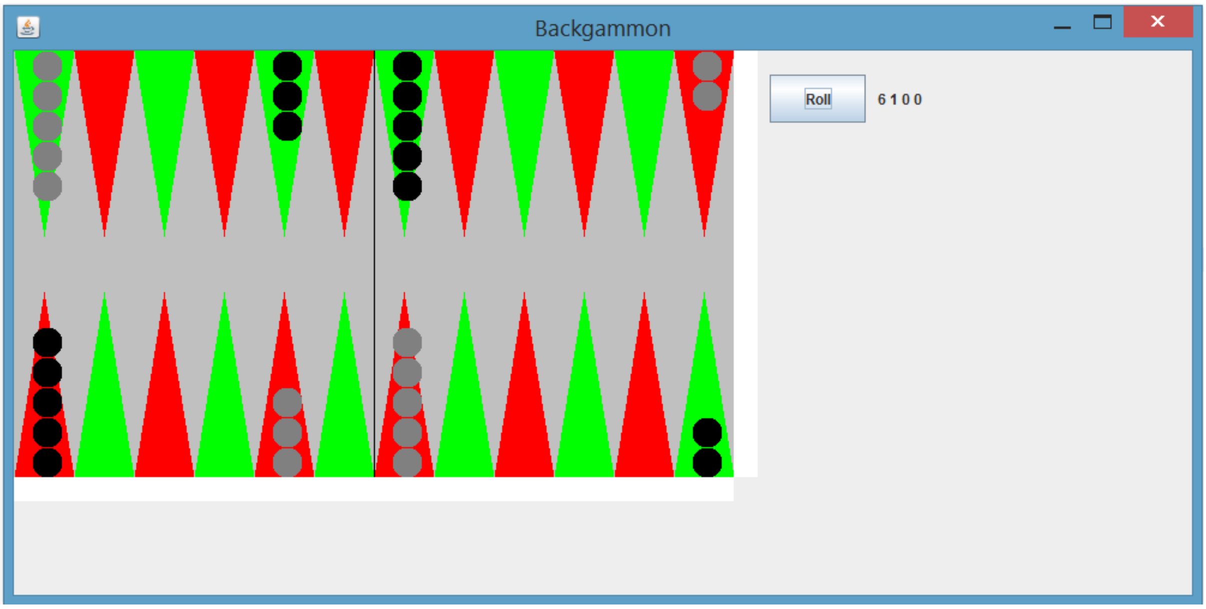
Tournament selection will be done by simply playing Individuals off against each other with the winner being added to the mating pool. A typical tournament selection takes a number of Individuals, of a specified tournament size. Then the Individual with the highest fitness is then added to the mating pool. In the case of backgammon, as it is a two-player game, with the best indicator of fitness being the distance travelled, half the population will have the same fitness. As such I decided to simplify the selection process and just add the winning Individual to the mating pool. This will be the same as running a typical tournament selection with a tournament size of 2. It also prevents Individuals with the same fitness being selected together.

4. Implantation:

In this section, I will be describing the implementation of the Backgammon AI solution. Like the design section I will be breaking this down into three main sections, the two-player Backgammon game, the heuristic/greedy algorithm, and the genetic algorithm.

4.1. Two-player Backgammon:

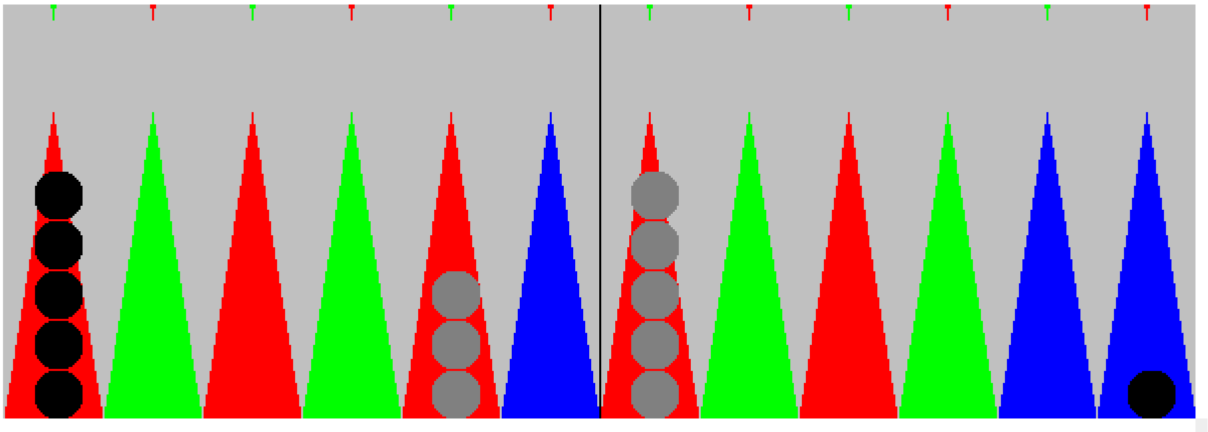
The two-player Backgammon game is implemented as a JPanel object from the javax.swing library. This will allow for a graphical user interface to be used for the Backgammon game. The JPanel has a method called paint, which allows for drawing components onto the JPanel. By overriding this method, I can draw all the components I will be needing. The Point object discussed in section 3 has x and y co-ordinates as parameters. These are used to plot the graphics on the JPanel. These locations also help draw the counters for each point. A white bar is placed horizontally at the bottom of the main board section. This is the bar which stores counters which have been hit, and are waiting to re-enter the board. The white bar placed vertically on the right of the main board section is the "home" section, this is where counters are shown after bearing off. See Fig.1 in section 4 in the appendix for the code. A button using the JButton object is added to the board. This allows the players to roll the dice. As you can see from this image, the roll resulted in a 6 and a 1. The two zeros show that a double wasn't rolled. As 6 is the first value, player one moves first, in this case it is black.



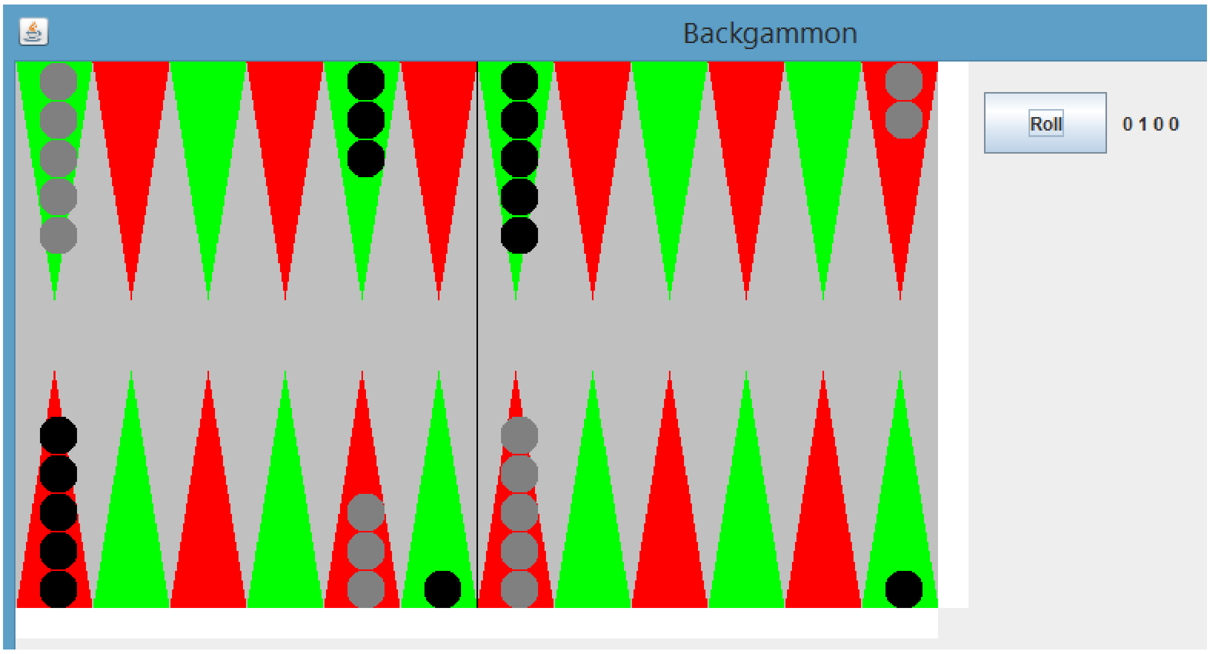
Clicking on the board will activate the mouseClicked method implemented with the MouseListener interface. The mouseClicked method determines the x,y co-ordinates of where the mouse was clicked. It then will determine the next operation that needs to be conducted. This means whether a counter will need to be selected from a point, to be moved, or if the counter has already been selected and the mouse click is indicating the location which it will be moved to. This is done using the selectedCounter object. If a counter is already selected, the id of the selectedCounter will be between 0-29 (30 counters). However, if a counter has not been selected, then the selectedCounter object will be a new Counter, which by default start with an ID of 500. See Fig.2 in section 4 in appendix for the mouseClicked code.

If a point is being selected for moving a counter from, then the availablePoints(Point p) method will be called. This method loops through all points and all dice values. This is to check if a point can be reached using the dice roll, from the original point, p passed to the method. Once a point is found, it can be tested to see if the opponent has control of that point, and to see if more than one of the opponent's counters occupies that point (meaning it cannot be taken). All available points are then set to available, this means they will be highlighted in blue in the paint method. See Fig.3 in section 4 in the appendix.

