Bachelor Degree Project

UNIVERSITY OF SKÖVDE

EVOLUTIONARY AI IN BOARD GAMES

An evaluation of the performance of an evolutionary algorithm in two perfect information board games with low branching factor

Bachelor Degree Project in Informatics

G2E, 22.5 credits, ECTS
Spring term 2015

Viktor Öberg

Supervisor: Joe Steinhauer
Examiner: Anders Dahlbom

Abstract

It is well known that the branching factor of a computer based board game has an effect on how long a searching AI algorithm takes to search through the game tree of the game. Something that is not as known is that the branching factor may have an additional effect for certain types of AI algorithms.

The aim of this work is to evaluate if the win rate of an evolutionary AI algorithm is affected by the branching factor of the board game it is applied to. To do that, an experiment is performed where an evolutionary algorithm known as “Genetic Minimax” is evaluated for the two low branching factor board games Othello and Gomoku (Gomoku is also known as 5 in a row). The performance here is defined as how many times the algorithm manages to win against another algorithm.

The results from this experiment showed both some promising data, and some data which could not be as easily interpreted. For the game Othello the hypothesis about this particular evolutionary algorithm appears to be valid, while for the game Gomoku the results were somewhat inconclusive. For the game Othello the performance of the *genetic minimax* algorithm was comparable to the *alpha-beta* algorithm it played against up to and including depth 4 in the game tree. After that however, the performance started to decline more and more the deeper the algorithms searched. The branching factor of the game may be an indirect cause of this behaviour, due to the fact that as the depth increases, the search space increases proportionally to the branching factor. This increase in the search space due to the increased depth, in combination with the settings used by the *genetic minimax* algorithm, may have been the cause of the performance decline after that point.

Keywords: Branching factor, game tree, artificial intelligence, ai, evolutionary algorithm, minimax, alpha-beta

Contents

1 Introduction

The branching factor of a tree structure is well known to have an effect on the time an algorithm takes to search through the tree (Levinovitz, 2014). However, to the author's knowledge, the effect the branching factor has on the win ratio of evolutionary AI algorithms in board games does not appear to have been studied extensively.

However, two recent studies indicate that the branching factor of a tree structure in a perfect information board game might affect certain AI algorithms more than what it's generally known to. Baier and Winands (2014) found that if the game Breakthrough was modified so that the branching factor was reduced, an AI algorithm based on "Monte Carlo Tree Search", combined with Minimax was found to increase its win frequency by a large amount.

In yet another recent study Johnsson (2014), an evolutionary AI algorithm referred to as a "Genetic Minimax", previously proposed by Hong, Huang and Lin (2001), was evaluated for the perfect information board game Pentago. In her results Johnsson (2014) came to the conclusion that the poor performance in regard as to how many times the algorithm managed to win against an alpha-beta algorithm was potentially caused by the high branching factor of the game.

Primarily based on the study by Baier and Winands (2014) there is reason to believe that certain AI algorithms are affected by high branching factor in a negative way. At least when they are used in perfect information board games. If the branching factor is too high, it could cause an algorithm to select a good move over a bad move. That is, if this choice is somehow affected by some randomness built into the algorithm. An evolutionary algorithm such as *genetic minimax* is a prime example of this, that an element of randomness affects the final choice of move. If the branching factor is lower, then there are fewer choices to decide between, which may in turn cause a higher chance of selecting a good move over a bad one.

In both the study by Baier and Winands (2014) and the study by Johnsson (2014), the algorithms they evaluated had some element of randomness built in. This combined with the results obtained by Johnsson (2014), in her study about the branching factor being the potential cause of the poor performance of the evaluated *genetic minimax* algorithm, merits a further look at this topic. This has been boiled down to the following hypothesis:

> *The branching factor of the tree structure in a perfect information board game has an effect on the performance of an evolutionary algorithm.*

To investigate this hypothesis the evolutionary *genetic minimax* algorithm, previously evaluated by Johnsson (2014), is evaluated once again. However, this time it is evaluated for two low branching factor board games. The two games chosen for this were Othello and Gomoku.

The resulting data from the experiment showed some promising results, at least in one case. For the game Othello the results indicate that the hypothesis may be valid. It may be valid

because the performance of the *genetic minimax* algorithm started to drop considerably after searching deeper than 4 in the search tree, after being equal in performance to the opponent up to that point. For the game Gomoku the results were not as clear. At depth one the *genetic minimax* managed to win a few times against the opponent, but not close to an equal amount. After that the performance dropped considerably and it failed to win a single game at any depth above 1.

The structure of this thesis is as follows, divided in to seven chapters not including the introduction. The second chapter is intended to give a background to what this work was derived from and a brief introduction to all of the relevant terms used in the thesis. Following this is the Method chapter. Here is a brief explanation on how the problem is solved. The chapter following briefly describes the implementation details regarding the games and algorithms used. This includes, for example, specific settings used for the evolutionary algorithm and the evaluation functions for the games. The final chapters present the experimental results, the analysis of the results and the conclusions drawn from the analysis. After that there is a brief discussion and suggestions for future work.

2 Background

2.1 Related research

To the author's knowledge, few to no studies appear to have been conducted regarding how the branching factor of a game tree affects the win ratio of an AI algorithm applied to it. More specifically, there does not appear to exist any studies where evolutionary AI algorithms were addressed in this regard, and if there is any connection between the branching factor of the game tree and how the evolutionary algorithm performed regarding the number of wins.

However, there is at least one occurrence of where the effect of the branching factor on performance was examined, and how it affected the win ratio of another type of AI algorithm. Baier and Winands (2014) proposed a hybrid algorithm which consisted of a mix between "Monte Carlo Tree Search" and *minimax*. This hybrid was run in the game Breakthrough. When they varied the branching factor of the game by changing the board size, they noticed a significant change in how often the algorithm managed to win. When they went from a board size of 6x6 with an average branching factor of 15,5 to a board size of 18x6 with an average branching factor of 54,2, one variant of the algorithm reduced its win rate from 30,8% to 10,2%. The same algorithm searching deeper reduced its win rate even further from 20,2% to 0,1% while varying between the two above mentioned board sizes. Other variants of the algorithm worked better and did not appear to be strongly affected by varying the branching factor, although not as significantly as the first mentioned variant, but they were affected somewhat nonetheless.

In a recent study performed by Johnsson (2014) an evolutionary AI algorithm referred to as a "Genetic Minimax", previously proposed by Hong, Huang and Lin (2001), was evaluated for the board game Pentago. Part of the reason for evaluating this algorithm was that it showed some initial promise based on the preliminary tests performed in the original proposal, but had since then not been applied in any extent to existing problems. In Johnsson's (2014) study there was some concern as to why there had been little to no research on the algorithm beyond the original proposal. However, since it did show some promise she evaluated it against the *minimax* algorithm with the *alpha-beta* optimisation in the board game Pentago.

In her experiment Johnsson (2014) gathered a large amount of data regarding how the *genetic minimax* algorithm performed under different conditions. One example of the data gathered was information on how fast the algorithms could make a move when searching down to a given depth, where the depth was increased up to 5 from a starting depth of 1. For this her results showed that the *genetic minimax* had an increase in execution time approximately linear to the depth searched. This with an average of about a 26% increase in execution time when the depth increased. This performance was far superior to the averaging 90% increase of the baseline *alpha-beta* algorithm she compared with.

In addition to the above, statistics on how well the *genetic minimax* algorithm actually performed in playing the game was also gathered by Johnsson (2014). However, as seen in

Table 1 below it was relatively one sided with the *alpha-beta* algorithm having a clear advantage.

Depth	AlphaBetaPlayer	GeneticMinimaxPlayer	Tied
1	100	100	0
2	188	0	12
3	196	4	0
4	200	0	0
5	195	5	0

Table 1 - Number of wins and losses for each algorithm at a given depth, as tested by Johnsson (2014).

Johnsson's (2014) conclusions were, in short, that the *genetic minimax* algorithm was not a suitable choice when applied to the board game Pentago. At least compared to the regular *alpha-beta* algorithm, which is evident in Table 1 above. In her analysis she concluded that a possible cause of this poor performance was the high branching factor of the game Pentago.

In her conclusions Johnsson (2014) suggested two alternative games where the *genetic minimax* algorithm might be better suited. One of the alternatives suggested was the board game Gomoku due to it being less "dynamic" than Pentago, and the second was Abalone since it had a lower branching factor. Gomoku also has a lower branching factor than Pentago, but this is not mentioned by her.

2.2 Perfect information board games

Games where everything about the state of the game is known by every player involved, such as Chess, Gomoku, and Othello, are known as *perfect information games*. In these types of games the players know what the result of every move will be, and subsequently what options are available for the next move. All of this is known from the start of the game until its very end. However, the *perfect information* part only involves knowing about which potential moves the opponent can make and what effects they will have, not the exact move the opponent actually decides upon. The opposite of a *perfect information* game is an *imperfect information* game. An example of this is the game Backgammon where there is a random element built into the game. The random element consists of a dice roll which decides the moves you are allowed to make that turn. This causes the player to not know its moves until the dice roll has been made. Since both players rely on the dice to determine which moves are available to play, the dice roll element of the game also causes the player to not know which moves the opponent has available at its next turn (Millington & Funge, 2009).

2.3 Turn based board games and Articifial Intelligence

A turn based board game is a board game where there is more than one player. A game such as this can be both *perfect information*, and *imperfect information*. The players in these games take turns in making their move on the game board. According to Millington and Funge (2009), most of the popular AI algorithms are in their basic form limited to only two player games, but can be adapted for use in games with more players than that if necessary. However, most of the optimisations for these algorithms assume that there are only two players as well, and the

optimisations are often more difficult to adapt (Millington & Funge, 2009). One well known optimisation of the *minimax* algorithm called *alpha-beta* is explained further in Section 2.5.2.

2.3.1 Board game complexity

Any perfect information turn based board game can be represented as a game tree (Millington & Funge, 2009). Each node in this tree would then represent one state of the game, and each branch coming from one of these nodes would represent a possible move from that game state to another. The complexity of one of these board games might refer to several different definitions, for example how two of these game states differ from each other after a move has been made. However, a more common way of describing the complexity of a board game is by the term *branching factor* (Allis, 1994).

The branching factor of a tree structure is, if the tree is uniform, equal to the number of children of each node. An example of this is a full binary tree. In a full binary tree the branching factor is equal to 2, since each node has exactly two children. The exception to this are the leaves which have none, as seen in Figure 1 (Black, 2014).

A tree structure stemming from a board game would generally have a structure where the nodes in the tree have a varying number of children. That is, not every move made by one player results in the same number of moves available for the next player to make. For a tree structure like this an average branching factor can be calculated instead (Allis, 1994). According to Millington and Funge (2009) the branching factor is a good indicator of how much difficulty a computer program will have in playing the game.