Here you can see that player 1 has selected the point 1. This has popped the top counter from point 1 (now selectedCounter) and highlighted the available points. A 1 and a 6 were rolled so points 2 and 7 are highlighted.



The selectedCounter was moved to point 7. This was done going through the mouseClicked method, like the previous time, however as the selectedCounter is the popped counter from point 1, the moveChecker(Point p) method is called. This method mainly pushes the selectedCounter to the point identified by the mouse click. The method checks if the move will hit an opponent's counter. This is to check if the opponents counter needs to be added to the bar, before the players counter is added to the point. Once this is done, the dice values are adjusted for the move. See Fig.4 in 4 for the code.



If a counter is taken by an opponent it will be added to the bar. A player cannot then make a move until all their counters are taken off the bar and placed onto the game board. This is done in the mouseClicked method, by checking the player turn and checking the size of the corresponding bar stack. If the player is in the bearing off stage, then the player will be allowed to move a counter into the "home" section. This test is being done using the MovementGenerator, which will be covered in the next section. In the initial prototype, no checking method was used. This was purely to help me with bug testing.

4.2. Heuristic/Greedy Algorithm:

This section will be covering the implementation of the MovementGenerator class. The movement generator works by calculating which moves can be made. The MG has two constructor methods, the first is used for this section, the second is used with the genetic algorithm section. The difference in the constructors is, the constructor for this section does not take in any weights (an array of doubles). Instead it sets all the weights to 1. This will be explained later.

The MG has a method called generateMovement, this returns the best movement based on a calculation discussed briefly in the section 3 which I will expand on in this section. The MG generates all possible movements using a similar algorithm to the one used in the availablePoints method in 4.1. This algorithm, instead of setting valid points to available, creates an object called Movement. The Movement consists of the original position, the position moved to, the difference (this is to find the dice value which corresponds with this movement) and the movement value (the result of the MGs calculation, this is how the best movement is found). Once the Movement is created, it is added to an ArrayList of Movements. All the Movements with a difference of 0 will be removed from the list. This removed all Movements which do not move the counter. The values for each move is calculated using the boardValue method. Finally, the highest value is found using the getHighest method, and then returned.

The boardValue method is passed a Movement, the boardValue method makes a clone of all the objects in use on the board (Points, Counters, CounterStacks). This allows the move to be applied without affecting the main game. This means that the value of the Movement is based on the state of the board, once the move has been applied. Cloned Points are then passed into the following methods.

1) numberOfBearOffs

Returns the number of counters in the "home" section for the corresponding team.

2) numberOfSafeCheckers

Counts all the counters on points containing two or more counters.

3) distanceFromEnd

Returns 375 – (i * n)   Where i is the point location and n is the number of counters on the point.

4) existanceOfPrime

Loop through points, if a point has a two or more team counters on it, start counting. If a break is found, count = 0. If count gets to 6, a prime is found. Return 1.

5) greaterThanFive

Loop through each point. If a point contains more than one counter, add the overflow to the count. Return the count.

6) innerBoard

Count the number of counters on each point in the inner board.

7) moveApart

Loop through all points. If a team counter is found, check the next six points for team counters. If one is found, stop counting and go back to looping though points. If a team counter is not found, add one to the count. Once the loop is complete, return the count.

8) homeBoard

Count the number of counters on each point in the home board.

9) boardWidth

loop through all points. Make a note of the first team counter location. Make a note of the last team counter. subtract the first location from the last. Return the result.

These calculate the parameters from 3.2. Checking for a hit on an opponent is done in the boardValue method itself, when applying the move.

The getHighest method starts with an extremely low number as the best value, it then loops through all the Movements in the movement list. If it finds a movement with a higher value best value set earlier, set that movement as the new best.

4.3. Genetic Algorithm:

The genetic algorithm is contained within a class called Population. This again comes back to the prototyping development, and how I modularised the code. The Population class also has a number of Individuals. The Individuals were discussed in section 3.3 and have not differed from their design. The Population class extends threads from the main java library. This was to help in development to separate the running of the games, to the other parts of the program. The run method has a select case algorithm. This will make it easier when implementing different tests. I just need to change the selection variable.

Case 0) This runs the roulette wheel selection algorithm. Once this is complete it writes the finished genepool to a CSV file.

Case 1) This runs the tournament selection algorithm. Once this is complete it writes the finished genepool to a CSV file.

Case 2) This was used for testing, creating a new population from genepool read in from a CSV.

Case 3) This was used for testing, it reads in an initial genepool from a CSV, then reads in the roulette wheel genepool from a CSV. It then plays the two genepools against each other and outputs the results.

Case 4) This is the same as the previous case but plays the tournament genepool against initial genepool.

Case 5) This case plays a genepool, read from a CSV against the heuristic/ greedy algorithm for multiple games to reduce the effect of random chance from the dice rolls. It then outputs the results into a results CSV.

Case 6) This case plays two genepools, both read from CSVs, for multiple games. It then outputs the results into a results CSV.

The rest of the population class was developed almost identically to the design. I will highlight the difference however. When running the roulette wheel selection in its initial tests, I noticed that no evolution was taking place. The values were changing, but no improvement was being made. I suspected that this was due to too many slots in the roulette wheel being given to Individuals which lost. As a result, I changed the code to double the amount of spaces for the winning Individual. This resulted in successful improvement in the next tests. The Population class also needed methods for writing and reading CSV files.

4.5. Conclusion:

In section 3 and section 4, I have shown the design and development of the Backgammon AI solution project. As can be seen many of the implementation elementals follow the design, except for when specified. I did however decide against the separate race and contact stages from section 2, as well as the MiniMax algorithm. This section has been very technical and I believe it would be best to follow the code in project provided in the appendix. I would also like to make a note of a few things. I am happy with the overall design of the main board. However, I have noticed that this leaves too much blank space. A second build could use this to adjust the GA parameters. However, for my purposes, changing them at run time is sufficient. This would also allow me to remove the Game class, which was only in use for the two-player backgammon game. The GA and greedy algorithm can be run from the Population class. See Fig.1 in section 4 of the appendix for the full UML class diagrams.

5. Evaluation:

This section is split into two subsections. In part 1 I will be testing the project. I will explain what is working as well as making notes of the errors. In part 2 I will be discussing the results data I achieved from testing my backgammon genetic algorithm project. This will explore the effect of selection method, the mutation rate, population size, and generations on building an optimised solution. I will be evaluating all the data by playing them off against the greedy algorithm. This is to establish a suitable bench mark.

5.1. Testing:

The project has been implemented as shown in section 4. I noted in this section on where the design had differed from the original design as well as how the design was translated to the implementation. For the testing of my project I chose unit testing. This is due to the prototyping/ waterfall style methodology discussed in section 3. The modularisation makes unit testing more suitable.

5.1.1. Two-Player Backgammon Game:

For the first test I will be testing the dice roll button. Fig 1 in section 5 in the appendix will have the list of moves output in the console. This test table shows the first 8 movements. As can be seen the first dice value is lower, so player 2 goes first. You can also see that on move 6, player 1 makes a hit to player 2. This forces player 2 to re-enter a counter from the bar on move 7. Move 7 and move 8 both show how rolling a double, is equal to rolling 4 dice of the same value, giving the player 4 movements. This demonstrates that the rules of backgammon are being followed.

Test	Player	Dice	Expected Move 1	Expected Move 2	Actual Moves	Passed
1	2	1,4	24-20	24-23	24-20,24-23	Yes
2	1	3,2	1-4	1-3	1-4,1-3	Yes
3	2	4,2	23-21	20-16	23-21,20-16	Yes
4	1	5,3	4-9	9-12	4-9,9-3	Yes
5	2	6,3	21-18	13-7	21-18,13-7	Yes
6	1	1,2	3-5	17-18	3-5,17-18	Yes (Hit)
7	2	3,3,3,3	Bar – 22, 16-13	7-4, 6-3	Bar(25)-22,16-13,7-4,6-3	Yes (Re-entry)
8	1	2,2,2,2	5-7,7-9	9-11, 12-14	5-9,9-11,12-14	Yes

5.1.2 Heuristic algorithm:

For this I will be testing the calculations of the MovmentGererator. This will test whether the workings out of the MG are working correctly. I will also show which Move it decided to make. Move shows which points the counter will be moved between. B is the number of bear offs it will result in. S is the number of counters which will be safe, D is the distance from the end. G is if a move puts more than 5 counters on a single point. I, is how many counters will be on the inner board. A is if this move takes two counters more than 6 points part. H is how many counters are on the home board. SP is the span of the board, how far from counter 1 to counter 15. Eval is the evaluation result. The dice roll for this is 5,2 and player 1 is moving first. I am not checking for a prime in these tests as these are a rare occurrence and won’t appear in the first turn.