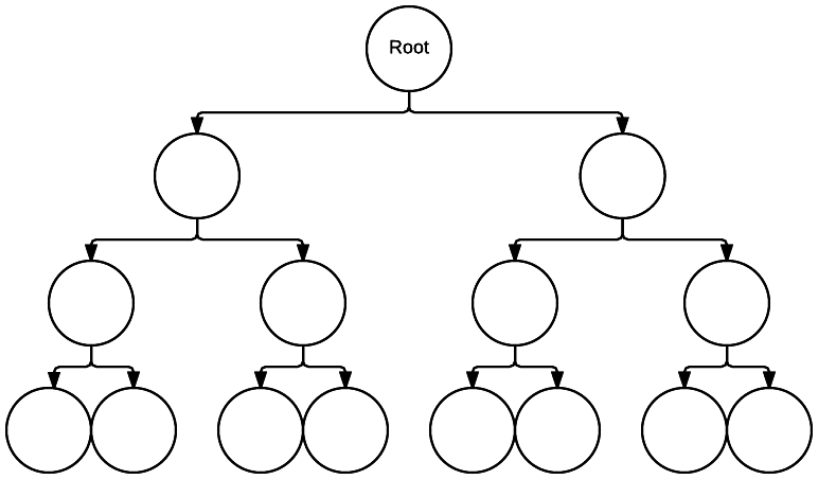

Figure 1 - A simple full binary tree structure. Apart from the leaf nodes each node has exactly two children, resulting in a branching factor of 2.

The branching factor is known to primarily have an impact on algorithms which uses brute force to search through the game tree, such as *minimax*, since the higher the branching factor the longer time it takes to look through the tree (Levinovitz, 2014). A tree with a branching factor of 10 for example has a node count of 10^n, where n is the depth of the tree. The root

has 10 children, each of which has 10 children, and so on. This does not affect the final outcome of a search algorithm such as *minimax*, but it does take longer to reach a solution compared to a lower branching factor tree.

The reason why the algorithm needs more time to search through a high branching factor tree compared to a low branching factor tree is that there are more nodes to evaluate. If a Minimax algorithm was to search through a tree with a branching factor of 2, to a depth of 10, 2^{10} = 1024 nodes would need to be evaluated. If the same algorithm had to search through a tree with a branching factor of 10 to the same depth, the number of nodes it would need to evaluate would be 10^{10} = 10 billion nodes, which would obviously take a lot more time.

2.4 The evaluated board games

Two board games were chosen for this evaluation, the games are called Othello and Gomoku and are both *perfect information*. The branching factor of either of the games chosen had to be considerably lower than that of Pentago, which was evaluated by Johnsson (2014). Gomoku fits into this constraint since it has a branching factor of approximately 1/8th that of Pentago. Othello also fits here since it has an average branching factor of 10 (Allis, 1994).

2.4.1 Gomoku

Gomoku is a two player, perfect information, board game. It is more commonly known as *five in a row*. The goal of Gomoku is to place five consecutive game pieces of your own color either horizontally, vertically or diagonally in a grid. This is made a little more difficult with an opposing player working against you with the same goal in mind, only with its own game pieces (Wang & Huang, 2014). The grid on which the game is played can be of different sizes, but should preferably be at least 5 rows and 5 columns big since the goal won't be achievable otherwise.

The reason Gomoku has a branching factor approximately 1/8th that of Pentago is that they are essentially the same game with one exception. What sets them apart is that Pentago is divided into 4 quadrants. A move in Pentago involves also rotating one of these quadrants 90 degrees either clockwise or anti-clockwise, in addition to placing a game piece.

2.4.2 Othello

Othello is also a two player, perfect information, board game. The basic gameplay of Othello is the following. At the start of the game four game pieces are already placed on the game board, which usually consists of 8 rows and 8 columns. The starting pieces are aligned as seen in Figure 2a.

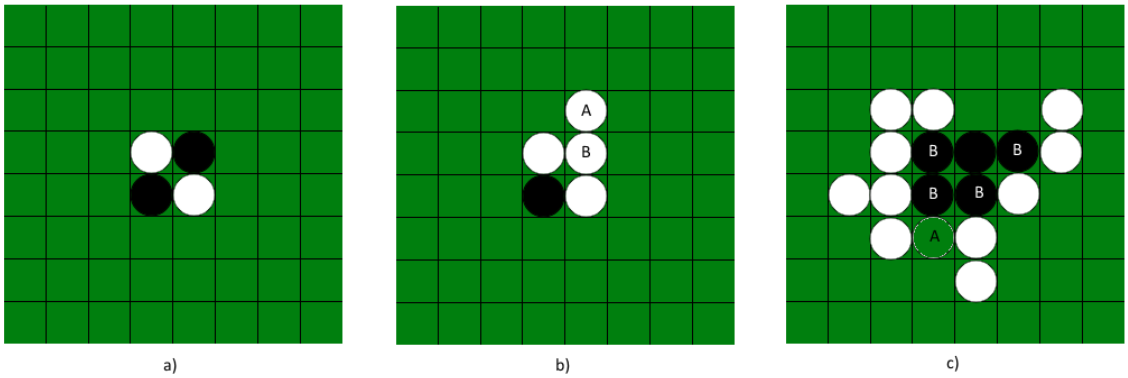

Figure 2 - Three situations in Othello. a) Initial board state. b) White made move (A) capturing the black (B) game piece. c) White can make the move at (A) capturing all the black (B) game pieces in both the vertical and diagonal directions at the same time.

A legal move in Othello is a move where the current player places a game piece which traps one or more of the opponent's game pieces between one game piece of the players color already on the board, and the one being placed, see Figure 2b. This trapping can also occur in multiple directions in one move, see Figure 2c. If a move cannot be made then the turn reverts to the opponent. When neither player can make a legal move then the game is over and the winner is the player which has the largest amount of game pieces of its own color on the board (Lazard, 1993).

2.5 Search algorithms

Search algorithms seen in the context of board games are often placed in the category *adversarial search* (Russel & Norvig, 2010). Adversarial search refers to a type of competitive environment where the goals of the players are diametrically opposed to each other. That is, if one player wins, the other player has to lose (Russel & Norvig, 2010). These types of games are often turn based, and a common example is the board game chess. Since they are turn based you need to take into account the moves of the opponent as well. While you obviously make your moves so that they benefit you the most, every other turn your opponent makes a move based on what they consider is the best move for them to make (or a move which benefits you the least). This will henceforth be referred to as maximising and minimising, where a player maximises its own gain while minimising the opponents gain for every move made.

A regular search algorithm, such as depth-first-search to a given depth (Russel & Norvig, 2010), would not work very well for this. This is because it does not take into account the fact that there are two competing goals which work against each other. An alternative, albeit naïve, use for a regular depth-first search in the context of games is for example to find the shortest path on a map from point A to point B.

However, for an adversarial board game depth-first search is not very useful. Consider the tree structure depicted in Figure 3, this depicts a depth-first search algorithm on a simple, complete, game tree.

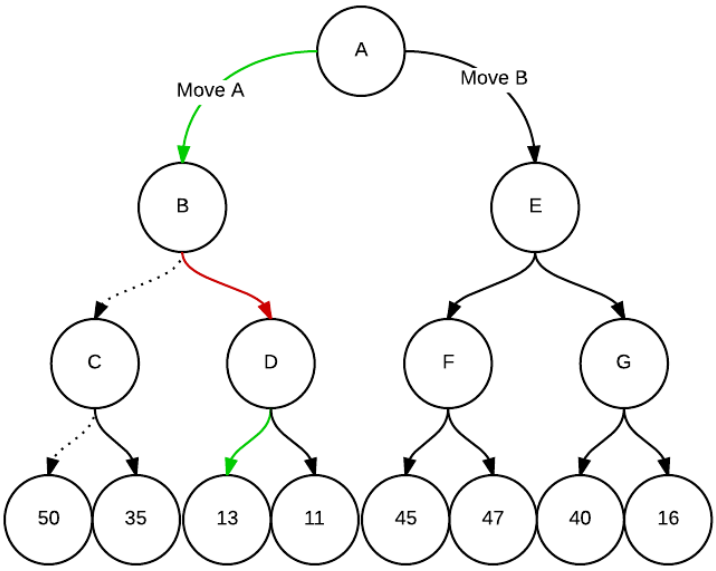

Figure 3 - Simple depth-first search on a game tree. The dotted path is the desired path seen from node A, while the actual path turns out different because of the adversarial nature of the game.

The dotted path represents what the depth first search found was the best option for the maximising player, since at the end there is a leaf with the value of 50, which is higher than all the other leaves. Therefore the depth-first search algorithm decides that *Move A* is the best move. However, as stated earlier this does not take into account that as soon as *Move A* is chosen, the minimising player has the same opportunity to search from that point on. The minimising player will then choose what it considers is in its own best interest, which happens to be the path to node D which in turn leads to the leaf with a value of 11. The leaf nodes in this example are seen from the perspective of the maximising player, so the leaf with the lowest value will be the best choice for the minimising player and vice versa. The maximising player then has the option to choose between either 11 or 13, a far cry from what the initial expected outcome from choosing path A was, and will consequently chose 13 since it is the best at that point. This is where the *minimax* algorithm comes in.

2.5.1 The minimax algorithm

Minimax is a depth-first recursive algorithm. It was originally designed for two-player zero sum games, but has since then been adapted to use in a variety of scientific fields. The algorithm works by searching down to the leaves at a previously set depth in the game tree. The algorithm then evaluates all of the leaves at that depth which belong to the same parent node, using an evaluation function. The evaluation function is meant to be an estimate of the value of a particular state of the game being played. This function simply evaluates the state of the

game at that particular leaf and based on what makes this state a good state, depending on what game it is, returns a value indicating how advantageous the state is for the evaluator (Papadopoulos, Toumpas, Chrysopoulos & Mitkas, 2012).

Depending on if the transition to the leaves from the parent is for the minimising or maximising player, the algorithm sets the value of the parent to be either the lowest estimated or the highest estimated value of its children. The algorithm proceeds with this for all branches of the tree until the best move to make is "bubbled" up to the root node. The algorithm is illustrated in Figure 4 below.

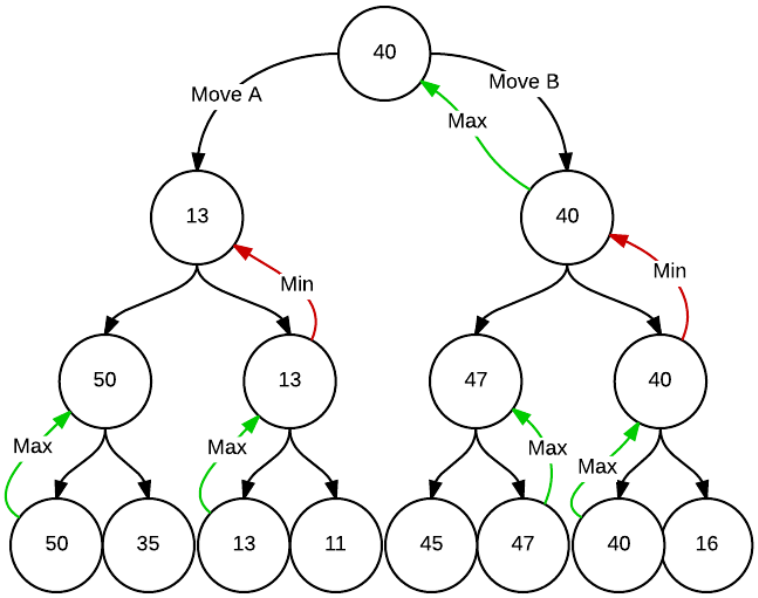

Figure 4 – The minimax algorithm illustrated in an example search tree.

As you can see in Figure 4 the *minimax* algorithm might not reach the absolute best leaf in the tree. It will however help reach the best possible one given that there is an opponent working against you trying to prevent you reaching that leaf. As one can see in Figure 4 the best possible value is actually 50, but since the opponent is actively trying to minimise your gain, the best one that can be reached given this restriction is actually 40.

This algorithm is meant to mimic a situation where there are two alternating players, and where the algorithm also assumes that both of the players make the most optimal move every single time. Most human players obviously don't do this, and if this were to happen then the *minimax* algorithm will perform even better (Russel & Norvig, 2010). In the example above in Figure 4 for example, if the minimising player makes a mistake in either of its moves then the maximising player will end up on either 47 or 50, instead of 40, depending on where the minimising player made the mistake.

There are some problems with the minimax algorithm though. The time cost of the algorithm can become quite impractical if actually applied to a real game. This is because it does perform

a complete depth-first search of the entire game tree. The number of game states the algorithm has to examine is so large for a real game that going any further down than just a few levels will slow the algorithm down considerably. A solution to this problem, which should allow the search to continue further down at the same time cost while returning the same solution, is an optimisation known as Alpha-beta-pruning (Russel & Norvig, 2010).

2.5.2 Alpha-beta

The purpose of the *alpha-beta* algorithm is to avoid evaluating any branches in the game tree that does not influence the final choice of move the algorithm makes (Russel & Norvig, 2010). The algorithm gets its name from the two parameters getting passed along as the algorithm traverses the tree. The first one is *alpha*. Alpha is the best choice which has been found so far for the maximising player. The second one is beta, which is the best choice found for the minimising player (Russel & Norvig, 2010).

Consider Figure 5, where the *alpha-beta* algorithm has been applied to an example game tree. The algorithm first travels down to node C where it's the turn of the maximiser. The maximising step returns the most promising child of C to the parent, and then returns it further up to node B. The first child of Node B was evaluated to 12, and since B will be assigned the smallest value from its children, Node B will now have an absolute maximum value of 12. The algorithm then travels down from node B to node D. Node D will eventually be assigned the maximum value of its children, and it will be at least 15 since the first child has a value of 15. As one might realise, there is no point in looking at any of the other children after this since 15 is greater than the value 12 assigned to node C, and node B will be assigned the value of the smallest of its children. Now that B has explored all of its children it returns a tentative max-value of 12 to the root node, and this value is then passed down to the second child of the root node.

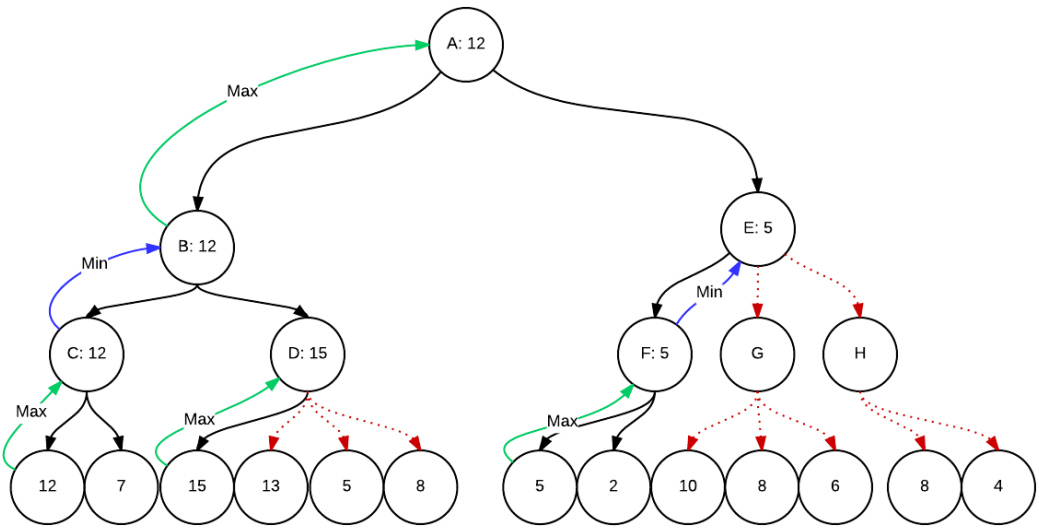

Figure 5 – A minimax search tree with alpha-beta pruning. Red dotted arrows indicate choices not even being considered because of the pruning optimisation.

The algorithm can then travel down to node F with this tentative max value for the root in tow. The algorithm then evaluates the first child of node F, and after an evaluation of its children determines it to be valued at 5. Since node E will be assigned the smallest value of its children, we can now determine that it will be valued 5 at most due to its first child being evaluated to 5. Since the root node will be assigned the largest value of either node B or node E, and knowing that node E is at most 5, the algorithm can ignore the rest of the children attached to node E. This will effectively cut of a large part of the search tree, thus saving a lot of time.

2.6 Evolutionary algorithms

Originally, evolutionary algorithms were first developed in the 1960s. Inspired by natural evolution, they were meant to handle problems which were too difficult for traditional analytical methods of problem solving. Apart from games there are a multitude of applications for these types of algorithms. Areas include, among others, machine learning, hardware design, and robotics (Floreano & Mattiussi, 2008).

Since these algorithms are inspired by natural evolution, the two naturally have some things in common. They both rely on a *population*, which is required due to the fact that a single organism cannot evolve on its own. The population in turn is *diverse* to some extent. This basically means that the individuals in the population differ from each other in one way or another. These individuals have characteristics which can be *hereditary*. This means that these characteristics can be transferred down to any potential offspring generated by two individuals when they reproduce. The individuals chosen for procreation are done so in a process called *selection*. While these terms are common between the two, a major difference exist between natural evolution and artificial evolution. Natural evolution does not have a predefined goal, whereas artificial evolution does. (Floreano & Mattiussi, 2008).

The evolution process in an evolutionary algorithm is basically the following, as can be seen in Figure 6. It consists of a number of separate steps which are repeated until a set end condition is satisfied. This end condition could for example be that a set number of generations have evolved, or that a predefined time limit has been reached.

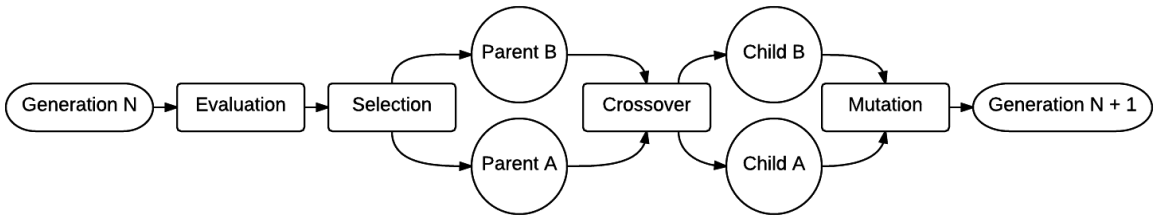

Figure 6 - Illustration of the algorithm (Johnsson, 2014).

2.6.1 Genetic representation

The initial population generated at the beginning of the algorithm consists of individuals. Each individual is represented by a chromosome, which in turn contains a set of genes. These genes

in turn encodes a possible solution to the problem the algorithm is meant to solve (Buckland, 2002). How the information needed to solve the problem is encoded in the genes is naturally problem specific, since in general no two problems are entirely alike. For example, a possible encoding for the game Gomoku which is used in this evaluation could be an X/Y position on the game board with a colour also associated with that position, as can be seen in Figure 7. A set of these genes would then represent a series of moves. In this case the set of genes in Figure 7 could represent one way from the root to a leaf in a *minimax* game tree.

In Java for example, each of these genes could be represented by an object with two integers representing the x and y position, along with a boolean telling if it's black or white. This would represent a move by the player of that colour, on that position on the board. In the context of the *genetic minimax* algorithm described in the previous chapter, the length of the chromosome represents the depth which will be searched in the game tree.

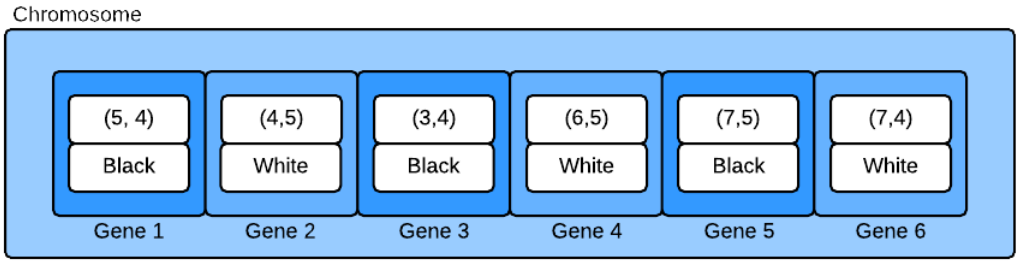

Figure 7 - Example of how a move sequence for the game Gomoku might be encoded.

An individual doesn't necessarily have to have this particular genetic representation though. An individual could also be represented by a string of characters from an alphabet, although not necessarily the English alphabet, where each letter is a gene. It could for example be represented by a sequence of values drawn from an alphabet of length 2, an alphabet which in this case could consist of the numbers 1 and 0. This is called a *binary representation* (Floreano & Mattiussi, 2008). A *binary representation* of an individual could be interpreted in many different ways. Floreano and Mattiussi (2008) gives the string 01010100 as an example. The string could be interpreted as the integer 84, but could also be interpreted in a more abstract way. Each bit in the string could represent a certain task that had to be completed. In this case there are 8 different tasks where each task corresponds to a bit in the sequence. Each bit could at the same time represent whether the task had to be done in the morning or in the evening. So in this case, the 0s represent a job that will be done in the morning, and the 1s are the jobs which will be done in the evening.

This type of representation can also be translated into a representation for the individual depicted in Figure 7. Given that the colour of the gene is always alternating within the chromosome and that the range of the range of each X/Y position does not exceed 16, the leftmost gene would then be represented by the byte 1010 0010. In this representation, the

first half represents the 5 and the second half represents the 4, written in binary. The position of this particular gene in the chromosome would then decide which colour the gene represented, since the colours alternate with the depth. This would simplify the gene in that it could fit within a single byte.

An alternate problem could instead be a labyrinth that had to be solved. If the directions you could move were limited to horizontal and vertical, as seen from above, only 2 bits would be needed to represent either moving north(00), south(01), east(10) or west(11).

2.6.2 A basic evolutionary algorithm

A basic evolutionary algorithm starts out as follows. First an initial population of individuals is generated. The size of this population depends largely on the problem size and how diverse you want the population to be. Large diversity is generally desired since it's important that the individuals display different fitness values (Floreano & Mattiussi, 2008). If the fitness values are not diverse enough then the selection does not operate properly.