	Move	B	S	D	G	I	A	H	SP	Eval
Test	1,3	0	13	165	0	5	0	2	18	-167
Expected	1,3	0	13	165	0	5	0	2	18	-167
Test	12,17	0	15	162	0	5	0	2	18	-162
Expected	12,17	0	15	162	0	5	0	2	18	-162
Test	12,14	0	14	165	0	5	0	2	18	-166
Expected	12,14	0	14	165	0	5	0	2	18	-166
Test	17,22	0	14	162	0	6	0	2	21	-165
Expected	17,22	0	14	162	0	6	0	2	21	-165
Test	17,19	0	15	165	1	6	0	2	18	-165
Expected	17,19	0	15	165	1	6	0	2	18	-165
Test	19,21	0	14	165	0	5	0	2	20	-168
Expected	19,21	0	14	165	0	5	0	2	20	-168

Expected Selected Move: 12-17 Actual Selected Move: 12-17

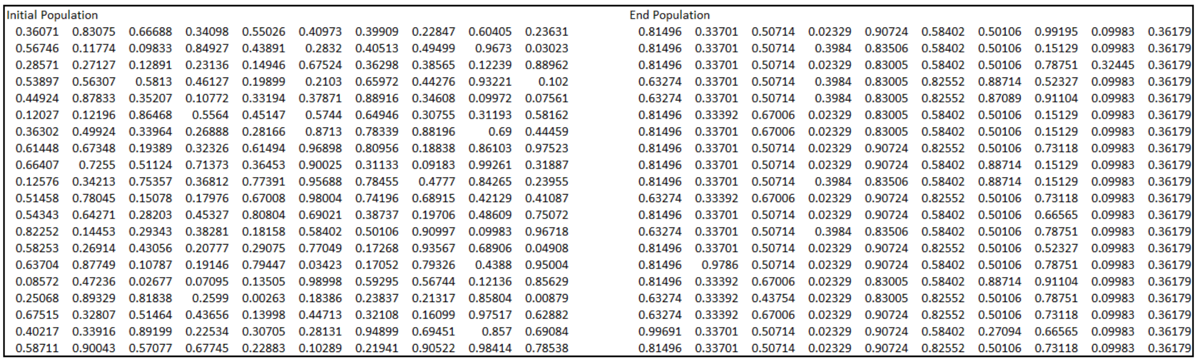
	Move	B	S	D	G	I	A	H	SP	Eval
Test	1,3	0	13	160	0	5	0	2	18	-162
Expected	1,3	0	13	160	0	5	0	2	18	-162
Test	12,14	0	14	160	0	5	0	2	18	-161
Expected	12,14	0	14	160	0	5	0	2	18	-161
Test	17,19	0	15	160	1	6	0	2	18	-160
Expected	17,19	0	15	160	1	6	0	2	18	-160
Test	19,21	0	14	160	0	5	0	2	20	-163
Expected	19,21	0	14	160	0	5	0	2	20	-163

Expected Selected Move: 17-19 Actual Selected Move: 17-19

See Fig.2 in section 5 in the appendix for the output.

5.1.3 Genetic Algorithm:

To test the genetic algorithm, I have decided to first show that the population is evolving, and second that this evolution results in an improvement in Backgammon playing ability. I decided that unit testing in this section would not give much information. The genetic algorithm produces a huge number of parameters, and takes many generations until it can be verified that it is working. Also, the nature of genetic algorithms means that no expected outcomes can be determined. Here you can see a sample of the initial population. These started out as random doubles. They have no identifiable ordering or pattern. However, in the end population, you can see that many of the values share the same values on certain attributes, this shows that selection and crossover is occurring. Some of the attributes are completely homogenous apart from a single instance, this is evidence of mutation occurring. This training was done using a population of 100, a training time of 100 generations, and a mutation rate of 0.005.



Initial Population											End Population									
0.36071	0.83075	0.66688	0.34098	0.55026	0.40973	0.39909	0.22847	0.60405	0.23631		0.81496	0.33701	0.50714	0.02329	0.90724	0.58402	0.50106	0.99195	0.09983	0.36179
0.56746	0.11774	0.09833	0.84927	0.43891	0.2832	0.40513	0.49499	0.9673	0.03023		0.81496	0.33701	0.50714	0.3984	0.83506	0.58402	0.50106	0.15129	0.09983	0.36179
0.28571	0.27127	0.12891	0.23136	0.14946	0.67524	0.36298	0.38565	0.12239	0.88962		0.81496	0.33701	0.50714	0.02329	0.83005	0.58402	0.50106	0.78751	0.32445	0.36179
0.53897	0.56307	0.5813	0.46127	0.19899	0.2103	0.65972	0.44276	0.93221	0.102		0.63274	0.33701	0.50714	0.3984	0.83005	0.82552	0.88714	0.52327	0.09983	0.36179
0.44924	0.87833	0.35207	0.10772	0.33194	0.37871	0.88916	0.34608	0.09972	0.07561		0.63274	0.33701	0.50714	0.3984	0.83005	0.82552	0.87089	0.91104	0.09983	0.36179
0.12027	0.12196	0.86468	0.5564	0.45147	0.5744	0.64946	0.30755	0.31193	0.58162		0.81496	0.33392	0.67006	0.02329	0.83005	0.58402	0.50106	0.15129	0.09983	0.36179
0.36302	0.49924	0.33964	0.26888	0.28166	0.8713	0.78339	0.88196	0.69	0.44459		0.81496	0.33701	0.67006	0.02329	0.83005	0.58402	0.50106	0.15129	0.09983	0.36179
0.61448	0.67348	0.19389	0.32326	0.61494	0.96898	0.80956	0.18838	0.86103	0.97523		0.81496	0.33701	0.50714	0.02329	0.90724	0.82552	0.50106	0.73118	0.09983	0.36179
0.66407	0.7255	0.51124	0.71373	0.36453	0.90025	0.31133	0.09183	0.99261	0.31887		0.81496	0.33701	0.50714	0.02329	0.90724	0.58402	0.88714	0.15129	0.09983	0.36179
0.12576	0.34213	0.75357	0.36812	0.77391	0.95688	0.78455	0.4777	0.84265	0.23955		0.81496	0.33701	0.50714	0.3984	0.83506	0.58402	0.88714	0.15129	0.09983	0.36179
0.51458	0.78045	0.15078	0.17976	0.67008	0.98004	0.74196	0.68915	0.42129	0.41087		0.63274	0.33392	0.67006	0.02329	0.90724	0.82552	0.50106	0.73118	0.09983	0.36179
0.54343	0.64271	0.28203	0.45327	0.80804	0.69021	0.38737	0.19706	0.48609	0.75072		0.81496	0.33701	0.50714	0.02329	0.90724	0.58402	0.50106	0.66565	0.09983	0.36179
0.82252	0.14453	0.29343	0.38281	0.18158	0.58402	0.50106	0.90997	0.09983	0.96718		0.63274	0.33701	0.50714	0.3984	0.83506	0.58402	0.50106	0.78751	0.09983	0.36179
0.58253	0.26914	0.43056	0.20777	0.29075	0.77049	0.17268	0.93567	0.68906	0.04908		0.81496	0.33701	0.50714	0.02329	0.90724	0.82552	0.50106	0.52327	0.09983	0.36179
0.63704	0.87749	0.10787	0.19146	0.79447	0.03423	0.17052	0.79326	0.4388	0.95004		0.81496	0.9786	0.50714	0.02329	0.90724	0.58402	0.50106	0.78751	0.09983	0.36179
0.08572	0.47236	0.02677	0.07095	0.13505	0.98998	0.59295	0.56744	0.12136	0.85629		0.81496	0.33392	0.67006	0.02329	0.83005	0.58402	0.88714	0.91104	0.09983	0.36179
0.25068	0.89329	0.81838	0.2599	0.00263	0.18386	0.23837	0.21317	0.85804	0.00879		0.63274	0.33392	0.43754	0.02329	0.83005	0.82552	0.50106	0.78751	0.09983	0.36179
0.67515	0.32807	0.51464	0.43656	0.13998	0.44713	0.32108	0.16099	0.97517	0.62882		0.63274	0.33392	0.67006	0.02329	0.90724	0.82552	0.50106	0.73118	0.09983	0.36179
0.40217	0.33916	0.89199	0.22534	0.30705	0.28131	0.94899	0.69451	0.857	0.69084		0.99691	0.33701	0.50714	0.02329	0.90724	0.58402	0.27094	0.66565	0.09983	0.36179
0.58711	0.90043	0.57077	0.67745	0.22883	0.10289	0.21941	0.90522	0.98414	0.78538		0.81496	0.33701	0.50714	0.02329	0.90724	0.82552	0.50106	0.73118	0.09983	0.36179