After the initial generation of the population the evolution loop begins, with the population fed to it as the starting point. The first step is to evaluate each individual of the population to give them their fitness score. This fitness score represents how well they solve the problem, which, like the genome representation, is also problem specific. If the fitness score is high, then that individual is likely one of the better candidates for solving the problem. In the case of the game Othello, described in section 2.4.2, a simple fitness evaluation could be the difference in how many game pieces the two players have on the game board, if the sequence of moves that individual represents is carried out.

After that, two individuals from the population are selected based on what their fitness score was evaluated to. According to Floreano and Mattiussi (2008), the selection process can be done in several different ways. One way is called Roulette wheel selection which basically means that the individuals are selected with a probability proportional to their fitness. However, this does not guarantee that the best individual is selected, only that it has a very good chance of being so. Another type of selection process is for example Tournament selection (Buckland, 2002). Buckland (2002) describes a variant of Tournament selection where a number of individuals are randomly chosen from the population. These individuals are then added to a new population based on how high their fitness score is. The chosen individuals are not removed from the initial population, so they can be chosen again. This selection process continues until the new population is the desired size.

After the selection phase the two individuals are considered for *crossover*. Floreano & Mattiussi (2008) describes the crossover process as making sure that offspring of the selected individuals inherit some characteristics from their parents by means of pairwise recombination of the parents' genome. This basically means modifying the genome of the selected individuals by giving characteristics from the first individual to the second, and vice versa. Here also exists several variants of crossover operators which can be chosen from, much

like in the selection phase. One such variant is called *One-point crossover*. This operator selects a point at random along the two individuals' genomes, and swaps the genes after that point between the two individuals, as seen in Figure 8.

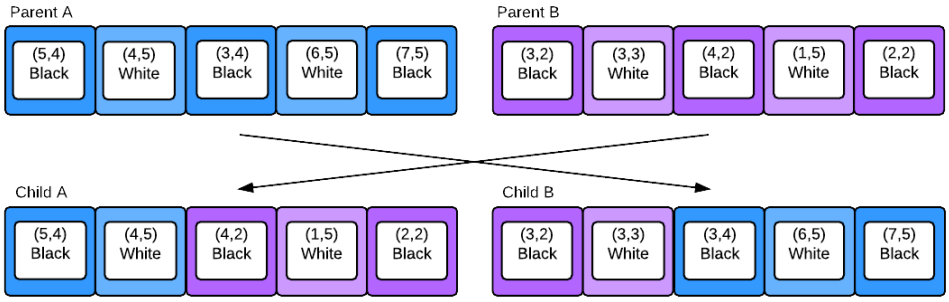

Figure 8 - Example of a One-point crossover between two genome.

The mutation phase is similar to the crossover phase, but it operates on the level of the individual rather than the pair. Floreano & Mattiussi (2008) describes the mutation as small random modifications to the genome which allow the evolution process to explore variations of existing solutions. The mutation is essentially the process of walking through each gene of the individuals and, based on some probability of doing so, changing the contents of this gene. The probability largely depends on the problem which is being solved, but common values are on the order of 0.01 per gene (Floreano & Mattiussi, 2008). A simple example is if the genes are represented by binary numbers. The mutation would then simply flip the bits around (a 1 turns into a 0, and a 0 turns into a 1) (Buckland, 2002).

The two mutated individuals are then added to a child population. When this child population has reached a set size, the best evaluated individuals are taken from this population and transferred to the parent population, either replacing or extending it. After that the next generation is evolved from the new parent population. This continues until a set condition is met, such as the number of generation reaching a certain point or a time limit being reached.

2.6.3 The Genetic Minimax algorithm

The *genetic minimax* algorithm operates on the principles mentioned in the previous section about evolutionary algorithms. What is special about this algorithm is that it incorporates elements of the minimax algorithm to help determine the value of an evolved genome. The algorithm does this by utilizing something referred to as a "Reservation tree" (Hong et. al, 2001).

The basic structure of the *genetic minimax* is the following. Just like a regular evolutionary algorithm it generates an initial population. The genetic representation of an individual in this population largely depends on the game. What is common between the representations for the *genetic minimax* algorithm is that the length of the genome is equal to the depth the algorithm searches in the game tree, and that each gene should represent a node in the game

tree. What may differ between the different representations is how the genes are encoded. An example gene encoding for the game Othello could be an X/Y position on the game board, along with which colour the disc placed was.

After the initial population has been generated, the end node of each individual is evaluated according to the evaluation function for the game. Depending on how the node is represented, the algorithm may need to play out the sequence of moves in order to evaluate the end node. The state at that end node may depend entirely on the sequence of moves being played out in exactly that order. This evaluation can be directly compared with evaluating a leaf node in a *minimax* game tree. The value assigned to the end nodes depends on which evaluation function the game uses. For Othello the value could be the difference between how many game pieces each player has on the board.

When the end node of an individual has been assigned a fitness value calculated from the evaluation function of the game, the individual is added to the structure referred to as the *reservation tree*. A simple example would be the four individuals ABA, AAB, ABC and ACA, which after being assigned their fitness values are added to a tree. These individuals would result in the tree structure seen in Figure 9.

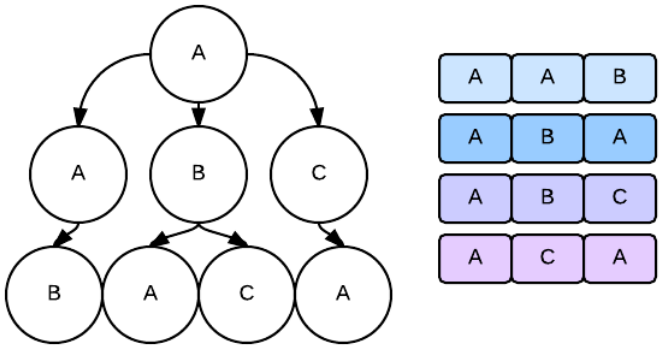

Figure 9 - Reservation tree with four genome added to it

If two or more individuals share the same genome up to a point, like the second and third individual in Figure 9, then the individuals are indistinguishable from each other in the tree structure up to that point. The nodes which represent their genome to that point is shared between them. In addition to adding the individuals to the tree, they are also added to a list so that then can be distinguished from each other.

When all of the individuals available have been added to the tree, the values of the end nodes are propagated upwards according to the *minimax* principle. In this case, since the leaf nodes are located at a minimising depth, the lowest value evaluated for a child leaf node is propagated up to its parent on the second level. From that level, the value of the node with the highest up-propagated value is then assigned to the root node, as seen in Figure 10.

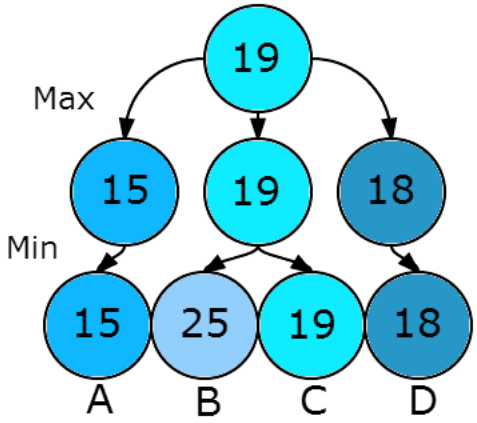

Figure 10 - Reservation tree with end node values propagated upwards

How far the value of an individuals end node is propagated up along the reservation tree determines how *genetically fit* that particular individual is. The higher its *genetic fitness* the closer that individual is to an optimal minimax decision, and vice versa. If a value manages to propagate all the way up to the root node, the individual to whom the value belongs would be equal to an optimal minimax decision at that depth, given enough diversity in the individuals added to the tree.

The *genetic fitness* is calculated by the following equation

H – Height of the *reservation tree*
K – The level the leaf value of an individual can be propagated up to
Genetic fitness = H - K + 1

By using this equation, the individuals in Figure 10 receive the values 2, 1, 3 and 2, respectively. The most fit individual of these four would be individual C, since the value of that leaf node is propagated all the way up to the root node and consequently given the highest *genetic fitness* value.

When the *genetic fitness* of all the individuals added to the reservation tree has been calculated, they are put up for selection, crossover and mutation. The individuals are taken from the list to which they were added previously. The selection is based on the calculated *genetic fitness* instead of the calculated value of the end node, while the crossover and mutation follow the principles described in Section 2.6.2.

The process of adding to the *reservation tree* and evolving based on what *genetic fitness* the individuals in the tree evaluate to continue until a set end condition is met. An end condition, at which point this choice is made, could be that an individual has achieved the highest *genetic fitness* score possible, or that a set number of generations have evolved.

A basic move made by the *genetic minimax* algorithm is as follows: Each move made by the *genetic minimax* algorithm starts with an initial population which is randomly generated based

on the current state of the game. The population is cleared and generated again every single time the algorithm is about to make a move, and the move is evolved from that newly generated population. Each generated individual represents a sequence of moves alternating between the two players, starting on the current game state, down to a set depth in the game tree. These individuals are then evaluated to determine their *genetic fitness* value, described earlier in this section. This value is used to determine which individuals are allowed to pass on their genes to the next generation through crossover and mutation. The selection process continue to select and evolve individuals from the current population until the next generation has enough individuals to match the current generation. When the next generation of individuals is the desired size, the algorithm starts to select individuals from this new population instead. When the desired number of generations have been evolved, either decided by a time limit or some other factor, the actual move the algorithm will make can be chosen. The transition to the first node of the individual with the best *genetic fitness* in the final generation is selected as the best move to make at this point. After the move has been made all information about the previous move is removed so the process can start fresh from the new game state, where the above process repeats. All of the above is repeated for every move the algorithm makes. In the most basic implementation of the *genetic minimax* algorithm no information is kept from one move to the next. Optimisations where still useful individuals are kept between the moves are possible, but will not be elaborated upon here.

3 Problem

When you analyse the game to which an AI algorithm will be applied, a common parameter to consider is the branching factor of the game tree. The branching factor is known to affect the search space complexity of a tree structure, since the more branches there are, the larger the tree becomes. This in turn leads to algorithms taking longer to search through the tree, and even longer if the tree is more than a few levels deep.

However, to the author's knowledge, little to no research has been done to evaluate if the branching factor of a game has any other effect on AI algorithms. In fact, recent findings suggest that this might actually be the case for certain algorithms. In a recent study, Johnsson (2014) determined that a possible cause for the performance problem experienced by her when evaluating a "Genetic-Minimax" hybrid algorithm in the board game Pentago was due to a too large branching factor in the game tree. This could not be verified however. In yet another recent study, it was shown that the branching factor in fact had a very large impact on the win ratio of a "Minimax-Monte Carlo Tree Search" hybrid algorithm in the game Breakthrough (Baier & Winands, 2014).

The aim of this thesis is to determine the following:

> *Does the branching factor of a tree structure in a perfect information board game have an effect on the performance of an evolutionary algorithm?*

With a base in the two studies mentioned above, the goal of this thesis is to evaluate whether or not the branching factor of a perfect information board game has an effect on the performance of an evolutionary AI algorithm. Much like how Baier and Winands (2014) showed that it did have an effect for a different kind of algorithm. The following hypothesis can in part be derived from the above two studies, and it is as follows:

Hypothesis:

> *The branching factor of the tree structure in a perfect information board game has an effect on the performance of an evolutionary algorithm.*

The performance of an algorithm is here defined as the ability to find (near) optimal solutions at a given depth of the game tree, similar to the Minimax algorithm, which will affect the win ratio of the algorithm.