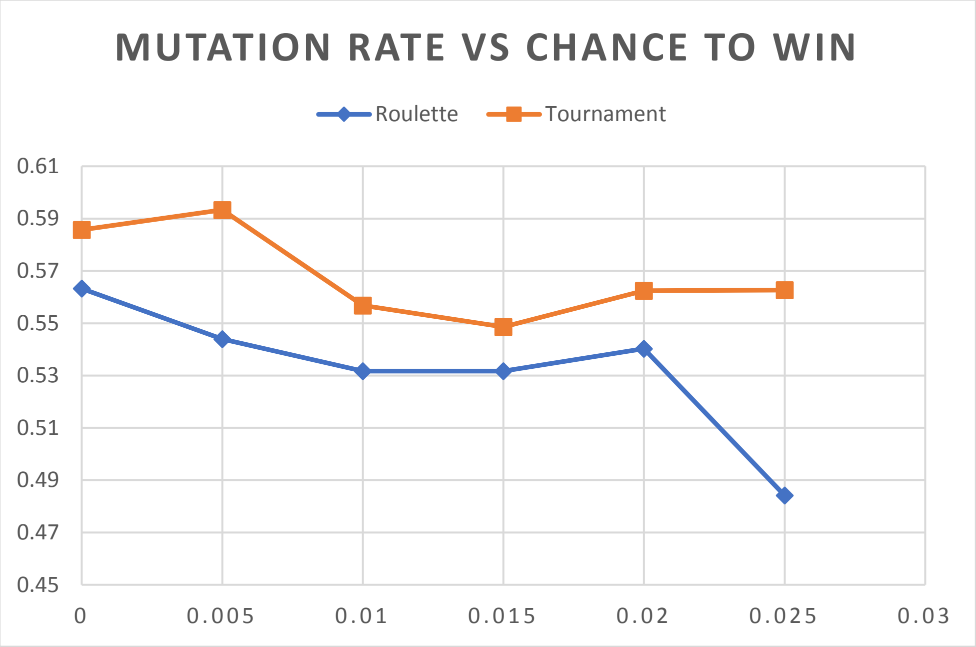
To test that an improvement is being made, I have competed the heuristic/ greedy algorithm against the genetic algorithm. I tested the whole population as I have no way of identifying the single best individual out of the population. This is because backgammon is a game of luck and attempting to get the single best individual, through a process of elimination, could result in accidently removing it. The population size is 100 although due to an error in the read from CSV method, I needed to remove the last individual to prevent a crash. Therefor these tests will be out of 99. Each test is performed 25 times. This is to reduce the effect of luck on the game. To perform tests, I have used two MG objects, one uses the weights trained by the genetic algorithm, and the other uses the default array of 1s as weights. See Fig.3 in section 5 in the appendix for table of results. This table shows that even with a small training time of 100 generations and a population size of 100, the genetic algorithm can outperform the greedy algorithm. The chance to win is the (total wins / number of tests)/ population size used in test. 0.5 meaning a 50/50 chance of winning each game.

5.1.4. Known Errors:

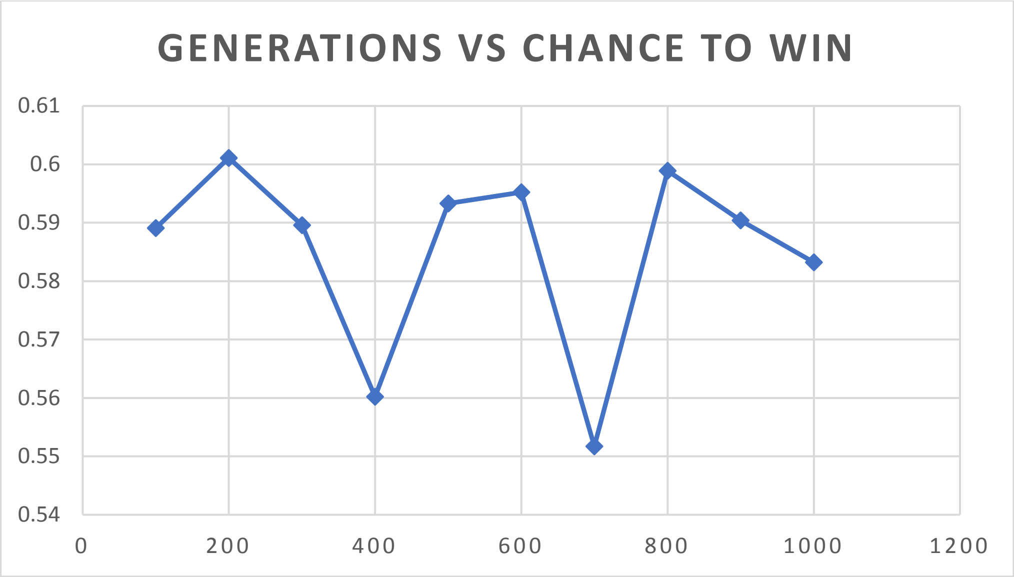
Three minor errors are in the code. The first I alluded to in section 5.1.3. This is that the read CSV file sets incorrect data for the final value. This results in a crash when trying to pass the data to the MG constructor. This error can be avoided by removing the last individual from the dataset. as the population normally consists of at least 100 individuals, this compromise is unlikely to have a major effect of the results. The second known error is when using the two-player backgammon game. In the bear off stage, a player can bear off any number of counters lower than the dice rolls. To follow the rules of backgammon, the player should be able to bear off counters that require a lower number than what they rolled. However, the dice cannot properly update the values, as it finds the values which need changing by using the exact distance moved. The heuristic and genetic algorithms use a different method to move counters, and so this error does not occur there. As it is only present in the human player mode, and does not affect the learning and it should be considered a minor error. The human players should take responsibility for not exploiting this and cheating. The third is also in the two-player backgammon game, and doesn't appear in the greedy or genetic algorithm. This error prevents a player from re-entering from the bar if multiple counters are currently occupying it. This seems to be avoided if the player selects the first counter, rerolls twice and then will be allowed to re-enter the counter.

5.2. Evaluation of Data:

In this section I will be evaluating and analysing the data. Fist I will be evaluating the best selection method and the best mutation rate. This was done by using a 100-population size, and 500 generations. I ran two tests at a time both the same mutation rate, but one used tournament selection, the other used roulette wheel selection. Here you can see that the tournament selection outperformed the roulette wheel selection in all the tests. This could be because the roulette wheel still has a chance of selecting individuals which lost. The mutation rate of 0.025 caused a huge drop in performance of the roulette wheel, to the point it is underperforming the greedy algorithm. This will be due to the instability caused by the higher mutation rate, coupled with the algorithm selecting individuals that lost. These tests provided a clear winner for further experiments though. The tournament selection with a very low mutation rate of 0.005. This mutation rate must be enough to overcome some local minima, but not too much that the system becomes unstable. This would make it the ideal mutation rate. Interesting to note that the tournament selection of 0.025 and 0.020 have the same chance of winning per game. This could be because the tournaments selection method of eliminating any individual that loses a game is more stable.



I then experimented with the generations using the 0.005 mutation rate, and tournament selection established earlier. As you can see the generations don't seem to have any major effect on the performance. This wasn't what I was expecting. On generation 400 and 700 a decrease can be seen. These could be explained by a very good solution being removed in an early generation, due to luck of the dice rolls. It is also worth noting neither are lower than 0.55, so they still have a higher than a 50/50 chance of beating the greedy algorithm. All the other values seem to be between 0.58 and 0.61. Performance might not be increasing with generations as eliminating half the population on each generation makes it easier to reach convergence. This would reduce diversity and will halt any predictable increase in performance, however when checking the weights in the CSV, they are not completly homogenous, but do share similarities across tests. This could be seen as the best solution being reached. I then ran a test of 3000 generations.



Here are the first 10 individuals from the four populations of three different tests. A 900 generation, and a 1000 generation with a population of 100. Showing the difference in 100 generations. A 1000 generation with a population of 200, showing the difference when a population is increased by 100. And a 3000-generation test showing how extreme training times effect chance to win. As you can see that many of the values across populations are similar but not identical. This could mean that the best solution has not been found yet, but is located somewhere near these values. However, as these similarities are showing, when changing both population size and from drastically increasing the number of generations, it could be argued that these represent some of the best values as these minima are very difficult to get out of. This is also shown with them all having a similar performance when compared with the greedy algorithm benchmark. As these represent weights, the best solutions may have the same/similar performance output, but with a different population.

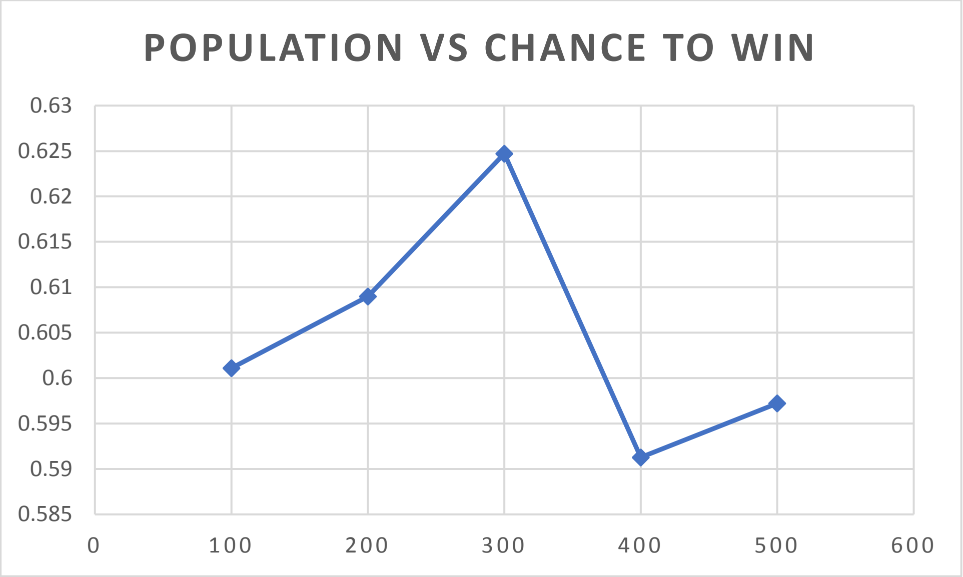
Generation 900 - Population 100				Chance To Win		0.59040404			
0.864681	0.356659	0.278399	0.185854	0.8411	0.57214	0.93297	0.760988	0.629209	0.350102
0.864681	0.356659	0.278399	0.185854	0.8411	0.364823	0.93297	0.760988	0.629209	0.350102
0.864681	0.356659	0.278399	0.185854	0.8411	0.364823	0.93297	0.760988	0.620004	0.350102
0.864681	0.356659	0.278399	0.185854	0.798807	0.57214	0.93297	0.760988	0.629209	0.350102
0.98323	0.356659	0.278399	0.185854	0.8411	0.57214	0.93297	0.760988	0.629209	0.350102
0.98323	0.356659	0.278399	0.185854	0.8411	0.364823	0.93297	0.760988	0.629209	0.350102
0.864681	0.356659	0.278399	0.185854	0.904279	0.364823	0.93297	0.760988	0.629209	0.350102
0.98323	0.356659	0.278399	0.185854	0.8411	0.57214	0.93297	0.760988	0.629209	0.350102
0.98323	0.356659	0.278399	0.185854	0.8411	0.364823	0.93297	0.760988	0.629209	0.350102
0.864681	0.356659	0.278399	0.185854	0.8411	0.57214	0.93297	0.760988	0.527685	0.350102