Time is normally also an important factor when evaluating the performance of an algorithm. In the case of an AI algorithm, time is normally an issue since there may be time restrictions on how long the algorithm is allowed per move. However, the time factor has already been evaluated for this particular algorithm both by Johnsson (2014) and Hong et al. (2001), and it did appear to perform very well in that regard, so it will not be included here.

3.1 Motivation for the evaluation

An evolutionary algorithm has an element of randomness within. This manifests itself both when generating the initial population for the algorithm, and when the individuals of this population evolve. This element of randomness could potentially affect the algorithm in such a way that if the algorithm was used in a high branching factor game, it could be the cause of the algorithm selecting a bad move over a good move. If the branching factor is lower, then there are fewer choices to decide between. For example when a gene in an individual is mutating, it may then cause a higher chance of receiving a good mutation over a bad one for the gene.

3.2 Motivation for the evaluated games

The games Othello and Gomoku won't be directly comparable to Pentago regarding any performance difference due to a differing branching factor, but to assess whether the branching factor is a factor regarding the performance of the *genetic minimax* algorithm, they had to have a lower branching factor than what Pentago has. In addition to that, there are a few additional reasons for evaluating these particular games, and also why only two were evaluated. First, both of the games are common to use when AI algorithms are evaluated, since they are both *perfect information* board games, and are relatively easy to implement. These are the main reasons for evaluating Othello. The evaluation was limited to two games due to the time restrictions not allowing implementations for additional games to be made.

In addition to the above, the game Gomoku is an obvious choice since it is essentially the game from which Pentago originated, but a more basic version without any of the twists added. It is also one of the alternative games suggested by Johnsson (2014) as likely being a better alternative for the *genetic minimax* algorithm, which could also be desirable to verify.

There are of course many games with low branching factors which might suit this study equally well as the two mentioned above. One example of this is English Draughts, also known as Checkers (Allis, 1994).

A set of only two board games may not be enough to definitively assess the validity of whether the branching factor has an impact on the win ratio of an evolutionary AI algorithm in any board game. It could however provide enough data to point in one of the two directions.

3.3 Motivation for the evaluated algorithms

Given that there can be, and are, many variations of evolutionary AI algorithms it is infeasible to evaluate them all. Therefore this evaluation will focus on one particular evolutionary algorithm called *genetic minimax*. This algorithm will be set against an algorithm called *alpha-beta*. This evolutionary algorithm was also used in the study by Johnsson (2014), which is further described in Section 2.1.

The *genetic minimax* algorithm and the *alpha-beta* algorithm were chosen for a couple of reasons. They were primarily chosen, at least the *genetic minimax*, since it is an evolutionary algorithm, which is a premise for this evaluation. The *genetic minimax* algorithm also showed

some promise regarding execution speed in both the evaluation by Johnsson (2014) and the initial proposal by Hong et al. (2001), which is a valuable property. One additional reason is that under optimal conditions and with enough variation in the population, the *genetic minimax* algorithm should perform equally to the *alpha-beta* algorithm given that they operate at the same depth in the game tree. These two algorithms set against each other will also help show that the evolutionary *genetic minimax* actually works as it should.

4 Method

The hypothesis given earlier in Section 3 states the following:

> *The branching factor of the tree structure in a perfect information board game has an effect on the performance of an evolutionary algorithm.*

The method chosen to test this is to perform an experiment. This method is a good choice according to Berndtsson, Hansson, Olsson and Lundell (2008) since it is usually used in order to verify or falsify a hypothesis. In order to achieve this a set of sub steps can be defined. Each of these sub-goals should be small and achievable, and formulated so that fulfilling the objectives also fulfils the overall aim of the project (Berndtsson et al. 2008). In this thesis there is one main objective, to perform an experiment, which needs to be completed in order to fulfil the aim.

An alternative method to this could be to examine all available literature in order to find data which either corroborates or contradicts the hypothesis. There is however no guarantee that any relevant data would be found since the question posed is rather specific. The approach in this case could instead be to search for studies where evolutionary algorithms were evaluated for games where the branching factor can be varied, compile the data from all of the sources and then attempt to draw conclusions from that.

The experiment is meant to either validate or reject the previously formulated hypothesis about whether or not the performance of an evolutionary algorithm is affected by the branching factor of a game tree in a perfect information board game. The design of the experiment is based on the experiment Johnsson (2014) did in her study. The two algorithms she used were *genetic minimax* and *alpha-beta*. This experiment will compare the same two algorithms in the games Othello and Gomoku. The choice of including two different games was in part made to help determine if there is some cause other than the branching factor affecting the end result. If the hypothesis appears to be valid for one game but not the other, then the branching factor might not be the sole cause of the performance increase.

The first thing to do is to implement the actual algorithms and the environment. The language chosen for this is Java, mostly because it is fast to develop with and has an extensive library of support classes in case they are needed (Oracle docs, 2015). An effect of this choice of language might be a somewhat slower execution time compared to C++ for example, but that is very dependent on how the implementation is done. Speed is not an issue in this experiment however since the speed of the *genetic minimax* algorithm already has been evaluated, therefore Java is chosen due to the reasons mentioned above.

The *genetic minimax* algorithm will have to be implemented from scratch. The reason for this being that, to the author's knowledge, there is no readily available implementation to use. The closest you can get is the description of the algorithm in the original paper (Hong et al. 2001), and for more implementation details the more detailed description of the implementation done by Johnsson (2014) will be utilized. The *alpha-beta* algorithm will be

adapted for use in these particular games from pseudocode provided by Russel and Norvig (2010).

The simulations run for the games will be 200 simulated games at each depth in the game tree, up to and including a depth of 8. The high number of simulated games will help when analysing the data. If the number of simulated games per depth were very low, for example 20 per depth, then there is a higher risk of the results being biased towards one of the algorithms. This is because one algorithm might win many matches in a row, without actually being any better in winning matches if evened out over a larger sample size. If one algorithm were to win 15 out of 20 games on a depth where they *should* be equally matched if they had played enough matches, then the sample size is naturally too low. The relatively high amount of matches at each depth should help alleviate this, and help with the conclusion validity (Wohlin, Runeson, Ohlsson, Regnell & Wesslén, 2012).

The increase in depth compared to the simulations by Johnsson (2014) is an attempt to provide additional confidence in the results. The increase in depth could also help detect any patterns similar to what Baier and Winands (2014) experienced regarding any performance drop when searching deeper, since the size of the search space when searching deeper is affected by the branching factor. Additionally, one more set of simulations of 200 matches will be run on depths 1 – 8 for each game, but with the *genetic minimax* algorithm replaced by a player making moves at random. This is done to help boost the internal validity of the experiment. It should provide more confidence in that the *genetic minimax* algorithm versus the *alpha-beta* algorithm is actually better than just playing out random moves against the *alpha-beta* algorithm, since the *genetic minimax* algorithm is in part based on randomness.

5 Implementation

This section will focus on any important information as to how the algorithms and the games were implemented. This includes, among other things, details on the evaluation function for each game and specific settings regarding selection and probabilities in the evolutionary algorithm.

This information is important primarily from an ethical standpoint. The implementation details of both the games and the algorithms used has to be included so that anyone who wishes can reproduce the experiment, and consequently acquire the same results seen in this work.

5.1 Gomoku

Gomoku does not have any real restrictions on where a game piece can be placed on the board, which in this case is a 15x15 grid, apart from that the space has to be unoccupied. In this case, a limitation was put in place so that a game piece could only be placed if there was an adjacent game piece next to it in some direction. This includes horizontal, vertical and diagonal. So if there is only one game piece on the board, then there are 8 available positions to place the next piece. One exception is the first piece placed. The first one is always placed in the middle of the board. This restriction has the added side effect that the branching factor is lowered considerably due to less positions to consider each move.

The evaluation function for the implementation of this game is a basic one. The implementation used here was inspired in part by Johnsson (2014) who used a similar evaluation function for the game Pentago, which is very similar to Gomoku, and also by Gennari (2000) who proposed a similar strategy for Gomoku. The evaluation function is based on a priority system, where there are different levels of priority. Each of the levels is assigned a value, where the lowest priority is the lowest, and the highest priority is the highest. The evaluation function does not scan the entire board to evaluate if a move is good or not, but rather scans the affected rows, columns and diagonals when a game piece is placed. The evaluation function evaluates each of these three when a game piece is placed and assigns the move a value out of eight different categories. The categories are the following, in order of which one is prioritised:

- *isLosingState : -10000*
  If the move made causes either the immediate, or future, loss of the current game, then this is returned. This is a large negative value, since this is a very undesirable state to end up in. A losing state is defined as five consecutive game pieces in a row of the opponents colour.
- *isWinningState : 10000*
  If the move made causes the immediate, or future, win of the current game, this large value is returned. A winning state is defined as five consecutive game pieces of the current players colour.
- *winBlocked : 5000*

If the move made is the cause of blocking the win of the opponent, this is returned. This is the end result of blocking a capped four. A blocked win is defined by four game pieces of the opponent in a row, with each end capped by one of the players own game pieces.

- *hasOpenFour : 1000*
  If the move made creates an open four, which is four consecutive game pieces of the players own colour with neither end blocked, this is essentially the same as winning. It is however prioritized after *winBlocked* so that the AI does not create an open four rather than blocking an opponent win.
- *hasCappedFour : 500*
  The move made attempts to block the opponent from winning, putting a cap on one end of an open four. Rather futile move, but should be there nonetheless.
- *hasBlockedThree : 100*
  Blocks a capped three from becoming a capped four, making it blocked in both ends.
- *hasOpenThree: 50*
  If an open three can be created, then it should be. An open three consists of three consecutive game pieces in a row, with no end blocked.
- *hasOpenTwo: 10*
  Same as the above, but with two. This is the default move if none of the states above can be achieved. This will be reached due to the first move on the board always being placed in the center, not taking the evaluation function into account.

The evaluation function did not operate on the entire board. It only evaluates the row, column and diagonal that were affected by the move in question.

The genetic representation used by the *genetic minimax* algorithm for Gomoku is a set of moves made on the game board by the two players. The length of the chromosome is equal to the depth searched by the algorithm, and each gene is a position on the game board along with the color of the player making that move. The genes in the chromosome alternate between the two players, representing two players making alternating moves.

5.2 Othello

For Othello there are set rules as to where a game piece can be placed on the game board, so no special considerations have to be taken here. The size of the game board is the standard 8x8 size.

The evaluation function for Othello was a rather more simple option than the one described above. This function was adopted mostly for its simplicity to implement, and also due to limitations in the time available. In this evaluation function the entire board is evaluated on each move. The evaluation function counts the number of game pieces of each players colour and returns the difference between them as the fitness value. If the number is positive then the evaluating player has the advantage, and vice versa. This is known as *the maximum disc strategy* (Lazard, 1993).