Generation 1000 - Population 100				Chance To Win		0.583232323			
0.89316	0.386543	0.906767	0.010237	0.946642	0.455129	0.59283	0.088334	0.868944	0.052306
0.89316	0.386543	0.72643	0.010237	0.946642	0.455129	0.416736	0.966432	0.868944	0.052306
0.89316	0.386543	0.72643	0.010237	0.946642	0.455129	0.59283	0.088334	0.868944	0.052306
0.89316	0.386543	0.72643	0.010237	0.534944	0.455129	0.59283	0.088334	0.369888	0.052306
0.89316	0.386543	0.72643	0.010237	0.946642	0.455129	0.59283	0.088334	0.868944	0.052306
0.89316	0.386543	0.72643	0.010237	0.946642	0.455129	0.012522	0.966432	0.868944	0.052306
0.89316	0.386543	0.72643	0.010237	0.946642	0.455129	0.59283	0.088334	0.369888	0.052306
0.89316	0.386543	0.72643	0.010237	0.946642	0.291609	0.416736	0.088334	0.369888	0.052306
0.89316	0.386543	0.72643	0.010237	0.946642	0.455129	0.59283	0.088334	0.868944	0.052306
0.89316	0.386543	0.72643	0.010237	0.946642	0.455129	0.416736	0.966432	0.868944	0.052306

Generation 3000 - Population 100				Chance To Win		0.599191919			
0.580801	0.205306	0.447091	0.037177	0.517981	0.859666	0.395398	0.413996	0.161705	0.15239
0.674226	0.205306	0.447091	0.037177	0.517981	0.705752	0.395398	0.413996	0.232333	0.15239
0.580801	0.205306	0.447091	0.037177	0.517981	0.859666	0.395398	0.413996	0.232333	0.15239
0.674226	0.190857	0.447091	0.037177	0.517981	0.738192	0.395398	0.413996	0.232333	0.15239
0.674226	0.205306	0.447091	0.037177	0.517981	0.859666	0.395398	0.413996	0.232333	0.15239
0.674226	0.205306	0.447091	0.037177	0.517981	0.859666	0.395398	0.413996	0.232333	0.15239
0.674226	0.205306	0.447091	0.037177	0.517981	0.859666	0.395398	0.413996	0.232333	0.15239
0.674226	0.205306	0.447091	0.037177	0.517981	0.859666	0.395398	0.413996	0.232333	0.15239
0.674226	0.205306	0.447091	0.037177	0.517981	0.738192	0.395398	0.413996	0.232333	0.15239
0.674226	0.190857	0.447091	0.037177	0.517981	0.738192	0.395398	0.413996	0.161705	0.15239

Generation 1000 - Population 200				Chance To Win		0.593065327			
0.861046	0.334425	0.216049	0.051854	0.980354	0.695835	0.817305	0.873916	0.122309	0.134973
0.861046	0.334425	0.884159	0.051854	0.684716	0.459691	0.817305	0.873916	0.683489	0.134973
0.861046	0.334425	0.884159	0.051854	0.980354	0.39275	0.817305	0.873916	0.683489	0.134973
0.861046	0.334425	0.884159	0.051854	0.684716	0.459691	0.817305	0.873916	0.122309	0.362438
0.861046	0.334425	0.884159	0.051854	0.980354	0.459691	0.817305	0.22629	0.122309	0.134973
0.861046	0.334425	0.712235	0.051854	0.980354	0.695835	0.817305	0.873916	0.122309	0.134973
0.861046	0.334425	0.883159	0.051854	0.684716	0.39275	0.817305	0.873916	0.683489	0.134973
0.861046	0.334425	0.884159	0.051854	0.980354	0.412235	0.817305	0.873916	0.43121	0.134973
0.861046	0.334425	0.884159	0.051854	0.980354	0.459691	0.817305	0.22629	0.122309	0.134973
0.861046	0.334425	0.884159	0.051854	0.980354	0.459691	0.817305	0.873916	0.672636	0.362438

I then started to experiment with the population size to see what effect this will have on trying to overcome this minimum. I did these experiments with 200 generations, this is because this had the best performance in previous tests. Also, the lower generations will be beneficial when increasing the population size to reduce training time. Here you can see that a new best performance is found. It is using a population size of 300. It then decreases after that, back to the local minima. A chance to win of 0.6247 is the best performance I could achieve using this GA. The first 25 individuals from this population will be available in the appendix Fig.4 section 5.



6. Conclusion:

In conclusion, I set out with the aims of developing a professional piece of software which is capable of playing backgammon. I also set out to gain a better understanding of artificial intelligence techniques such as neural networks and genetic algorithms. I believe I have achieved this. Not only can the AI play backgammon, but it can be clearly demonstrated to outperform a greedy algorithm approach. This, as well as my literature review have shown that I have achieved my aim of gaining a better understanding of AI techniques. When training the genetic algorithm, I have shown how mutation rate, selection method, generations, and population size can affect the performance. I did show that it was very difficult to get a performance greater than 0.6 (average chance of winning each game). I think this is partly due to the random element of backgammon. I did eventually achieve a population which got a performance of 0.647, so this could be the optimum for this approach. If further experimentation was to be done, I would change three things.

1) I would rescale the attribute the MovementGenerator performed the calculations on. I think this algorithm was limited by how much these attributes vary. (some are scaled in 100s, some are a binary 1 or 0).
2) I would use a fitness based off both the distance to the end and the number of turns it took to win. I used the distance to the end for the roulette wheel and it didn't perform as well as simply selecting the winner. However, an algorithm which also made a note of how many turns it took to win would give different fitness values to the winners. This would also mean that you could use a tournament selection that doesn't just pick the winner of each game. This could help prevent early convergence and could provide a better result
3) (Azaria And Sipper, 2005) and (Felice And Ciolfi Felice, 2012) both describe a separate evaluation being used for the bear off stage. In this stage parameters have different importance. This could be done using a longer array of weights, with the first 10 being for the normal game and the rest being used for the bear off stage. It could also be achieved with each individual having two genomes. Then fitness could be calculated based on who reached the bear off stage first, and who won the game.

7. References:

Shiffman, D. (2012) The Nature of Code p390-395 p 444-452

Michell, M. (1999) An Introduction to Genetic Algorithms, fifth edition

Michell, M. (1997) L.D. Davis, Handbook of Genetic Algorithms

Russel, S. Norvig, R. (2010) Artificial Intelligence A Modern Approach, third edition

Tesauro, G. (1994) TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play

Tesauro, G. (2002) Programming backgammon using self-teaching neural nets

Pollack, J. Blair, A (1998) Co-Evolution in the Successful Learning of Backgammon Strategy

Azaria, Y. Sipper, M. (2005) GP-Gammon: Genetically Programming Backgammon Players

Felice, L. Ciolfi Felice, M. (2012) Teaching Algorithms: Backgammon Resolution Using MiniMax Strategy

Chambers, L (2001) The Practical Handbook of Genetic Algorithms Applications, second edition

Jacoby, O. Crawford, J.R. (1970) The History of Backgammon

Hauk, T. Buro, M. Schaeffer, J (2004) Minimax Performance in Backgammon

8. Appendix:

Section 3:

Fig.1

```
if mouse clicked on a player controlled point which is unavailable{
    if selectedCounter != 500 // 500 used for new unassigned values{
        for each diceValue{
            if selectedCounter == teamOne{
                for each point{
                    if point ID == originalPoint + diceValue && point != controlled by more than 1 opponent checker{
                        set point to Available
                    }
                }
            }esle{
                for each point{
                    if point ID == originalPoint - diceValue && point != controlled by more than 1 opponent checker{
                        set point to Available
                    }
                }
            }
        }
    }else{
        selectedCounter = point.pop()
    }
}else{
    if mouse clicked on an available point{
        point.push(selectedCounter)
        selectedCounter = new counter(500)// create new counter with ID of 500
    }
}
```