The *maximum disc* strategy would be a very poor choice if used against some other AI incorporating some more advanced strategy, but in this case, both the players used this variant, so neither has an advantage in extra knowledge about the finer details of the game. If the goal was to create an AI which could play Othello well against AI players developed by others, then the evaluation function would have been far different.

If that were the case, with Othello being a little bit more complicated than simply valuing the game state by the number of game pieces present on it, some of the following could have been incorporated into the evaluation function. In Othello there is a term referred to as a "stable disc". This is a game piece which cannot be taken over by the opponent. Any disc placed in a corner is a *stable disc*, and discs of the same colour adjacent to these corner discs also often become *stable discs* (Lazard, 1993). An alternative approach could therefore be to favour game states in which the corner locations are occupied by discs of your own colour, and to try and avoid states where your discs are on the locations adjacent to the corner positions if the corners are unoccupied. If not, then the opponent can take the corner since you have a disc adjacent to it. Alternatively, the evaluation function could calculate the number of *stable discs* for a game state, and then assign a value to it based on this.

The genetic representation used by the *genetic minimax* algorithm for Othello is basically the same as for Gomoku. Each gene is a position on the game board along with the color of the player making that move. The genes in the chromosome alternate between the two players, representing two players making alternating moves.

5.3 Genetic Minimax

The *genetic minimax* algorithm was set up to evolve for 100 generations. The mutation frequency was set at 15% per gene, and the crossover frequency was set at 50%. For the selection process, *Roulette wheel selection* was used. The values for mutation and crossover frequencies was chosen due to them working best from the values tried. The values are quite far away from the commonly used values according to Floreano and Mattiussi (2008), at least for the mutation frequency which they say should be around 0.01, but for this particular problem the values chosen worked much better. For Gomoku, changing the values around did not make any difference to the results. Therefore the same values which were used in Othello were used. The number of generations they were allowed to evolve was primarily chosen due to the simulations otherwise would have taken too long to complete.

6 Results

Below, the raw data resulting from 200 simulations at search depths 1-8 are presented first for the game Othello, and then for the game Gomoku.

Equivalent simulations were also run where the *alpha-beta* algorithm was set against a player choosing its moves at random. The results from these simulations are shown in Table 3 and Table 5. The reasoning behind the random player is explained in Section 4, Method.

6.1 Othello

In Table 2 below the simulations for *alpha-beta* versus *genetic minimax* in the game Othello can be seen. At depths 1 - 4 they are very similar in performance, as they manage to win almost as many times. After that they start to diverge and the *alpha-beta* player gets the upper hand more and more as the depth increases.

Othello			
Depth	AlphaBetaPlayer	GeneticMinimaxPlayer	Tied
1	92	103	5
2	93	92	15
3	88	106	6
4	108	86	6
5	145	49	6
6	126	70	4
7	166	30	4
8	134	56	10

Table 2 - Raw simulation data resulting from playouts between the Alpha-beta algorithm and the Genetic Minimax algorithm in the game Othello.

Chart 1 below shows the data from the above simulations in a graph, to better illustrate how the performance of the algorithms varies as the depth increases.

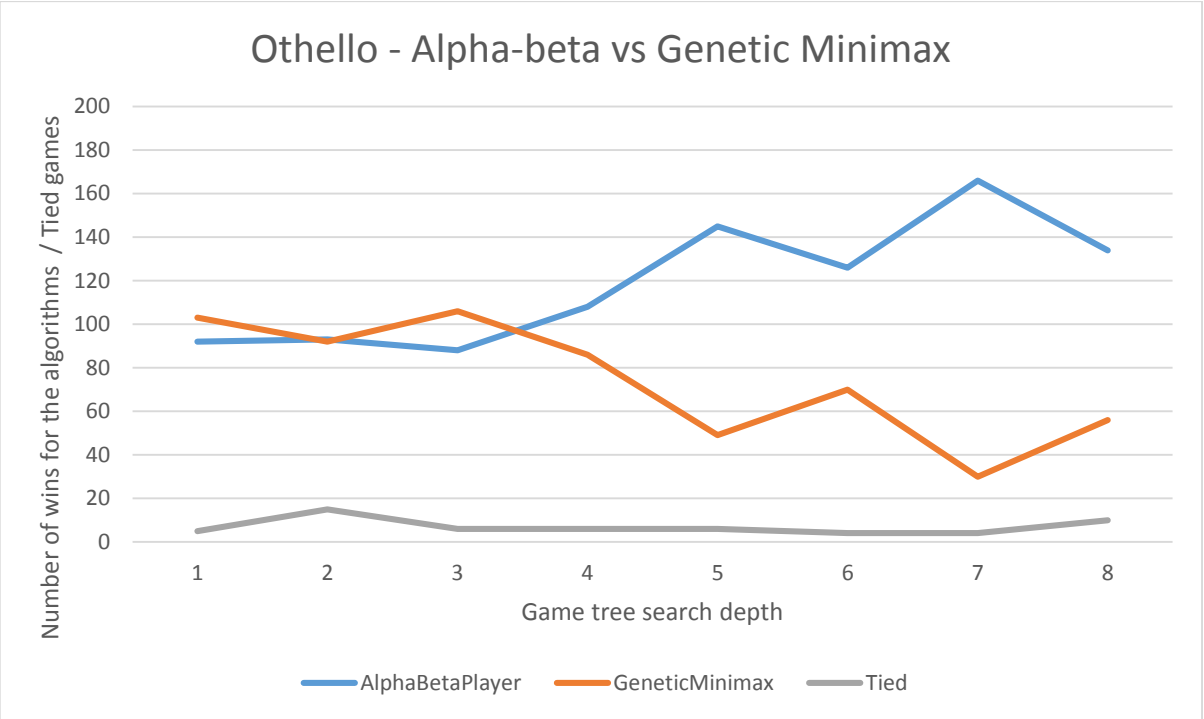

Chart 1 - The simulation data from the Alpha-beta algorithm vs the Genetic Minimax algorithm for the game Othello, plotted in a chart.

The following table depicts the *alpha-beta* algorithm versus a random player in the game Othello. The random player does not take depth into consideration, instead just playing out a random, legal, move which can be made on a given game state.

Othello			
Depth	AlphaBetaPlayer	RandomPlayer(No depth)	Tied
1	123	72	5
2	115	69	16
3	129	64	7
4	138	56	6
5	160	32	8
6	151	41	8
7	182	13	5
8	165	31	4

Table 3 - Raw simulation data resulting from playouts between the Alpha-beta algorithm and a random player in the game Othello.

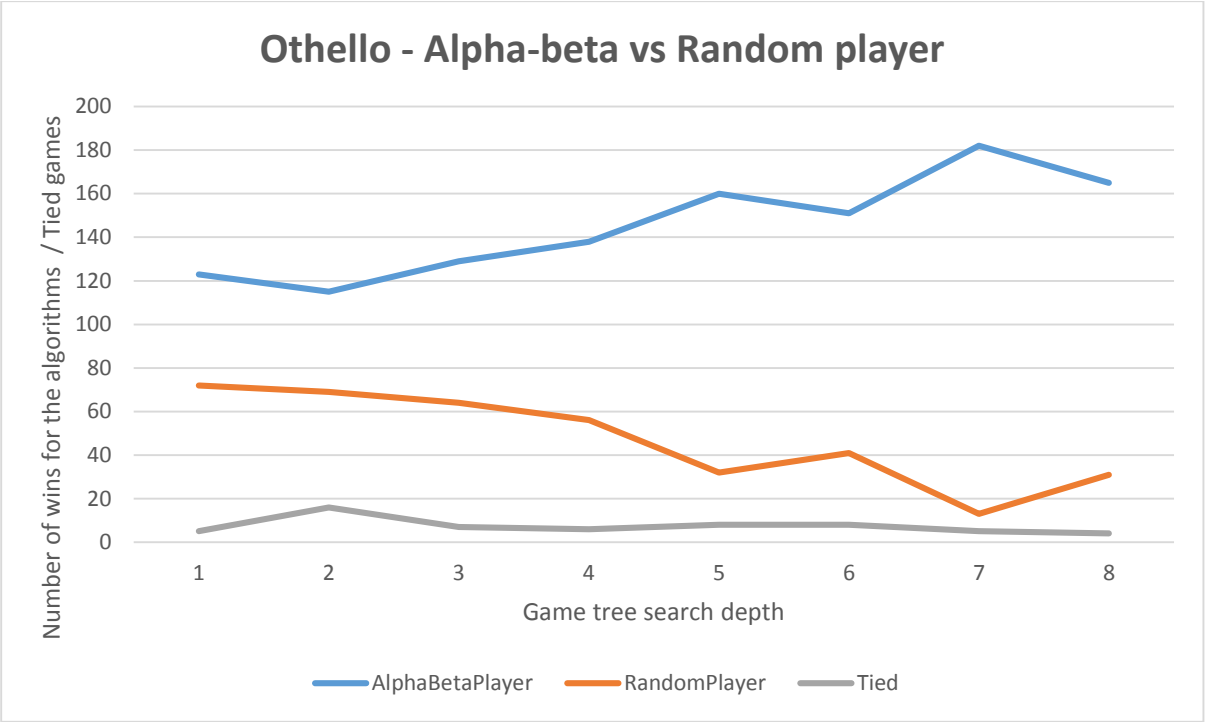

Chart 2 - The simulation data from the Alpha-beta vs Random player for the game Othello, plotted in a chart.

In Chart 2 above you can see the results from the simulation between the random player and the alpha-beta algorithm. The graphs are very similar in shape to the ones seen in Chart 1, with the exception being that in Chart 1 the graphs for the players are generally closer together, and almost equal at depths 1 – 4. For the *alpha-beta* versus the random player seen in Chart 2 the two algorithms start out relatively close to each other, with only about 50 wins apart at depth 1 in the search tree, but start diverging more as the depth increases. The two algorithms diverging like this is expected since the *alpha-beta* algorithm has some logic behind its decisions, causing it to make better moves which in turn causes it to win more the deeper it's allowed to search.

The relatively high number of wins for the random player at the higher depths may be caused by the relatively simple evaluation function used by the *alpha-beta* algorithm in Othello. The *alpha-beta* algorithm in its current state does not know that certain positions on the Othello game board are extra beneficial to possess. Among these positions are for example the corner-positions of the board, since they can never be lost once claimed. The random player may, at the higher depths such as 6, 7 and 8 for example, have managed to claim one or more of these beneficial positions, since the *alpha-beta* algorithm did not know they were good to own, which in turn may have given the random player a large advantage. Eventually this may have been the cause of the random player winning as much as it did even though the *alpha-beta* had the advantage of having at least some logic behind its decisions.

6.2 Gomoku

The following table show the results for the *alpha-beta* algorithm versus the *genetic minimax* algorithm in the game Gomoku.

Gomoku			
Depth	AlphaBetaPlayer	GeneticMinimaxPlayer	Tied
1	179	21	0
2	200	0	0
3	200	0	0
4	200	0	0
5	200	0	0
6	200	0	0
7	200	0	0
8	200	0	0

Table 4 - Raw simulation data resulting from playouts between the Alpha-beta algorithm and the Genetic Minimax algorithm in the game Gomoku.