Fig.2

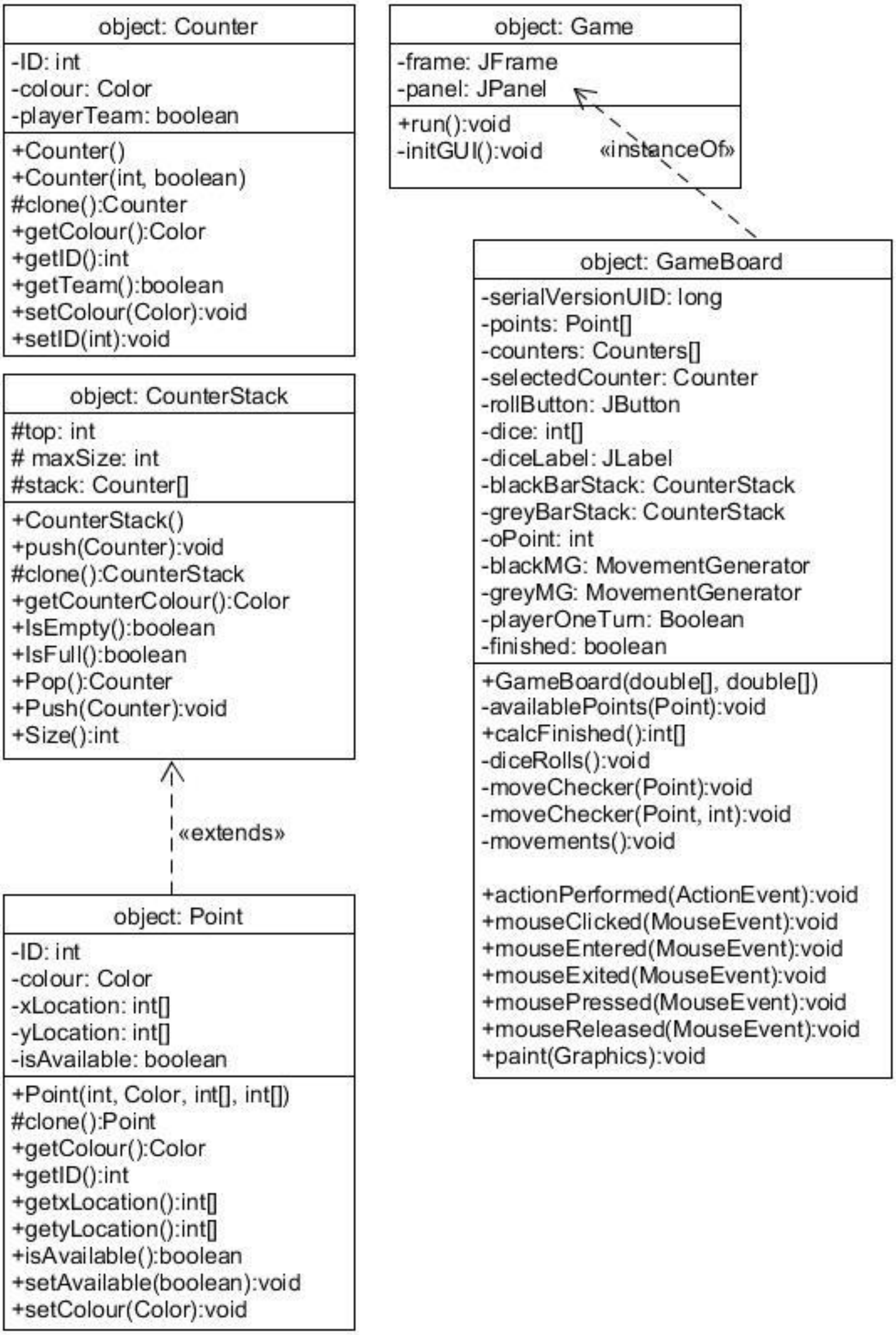

object: Counter
-ID: int
-colour: Color
-playerTeam: boolean
+Counter()
+Counter(int, boolean)
#clone():Counter
+getColour():Color
+getID():int
+getTeam():boolean
+setColour(Color):void
+setID(int):void
object: Game
-frame: JFrame
-panel: JPanel
+run():void
-initGUI():void
«instanceOf»
object: CounterStack
#top: int
# maxSize: int
#stack: Counter[]
+CounterStack()
+push(Counter):void
#clone():CounterStack
+getCounterColour():Color
+IsEmpty():boolean
+IsFull():boolean
+Pop():Counter
+Push(Counter):void
+Size():int
«extends»
object: GameBoard
-serialVersionUID: long
-points: Point[]
-counters: Counters[]
-selectedCounter: Counter
-rollButton: JButton
-dice: int[]
-diceLabel: JLabel
-blackBarStack: CounterStack
-greyBarStack: CounterStack
-oPoint: int
-blackMG: MovementGenerator
-greyMG: MovementGenerator
-playerOneTurn: Boolean
-finished: boolean
+GameBoard(double[], double[])
-availablePoints(Point):void
+calcFinished():int[]
-diceRolls():void
-moveChecker(Point):void
-moveChecker(Point, int):void
-movements():void
+actionPerformed(ActionEvent):void
+mouseClicked(MouseEvent):void
+mouseEntered(MouseEvent):void
+mouseExited(MouseEvent):void
+mousePressed(MouseEvent):void
+mouseReleased(MouseEvent):void
+paint(Graphics):void
object: Point
-ID: int
-colour: Color
-xLocation: int[]
-yLocation: int[]
-isAvailable: boolean
+Point(int, Color, int[], int[])
#clone():Point
+getColour():Color
+getID():int
+getxLocation():int[]
+getyLocation():int[]
+isAvailable():boolean
+setAvailable(boolean):void
+setColour(Color):void

Fig.3

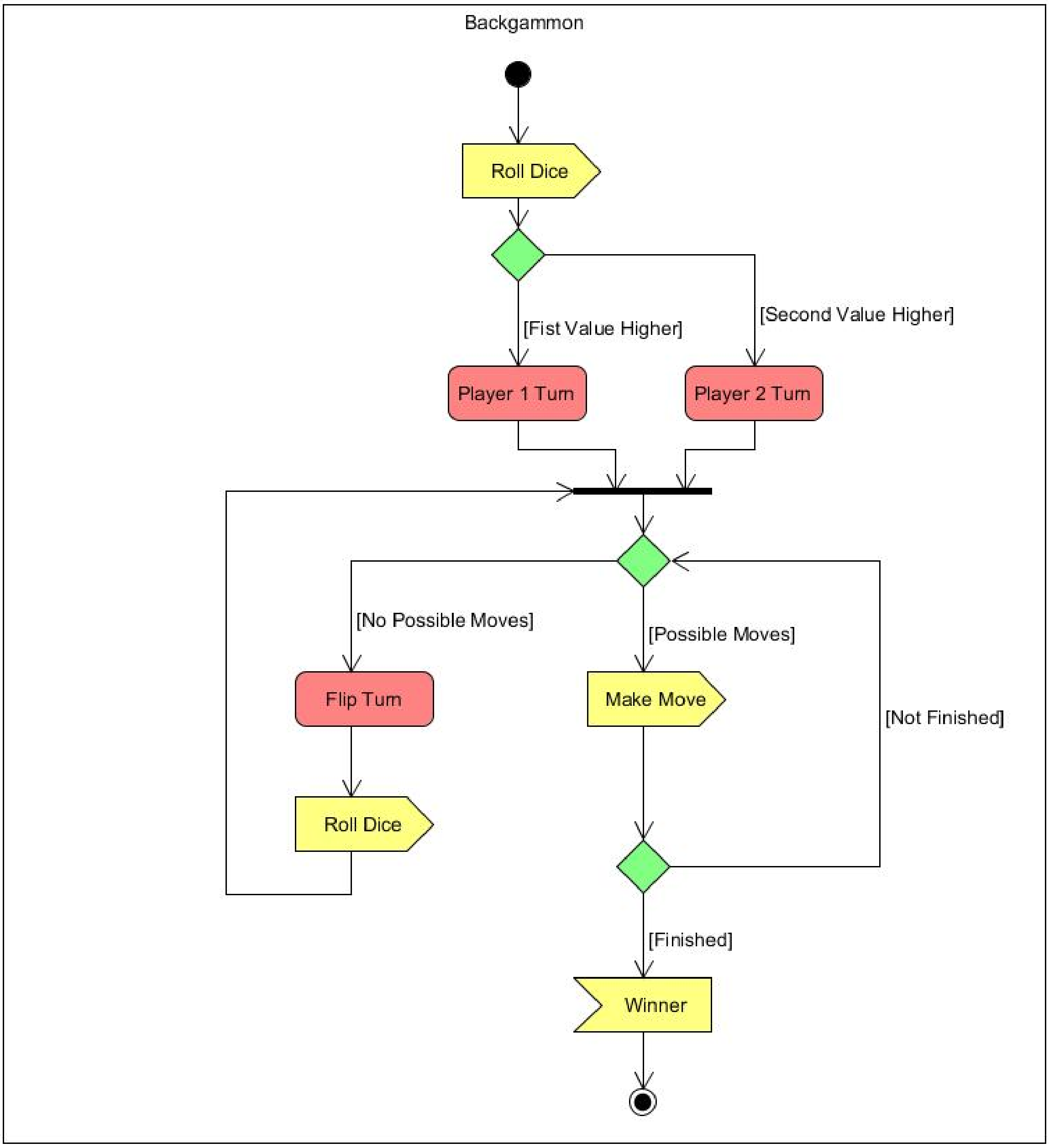

Backgammon
Roll Dice
[Fist Value Higher]
[Second Value Higher]
Player 1 Turn
Player 2 Turn
[No Possible Moves]
[Possible Moves]
Flip Turn
Make Move
[Not Finished]
Roll Dice
[Finished]
Winner