Here in Table 4 we can see that the results are very one sided in favour of the *alpha-beta* algorithm. The exception is at depth 1 where the *genetic minimax* managed to win about 10% of the games played.

Just to illustrate, the data from Table 4 can also be seen in Chart 3 below.

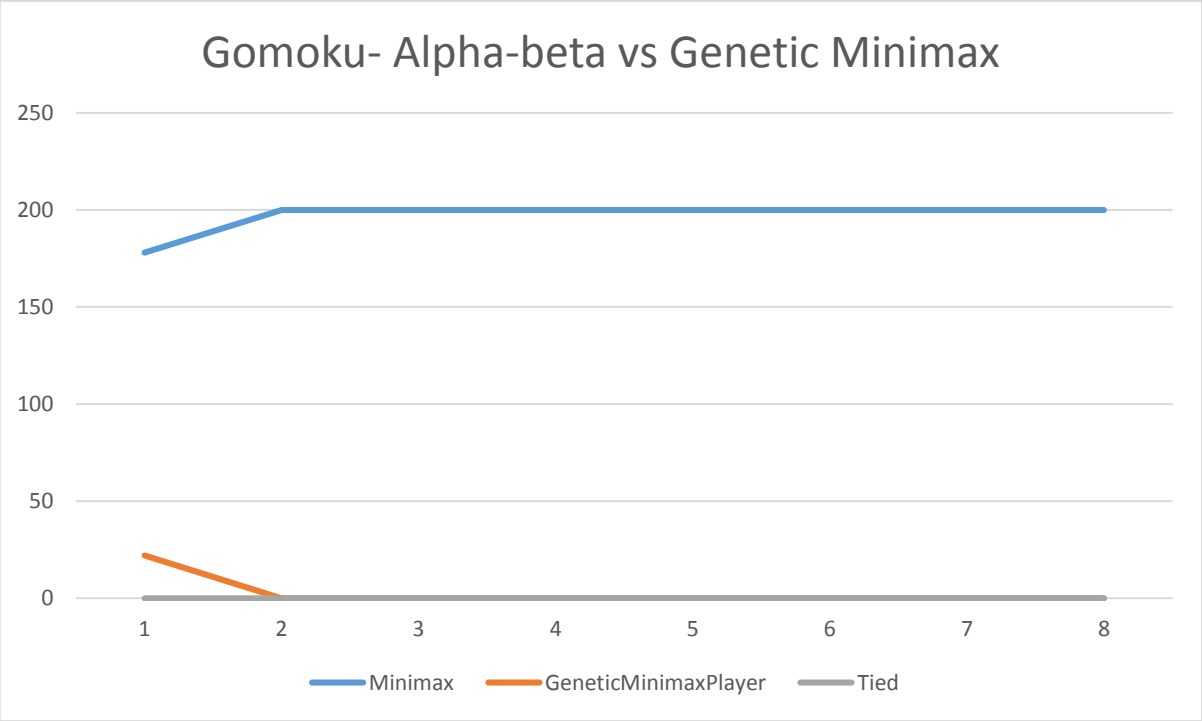

Chart 3 The simulation data from the Alpha-beta vs Genetic Minimax for the game Gomoku, plotted in a chart.

Table 5 below shows how the random player fared against the *alpha-beta* algorithm in Gomoku. In this case it was very one sided in favour of the *alpha-beta* player, where it won every single game at every depth.

Gomoku			
Depth	AlphaBetaPlayer	RandomPlayer(No depth)	Tied
1	200	0	0
2	200	0	0
3	200	0	0
4	200	0	0
5	200	0	0
6	200	0	0
7	200	0	0
8	200	0	0

Table 5 - Raw simulation data resulting from playouts between the Alpha-beta algorithm and a random player in the game Gomoku.

7 Analysis

All of the following observations are limited to the specific algorithms used in this evaluation, when used in the games Othello and Gomoku. The observations are as follows, based on what we see in Section 6.

First, simulations are run for the game Othello, both with the *alpha-beta* algorithm versus the *genetic minimax* algorithm and then, as a reference, *alpha-beta* versus a random player. In the first case visualised in Chart 1, *alpha-beta* versus *genetic minimax* in the game Othello, we can observe that the performance of the *genetic minimax* algorithm is about equal to that of the *alpha-beta* algorithm up to and including depth 4. This behaviour is expected since an optimal move for the *genetic minimax* algorithm should be equal to that of the *alpha-beta* algorithm searching to the same depth. If this is compared to the same depth range on Chart 2 you can see that there is a definite increase in the performance of the *genetic minimax* algorithm compared to just choosing random moves. Since the *genetic minimax* algorithm is in part based on random selection, this would indicate that the *genetic minimax* algorithm managed to evolve enough at these depths to match the performance of the *alpha-beta* algorithm as it should. The random player in Chart 2 has a relatively steady decline in performance as the depth increases, which also is to be expected since the *alpha-beta* algorithm can make better choices the deeper it is allowed to search. This in contrast to the random player which simply picks a legal move at random without considering either the depth or any implications that move might have.

After depth four the performance of the *genetic minimax* algorithm in Chart 1 start dropping considerably while the performance of the *alpha-beta* algorithm rises at approximately the same rate. Up until depth 5 the *genetic minimax* algorithm both matched, and at some depths surpassed, the number of wins by the *alpha-beta* algorithm. If more simulations had been run at each depth, this may have evened out further. One explanation for this behaviour is that there may simply be too many paths at these depths to take in the tree for the *genetic minimax* algorithm to evolve a good solution.

It is difficult to say whether or not the branching factor is a direct cause of this performance drop however. In favour of the branching factor being the cause, or at least part of the cause, is that the higher the branching factor, the more combinations of paths in the game tree from the root to the leaves exist. In addition to that, as the depth increases the number of combinations increase even further. This is due to the branching factor in combination with the depth of the tree structure having a direct correlation with the number of different paths in the tree. Given a sufficiently large number of paths from the root to every leaf in the tree, the *genetic minimax* algorithm has a far lower chance of converging on a good solution than in a tree with fewer paths.

As described previously in Section 2.6, an evolutionary algorithm relies heavily on changing parts of an initially randomly generated move sequence in order for it to gain as large a fitness

value as possible. These changes are made at random, both mutation and crossover. Since the *genetic minimax* algorithm only has a set amount of generations to change these parts around, and since the number of generations the algorithm was allowed to evolve was set to a constant at all depths, it might not have had enough generations available to evolve a good enough move sequence as many times as it did on the lower depths. At the lower depths the set amount of generations it was allowed to evolve might have been just enough for it to perform relatively well due to the lower amount of move sequence combinations available in the tree. It simply had a far lower amount of branches to consider when evolving, which caused the random elements of the algorithm to evolve good characteristics more often.

However, as the depth increased above level four in Chart 1, the number of possible move sequence combinations may have been too many. Since the increase in move sequence combinations is exponential (or close to, depending on the branching factor), the difference in the amount of combinations between two consecutive depths can be very large. If, hypothetically, the game tree had a constant branching factor of 10. At depth 4, that means a total of 10000 possible unique paths from the root to the leaf nodes are available. At depth 5, this increases even further by a factor of 10 to 100000 possible unique paths.

With the current simulation settings the *genetic minimax* algorithm handles the game tree well up to depth four but not from depth five onwards, compared to the *alpha-beta* algorithm. The *alpha-beta* algorithm has no random element involved in how it executes, so it remains unaffected by the branching factor in every way except for the time taken to perform the search. That is, the *genetic minimax* algorithm manages to evolve a good enough move sequence to win about as many times as the *alpha-beta* algorithm at depths 1 – 4. Just as it is meant to. After depth five it appears to be affected even more so by the move sequence combination increase, but the decrease in performance is relatively slow as the depth increases. It doesn't plummet to 0 wins instantaneously. The branching factor may then have been the cause of the performance decline of the *genetic minimax* algorithm observed in Chart 1 from depth five and forward, that in combination with the random elements of the algorithm and the depth increase.

Since the *genetic minimax* algorithm is designed to perform as well as the *alpha-beta* algorithm at the same depth, it is a little bit surprising that the *genetic minimax* algorithm actually managed to surpass the *alpha-beta* algorithm at depths one and three in Chart 1. Especially since it did this on uneven depths where the *alpha-beta* is generally stronger due to them being maximising depths. The opposite can be observed from depth five and onwards in Chart 1 where the *alpha-beta* algorithm has some performance dips on the even depths. This may, in part, be explained by the fact that on even depths the algorithm only gets to search to a minimising depth. That is, a depth where, in the algorithm, the opponent gets to make the decision on which leaf node value is propagated up first. This is also known as the "Odd-Even Effect" (Hauk, Buro & Schaeffer, 2004). This may also be why the *genetic minimax* performed well on depth 1 and 3, since it is based in part on the *minimax* algorithm. It is

unclear however why the advantage at the lower depths was for the *genetic minimax*, and the reverse at the higher depths.

The relatively good performance of the completely random player against the *alpha-beta* algorithm in Othello, seen in Chart 2, may have an explanation in the relatively simple evaluation function implemented for Othello. In Othello, certain positions on the board are worth more than others, such as corner positions which can't be lost once taken, see Section 5.2. However, the evaluation function used does not take this into account. This leads to the random player being allowed, more than it would have been otherwise, to choose between positions where corner positions might be included, which could give it an advantage.

For the game Gomoku, the following can be observed. The data in Table 4 and Chart 3 clearly shows that the *genetic minimax* algorithm had some issues with the game Gomoku, similar to how Johnsson (2014) had some issues with the same algorithm for the game Pentago. In her case it was not this severe though. In her simulations the *genetic minimax* algorithm and the *alpha-beta* algorithm managed to perform equally on depth 1, but almost as one sided as the simulations performed here on depths above that, see Table 6. A possible explanation for this, may be that the implementation of the *genetic minimax* algorithm was somehow wrong for Gomoku.

However, as can be seen in Table 4 the *genetic minimax* did manage to win around 10% of the matches at depth 1 versus the *alpha-beta* algorithm in this evaluation, so the implementation being flawed may not be the only explanation. Another possibility might be that the evaluation function used was somehow not the most ideal option. Since the evaluation function for Gomoku was based in part on what Johnsson (2014) used for the game Pentago, with the games being very similar, and the results being what they are, it may be what is to blame. The pattern seen in Chart 3 and the data from Table 4, shows that the *genetic minimax* could at least win some matches at depth 1. This is also seen in the results by Johnsson (2014), see Table 6. In her case it fared far better though, even managing to win a few matches at some of the higher depths. This discrepancy may have been caused by her implementation evolving for more generations, which may also have caused her implementation to also win some matches at the higher depths.

Depth	AlphaBetaPlayer	GeneticMinimaxPlayer	Tied
1	100	100	0
2	188	0	12
3	196	4	0
4	200	0	0
5	195	5	0

Table 6 - Number of wins and losses for each algorithm at a given depth, as tested by Johnsson (2014)

One final explanation could be that the *genetic minimax* algorithm is simply unsuitable for the game Gomoku. If this is the case, given that Pentago and Gomoku are very similar, then the conclusions Johnsson (2014) drew about the branching factor being the cause for her

experienced performance problems may not be entirely accurate. At least not in the sense that the performance problem of the *genetic minimax* algorithm in the game Pentago was due to the branching factor of the game.