Fig.4

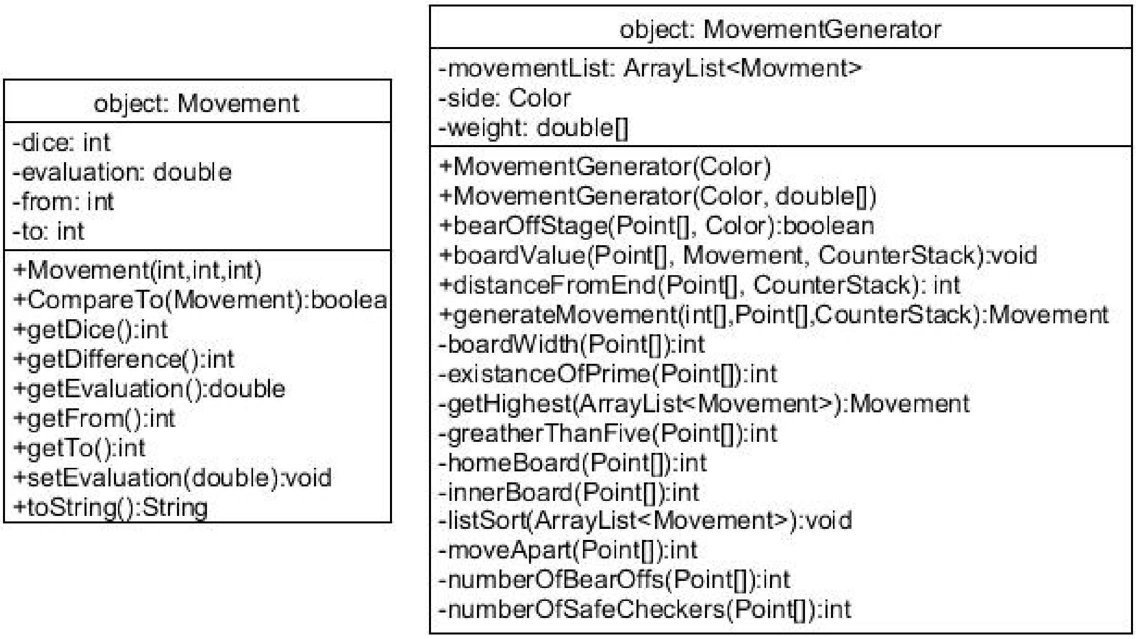
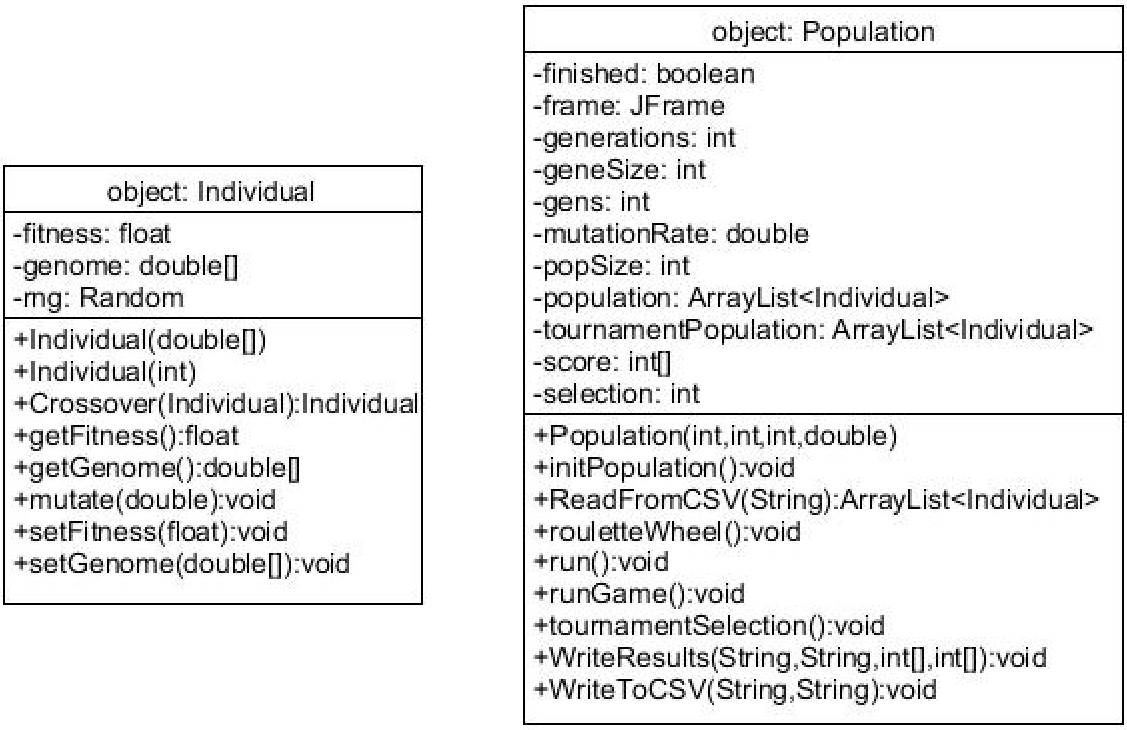
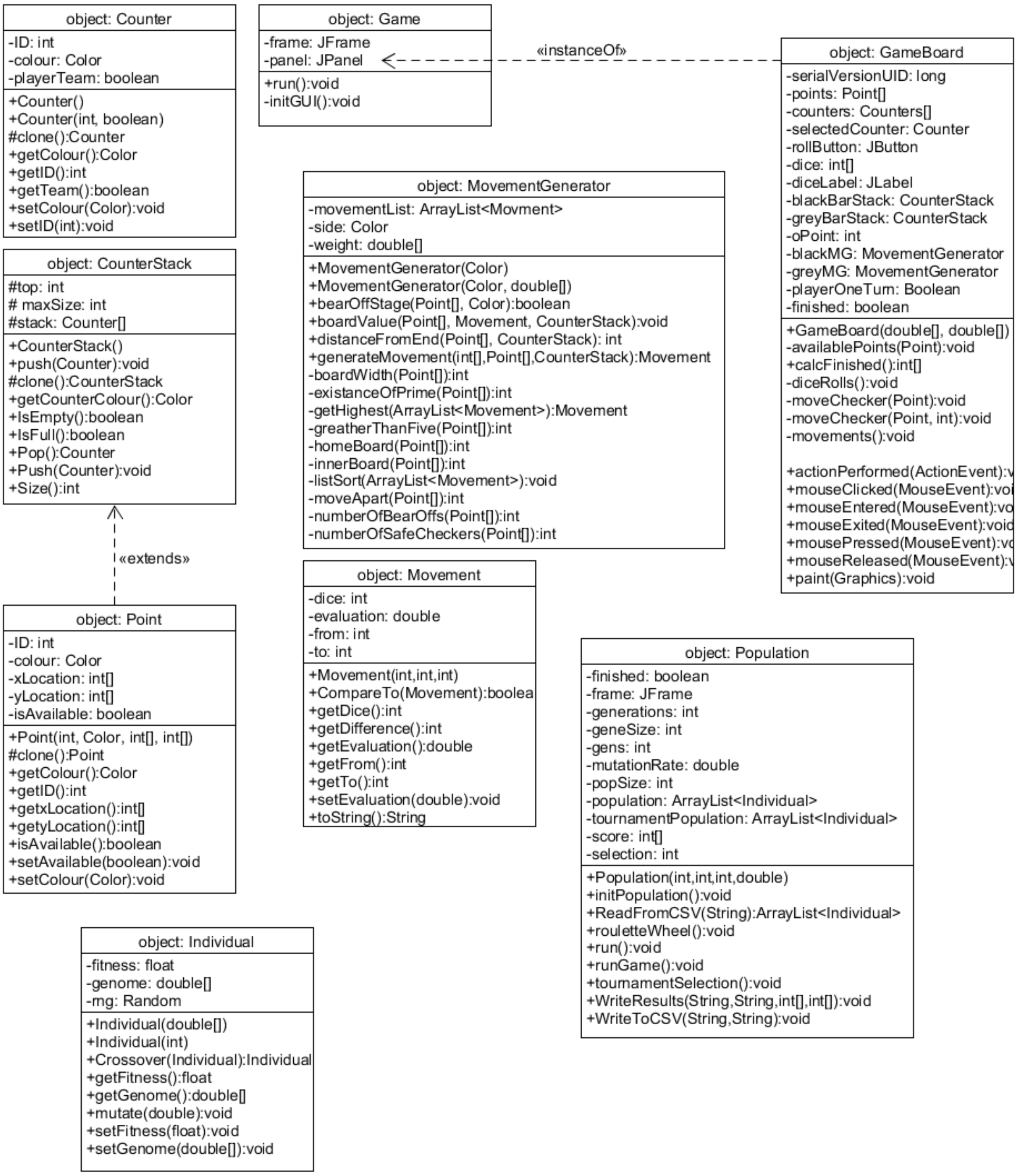

object: Movement
-dice: int
-evaluation: double
-from: int
-to: int
+Movement(int,int,int)
+CompareTo(Movement):boolea
+getDice():int
+getDifference():int
+getEvaluation():double
+getFrom():int
+getTo():int
+setEvaluation(double):void
+toString():String
object: MovementGenerator
-movementList: ArrayList<Movment>
-side: Color
-weight: double[]
+MovementGenerator(Color)
+MovementGenerator(Color, double[])
+bearOffStage(Point[], Color):boolean
+boardValue(Point[], Movement, CounterStack):void
+distanceFromEnd(Point[], CounterStack): int
+generateMovement(int[],Point[],CounterStack):Movement
-boardWidth(Point[]):int
-existanceOfPrime(Point[]):int
-getHighest(ArrayList<Movement>):Movement
-greatherThanFive(Point[]):int
-homeBoard(Point[]):int
-innerBoard(Point[]):int
-listSort(ArrayList<Movement>):void
-moveApart(Point[]):int
-numberOfBearOffs(Point[]):int
-numberOfSafeCheckers(Point[]):int