The reason for the unsuitability of the *genetic minimax* algorithm for Gomoku could be that the goal of the game is to form a pattern while simultaneously trying to block the opposing player creating the same pattern. This could also be related to how the evaluation function is constructed. This in stark contrast to the game Othello where the goal is to "simply" gain the advantage in the number of game pieces on the board with your colour at the end of the game. The "Genetic Fitness" which the *genetic minimax* algorithm relies heavily on to determine if a given move sequence is the most optimal one did not work particularly well here, while the prioritising evaluation function the *alpha-beta* algorithm used directly worked far better. The *genetic minimax* algorithm also utilised this prioritising evaluation function to some extent, but the final decision on which move was the best relied solely on which evolved individual evaluated to the highest "Genetic Fitness" value.

If the *genetic minimax* algorithm were to play against another player, or an algorithm other than *alpha-beta*, the performance of the *genetic minimax* would be comparable to that of the *alpha-beta* algorithm playing under the same circumstances if the *genetic minimax* algorithm is allowed to evolve enough. This is true at least for the game Othello since we have observed that the *genetic minimax* actually works in that game, and that it has comparable performance to the *alpha-beta* algorithm. It is possible that this could be extended to also apply for the game Gomoku, if it is possible to modify the implementation of the *genetic minimax* algorithm to work better in that game. As mentioned previously the problem may also lie in the evaluation function used for Gomoku, so an alternative to modifying the algorithm itself could be to use a different evaluation function.

The role of the genetic representation which the *genetic minimax* algorithm uses is unclear. At least in regard as to how well the algorithm performs in a specific game. The genetic representation an individual uses is a sequence of genes which is equivalent to a path from the root of the game tree to the leaf node represented by the final gene in the sequence. This representation can in this algorithm, to the author's knowledge, not be replaced by another representation. Since the algorithm is very dependent on the reservation tree-structure created by the specific genetic representation of each individual used in the algorithm, it is hard to imagine it having any real negative effects other than the algorithm being limited in performance to an equivalent minimax evaluation.

8 Conclusion

Despite the conflicting results regarding the performance of the *genetic minimax* algorithm in two low branching factor board games, a few conclusions can be made. First, there is some evidence to support the hypothesis about the branching factor having an effect on the performance of an evolutionary algorithm. In this evaluation however, this conclusion can only be clearly drawn for the board game Othello when the evolutionary algorithm *genetic minimax* is used, with the genetic representation utilized in this implementation. The analysis indicates that the branching factor of that game could have an effect on the performance of that particular evolutionary algorithm. This conclusion was not arrived at by varying the branching factor within either of the games, but rather by looking at how the algorithm behaved in the game Othello. If it had been possible to vary the branching factor within that game, then the graph in Chart 1 may have shifted to the left if the branching factor was raised, causing the algorithm to decline in performance at a lower depth than it did here. If the branching factor in the game had been lowered then the graph in Chart 1 may have instead shifted to the right, causing the algorithm to work well even deeper than it did here, and instead starting to decline at a greater depth than it did here. This is not verified of course, but rather arrived at from the analysis of why the *genetic minimax* algorithm suddenly started to decline in performance after a certain depth in the game Othello.

For the game Othello, it could be possible to extend this conclusion to other evolutionary algorithms which work for that game, and possibly other types of algorithms as well, given that they operate in a way similar to the *genetic minimax* algorithm. To extend this conclusion for other perfect information board games, or other types of board games in general, additional evaluations would have to be made.

In addition to this, there is very little data which at this time can contradict the hypothesis, at least for the game Othello. For the game Gomoku the results for the *genetic minimax* algorithm were a little bit disappointing in the sense that it did not work very well at all, and that very little useful data was gained from the results. In the analysis a few suggestions on why was made, but there was not enough data to interpret it in any direction regarding if the branching factor had any effect on the performance in that case.

Realistically, there needs to be a more systematic evaluation of evolutionary algorithms in these types of board games to give a definitive answer to the question and hypothesis posed in Section 3, and also to rule out one algorithm working better than another in a game due to factors other than the branching factor. More careful consideration also has to be taken when choosing the evaluation function for the games. A short discussion on this is given in Section 8.2, Future Work.

8.1 Discussion

The idea of evaluating if an evolutionary AI algorithm was affected by the branching factor of the game tree to which it's applied appeared to be a novel idea. The method chosen was also

the better one of the ones available, since an experiment is usually a good way to verify or falsify a hypothesis (Berndtsson et al. 2008). The method of approach within the experiment however, could have been significantly better. Only evaluating a single algorithm in two different environments, which in turn only represented two different branching factors, may not have been the best way to arrive at a generalised result regarding all evolutionary algorithms.

The variable for this experiment was the branching factor of a board game. The intent was to examine if the performance of an evolutionary algorithm was affected when the branching factor was varied. The initial method of approach was to have the variation in branching factor separated over three different games, where the game with the high branching factor was a third party study by Johnsson (2014), and the two games evaluated in this study being the ones with low branching factors. Later on this turned out to be a rather poor method of approach, so the comparison with the third party study was scrapped. The decision to stick with the two remaining games was made due to very little time remained in the project, but some data could be gathered from these two games nonetheless.

The better method of approach for the entire study would have been to perform the evaluation for one or more games where the branching factors could be varied within the games. It would have been most ideal if this had been done from the very beginning. If this had been done, then there could have been clear and concise comparisons between the different branching factors within the games. A clear connection would have existed between how the used algorithms performed and what the branching factor was set to. More than one game where the branching factor could be varied could have strengthened the results further, and no comparisons between the games would have been necessary since the variation in the branching factors were contained within the games themselves.

The implications of this work could in the near future be that it could act as an inspiration for further work in the same area, such as what is described above. One practical implication could be that evolutionary algorithms such as the *genetic minimax*, and algorithms similar to it, may be considered less useful for board games with high branching factors. Also, if more evaluations are made in the same area, a more generalised conclusion on evolutionary algorithms in these types of games could be made. This could in turn lead to evolutionary algorithms in general being used less in these types of situations.

8.1.1 Ethical concerns

Any concerns regarding research ethics are handled by means of explaining how the experiment was conducted, which algorithms and games were used and any specific implementation details within them. This also includes explaining how the algorithms used work. All of this is included so that anyone who wishes can reproduce the experiment, using the same algorithms and games, and get the same results which were presented here.

8.2 Future work

In the future a more systematic approach might be a better option on how to perform a similar evaluation. Although the results from this experiment did indicate that the branching factor may have had an impact on the win-ratio of the *genetic minimax* algorithm, albeit only in one of the tested games, this is likely not definitive evidence in favour of the hypothesis stated in Section 3 about whether the branching factor of a perfect information board game has a large effect on the performance of all evolutionary AI algorithms.

A more systematic approach might be to find some game where the branching factor can be either increased or decreased in an easy way without affecting anything else in the game. Then select a number of algorithms, either evolutionary or not and depending on how thorough you'll want the evaluation to be, where some random element in the algorithm has an effect on which move it chooses to make. Alternatives for this are for example other genetic algorithms or algorithms based on Monte Carlo Tree Search, the later which was mentioned briefly in Section 2.1. Both of these employ some form of randomisation within. Any algorithm chosen would have to be examined to determine whether or not the randomisation affects its choice of move. Multiple variations of the different types might also strengthen the results further.

9 References

Allis, V. (1994). *Searching for solutions in games and artificial intelligence*. Maastricht: Rijksuniversiteit Limburg.

Atallah, M. (1999). *Algorithms and theory of computation handbook*. Boca Raton: CRC Press.

Baier, H., Winands, M. (2014). MCTS-Minimax Hybrids. *Computational Intelligence and AI in Games, IEEE Transactions on*, PP(99), p.12.

Berndtsson, M., Hansson, J., Olsson, B., & Lundell, B. (2008). *Thesis projects - A guide for students in computer science and information systems*. Springer.

Black, P. (2014). full binary tree. Retrieved June 1, 2015, from Dictionary of Algorithms and Data Structures: http://xlinux.nist.gov/dads//HTML/deterministicAlgorithm.html

Buckland, M. (2002). *AI techniques for game programming*. Cincinnati, Ohio: Premier Press.

Floreano, D., & Mattiussi, C. (2008). *Bio-Inspired Artificial Intelligence – Theories, Methods and Technologies*, The MIT Press, Cambridge, Massachusetts, London, England.

Gennari, J. (2000). Game-Playing Search and Gomoku. Retrieved June 4, 2015, from University of Washington Faculty Web Server: http://faculty.washington.edu/gennari/teaching/AI-171/GomokuProject.pdf

Hauk, T., Buro, M., & Schaeffer, J. (2004). *-Minimax Performance in Backgammon. i Y. Björnsson, N. S. Netanyahu, & H. Herik (Red.), 4th International Conference, CG 2004, (ss. 51-66). Springer Berlin Heidelberg, 5-7 Juli, Ramat-Gan, Israel.

Hong, T.-P., Huang, K.-Y., & Lin, W.-Y. (2001). *Adversarial search by evolutionary computation*. Evolutionary Computation, 9(3), 371-385

Johnsson, S. (2014). *Ai till brädspel : En jämförelse mellan två olika sökalgoritmer vid implementation av AI tillbrädspelet Pentago*. Bachelor thesis. University of Skövde, The School of Informatics. http://urn.kb.se/resolve?urn=urn:nbn:se:his:diva-9426 [2015-01-31]

Lazard, E. (1993) Strategy guide. Retrieved April 16, 2015, from radagast.se: http://www.radagast.se/othello/Help/strategy.html

Levinovitz, A (2014), 'THE MYSTERY OF GO, THE ANCIENT GAME THAT COMPUTERS STILL CAN'T WIN', Retrieved March 1, 2015, from Wired: http://www.wired.com/2014/05/the-world-of-computer-go

Millington, I. and Funge, J. (2009). *Artificial intelligence for games*. Burlington, MA: Morgan Kaufmann/Elsevier.

Oracle corp. (2015). Overview (Java Platform SE 8). Retrieved April 1, 2015, from Oracle docs: http://docs.oracle.com/javase/8/docs/api/

Papadopoulos, A., Toumpas, K., Chrysopoulos, A., & Mitkas, P. A. (2012). *Exploring optimization strategies in board game Abalone for Alpha-Beta search*. 2012 IEEE Conference on Computational Intelligence and Games (CIG), (pp. 63 - 70). Institute of Electrical and Electronics Engineers (IEEE), 11-14 September 2012, Granada, Spain.

Russell, S. J., & Norvig, P. (2010). *Artificial intelligence, a modern approach*, 3rd edn, Pearson/Prentice Hall, Upper Saddle River, New Jersey.

Wang, J. and Huang, L. (2014). Evolving Gomoku Solver by Genetic Algorithm. *Advanced Research and Technology in Industry Applications (WARTIA), 2014 IEEE Workshop on*. IEEE, pp.1064 - 1067.

Wohlin, C. Runeson, M. Ohlsson, M. C., Regnell, B., Wesslén, A. (2012). *Experimentation in software engineering*. Springer-Verlag, Berlin Heidelberg.