Fig.5


object: Individual
-fitness: float
-genome: double[]
-rng: Random
+Individual(double[])
+Individual(int)
+Crossover(Individual):Individual
+getFitness():float
+getGenome():double[]
+mutate(double):void
+setFitness(float):void
+setGenome(double[]):void
object: Population
-finished: boolean
-frame: JFrame
-generations: int
-geneSize: int
-gens: int
-mutationRate: double
-popSize: int
-population: ArrayList<Individual>
-tournamentPopulation: ArrayList<Individual>
-score: int[]
-selection: int
+Population(int,int,int,double)
+initPopulation():void
+ReadFromCSV(String):ArrayList<Individual>
+rouletteWheel():void
+run():void
+runGame():void
+tournamentSelection():void
+WriteResults(String,String,int[],int[]):void
+WriteToCSV(String,String):void

Section 4:

Fig.1



Section 5:

Fig.1

```
1 4 0 0 Player One Turn:false
From: 24To: 20
From: 24To: 23
3 2 0 0 Player One Turn:true
From: 1To: 4
From: 1To: 3
4 2 0 0 Player One Turn:false
From: 23To: 21
From: 20To: 16
5 3 0 0 Player One Turn:true
From: 4To: 9
From: 3To: 3
From: 9To: 12
6 3 0 0 Player One Turn:false
From: 21To: 18
From: 18To: 18
From: 13To: 7
1 2 0 0 Player One Turn:true
From: 3To: 5
From: 17To: 18
3 3 3 3 Player One Turn:false
From: 25To: 22
From: 16To: 13
From: 7To: 4
From: 6To: 3
2 2 2 2 Player One Turn:true
From: 5To: 7
From: 7To: 9
From: 9To: 11
```

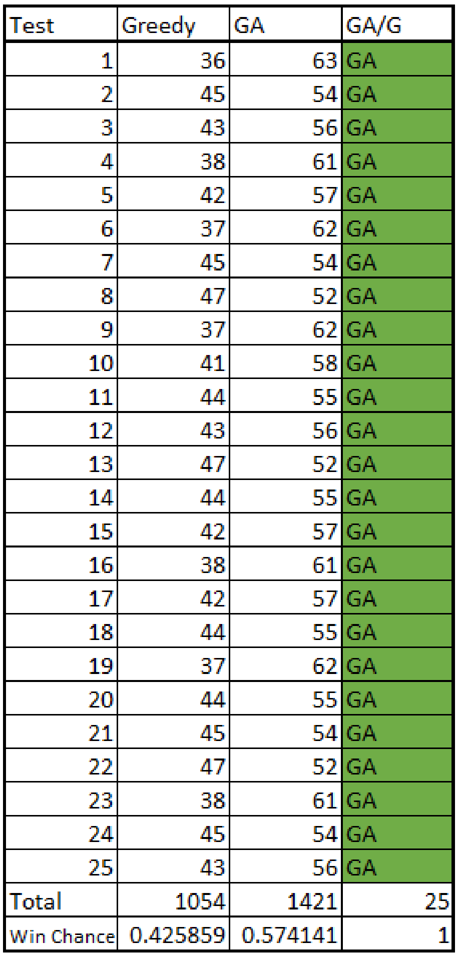
```
From: 12To: 14
```

Fig.2

```
5 2 0 0 Player One Turn:true

(from=1, to=3)

bear offs 0, safe checkers 13, distance from end 165, prime 0, >5 0, inner count 5, apart count 0, home board 2, distance 18

-167.0

(from=12, to=17)

bear offs 0, safe checkers 15, distance from end 162, prime 0, >5 0, inner count 5, apart count 0, home board 2, distance 18

-162.0

(from=12, to=14)

bear offs 0, safe checkers 14, distance from end 165, prime 0, >5 0, inner count 5, apart count 0, home board 2, distance 18

-166.0

(from=17, to=22)

bear offs 0, safe checkers 14, distance from end 162, prime 0, >5 0, inner count 6, apart count 0, home board 2, distance 21

-165.0

(from=17, to=19)

bear offs 0, safe checkers 15, distance from end 165, prime 0, >5 1, inner count 6, apart count 0, home board 2, distance 18

-165.0

(from=19, to=21)

bear offs 0, safe checkers 14, distance from end 165, prime 0, >5 0, inner count 5, apart count 0, home board 2, distance 20

-168.0

>>> (from=12, to=17)

//////////

(from=1, to=3)

bear offs 0, safe checkers 13, distance from end 160, prime 0, >5 0, inner count 5, apart count 0, home board 2, distance 18

-162.0
```

```
(from=12, to=14)

bear offs 0, safe checkers 14, distance from end 160, prime 0, >5 0, inner count 5, apart
count 0, home board 2, distance 18

-161.0

(from=17, to=19)

bear offs 0, safe checkers 15, distance from end 160, prime 0, >5 1, inner count 6, apart
count 0, home board 2, distance 18

-160.0

(from=19, to=21)

bear offs 0, safe checkers 14, distance from end 160, prime 0, >5 0, inner count 5, apart
count 0, home board 2, distance 20

-163.0

>>> (from=17, to=19)

//////////
```

Fig.3



Test	Greedy	GA	GA/G
1	36	63	GA
2	45	54	GA
3	43	56	GA
4	38	61	GA
5	42	57	GA
6	37	62	GA
7	45	54	GA
8	47	52	GA
9	37	62	GA
10	41	58	GA
11	44	55	GA
12	43	56	GA
13	47	52	GA
14	44	55	GA
15	42	57	GA
16	38	61	GA
17	42	57	GA
18	44	55	GA
19	37	62	GA
20	44	55	GA
21	45	54	GA
22	47	52	GA
23	38	61	GA
24	45	54	GA
25	43	56	GA
Total	1054	1421	25
Win Chance	0.425859	0.574141	1

Fig.4

End Population

0.686641	0.200166	0.936867	0.012229	0.905916	0.119041	0.302274	0.806642	0.017098	0.003349
0.686641	0.200166	0.747509	0.012229	0.905916	0.119041	0.302274	0.594862	0.017098	0.051283
0.686641	0.200166	0.936867	0.012229	0.98706	0.119041	0.488279	0.594862	0.017098	0.003349
0.812556	0.200166	0.747509	0.012229	0.966633	0.119041	0.525318	0.806642	0.017098	0.051283
0.686641	0.200166	0.04496	0.012229	0.928153	0.119041	0.525318	0.806642	0.017098	0.051283
0.686641	0.200166	0.747509	0.012229	0.966633	0.119041	0.302274	0.594862	0.233369	0.003349
0.988967	0.200166	0.013908	0.012229	0.966633	0.119041	0.488279	0.594862	0.233369	0.003349
0.686641	0.200166	0.747509	0.012229	0.905916	0.119041	0.360607	0.806642	0.017098	0.051283
0.812556	0.200166	0.013908	0.268554	0.941441	0.119041	0.525318	0.594862	0.017098	0.003349
0.812556	0.200166	0.659406	0.012229	0.966633	0.119041	0.525318	0.806642	0.137569	0.052343
0.988967	0.200166	0.747509	0.012229	0.966633	0.119041	0.302274	0.594862	0.017098	0.051283
0.988967	0.200166	0.747509	0.012229	0.928153	0.119041	0.302274	0.028365	0.017098	0.051283
0.988967	0.200166	0.747509	0.012229	0.966633	0.119041	0.488279	0.806642	0.017098	0.003349
0.645439	0.200166	0.747509	0.012229	0.941441	0.119041	0.360607	0.806642	0.303413	0.051283
0.812556	0.200166	0.747509	0.012229	0.966633	0.119041	0.525318	0.806642	0.137569	0.052343
0.645439	0.200166	0.747509	0.012229	0.966633	0.119041	0.488279	0.864249	0.017098	0.052343
0.988967	0.200166	0.747509	0.012229	0.905916	0.119041	0.360607	0.806642	0.017098	0.051283
0.645439	0.200166	0.747509	0.012229	0.966633	0.872805	0.302274	0.594862	0.017098	0.003349
0.988967	0.200166	0.747509	0.012229	0.905916	0.119041	0.488279	0.806642	0.017098	0.051283
0.686641	0.200166	0.936867	0.012229	0.98706	0.119041	0.488279	0.028365	0.017098	0.003349
0.988967	0.200166	0.747509	0.012229	0.905916	0.119041	0.302274	0.806642	0.137569	0.003349
0.686641	0.200166	0.747509	0.012229	0.966633	0.119041	0.488279	0.89048	0.017098	0.051283
0.686641	0.200166	0.747509	0.012229	0.966633	0.119041	0.302274	0.594862	0.303413	0.051283
0.686641	0.200166	0.747509	0.012229	0.966633	0.119041	0.525318	0.806642	0.017098	0.003349

Link to project file:

https://stummuac-my.sharepoint.com/:u:/g/personal/16051252_stu_mmu_ac_uk/ETsDNXX0gdtHtbYjgJyG2u0B7OC5_NSLAag705voLXtIYg