DEGREE PROJECT IN TECHNOLOGY

FIRST CYCLE, 15 CREDITS

Optimization Areas of the Minimax Algorithm

A study on a look-ahead AI as applied to the Fox game

LUDWIG FRANKLIN

HUGO MALMBERG

Stockholm, Sweden 2023

Optimization Areas of the Minimax Algorithm

A study on a look-ahead AI as applied to the Fox game

LUDWIG FRANKLIN

HUGO MALMBERG

Degree Programme in Computer Science and Engineering
Date: June 9, 2023

Supervisor: Erik Fransén
Examiner: Pawel Herman
School of Electrical Engineering and Computer Science
Swedish title: Förbättringsområden för minimax algoritmen
Swedish subtitle: En studie på en ”look-ahead” AI när den tillämpas på räv-spelet

Abstract

Artificial intelligence has become more prevalent during the last few years, revolutionizing the field of computer game-playing. By incorporating artificial intelligence as a computerized opponent, games can become more engaging and challenging for human players.

The minimax algorithm when applied to a two-player turn-based game looks a given number of steps ahead into the future and determines which move leads to the best scenario, given that the opponent plays optimally. The algorithm can be applied to a wide range of different games, the algorithm itself is quite simple and depending on the type of game it is often hard to perform better in the game than an AI using this algorithm. One big disadvantage of the algorithm however is that it is relatively slow since the amount of possible scenarios it has to consider often grows exponentially with each turn.

This thesis presents an analysis of various strategies to accelerate the performance of the minimax algorithm, with a particular focus on versions with and without alpha-beta pruning. The study further explores the performance implications of parallelization, examines how the optimal move selection rate is influenced by search depth, and assesses whether dynamic search depth offers a viable means of balancing optimal move selection rate and execution speed.

The results derived from the study indicate that alpha-beta pruning is more effective with higher search depths. It also indicated that alpha-beta pruning allowed for one step deeper searches compared to the standard minimax algorithm, for almost no added computational cost. The impact of parallelization varied, proving beneficial for deeper searches in the non-pruning version but had almost no impact on the alpha-beta pruned version. The optimal move selection rate did increase with added depth, but more data is needed for conclusive results. In regards to dynamic search depth, we found it only proved effective for the standard minimax algorithm and that it is better to use alpha-beta pruning.

Sammanfattning

Artificiell intelligens har blivit mer vanligt under de senaste åren, och det har påverkat datorspelsbranchen rejält. Genom att låta artificiell intelligens styra motståndare i spelen, så kan de bli mer engagerande och utmanande för spelarna.

Minimax-algoritmen när den appliceras på en två-spelar turbaserat spel kan undersöka ett givet antal steg in i framtiden och avgöra vilket drag som leder till den bästa resultatet, givet att motståndaren spelar optimalt. Algoritmen kan appliceras på en mängd olika spel, algoritmen i sig är ganska simpel och beroende på vilken typ av spel det är kan det ofta vara svårt att prestera bättre i spelet än en AI som använder denna algoritm. En stor nackdel med algoritmen är dock att den är relativt långsam eftersom mängden möjliga scenarier den måste överväga ofta växer exponentiellt med varje tur.

Denna rapport presenterar en analys av olika strategier för att förbättra prestandan av minimax-algoritmen, med ett särskilt fokus på versioner med och utan alpha-beta pruning. Studien undersöker vidare prestandakonsekvenserna av parallellisering, undersöker hur valet av optimala drag påverkas av sökdjupet och bedömer om dynamiskt sökdjup kan användas för att balansera valet av optimala drag och hastighet.

Resultaten som härleds från studien indikerar att alpha-beta pruning är mer effektivt vid högre sökdjup. Det indikerade också att alpha-beta pruning tillät en steg djupare sökningar jämfört med standard minimax-algoritmen, för nästan ingen extra beräkningskostnad. Påverkan av parallellisering varierade mellan versionerna, det var fördelaktigt för djupare sökningar i den icke-prunade versionen men hade nästan ingen inverkan på alpha-beta prunade versionen. Valet av optimala drag ökade med större djup, men mer data behövs för ett definitivt resultat. Vad gäller dynamiskt sökdjup fann vi att det endast var effektivt för standard minimax-algoritmen och att det är bättre att använda alpha-beta pruning istället.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Artificial intelligence (AI) has become more prevalent during the last few years, revolutionizing the field of computer game-playing. By incorporating AI as a computerized opponent, games can become more engaging and challenging for human players. Typically, a computerized opponent consists of various logical functional modules, with look-ahead being one of the most commonly used. Look-ahead is a type of AI most utilized for two-player board games and it is for example used in the famous chess-engine Stockfish. The algorithm analyzes each possible future outcome and returns the optimal one in regards to predetermined heuristics, given that the opponent plays optimally.

In this thesis, we apply the look-ahead algorithm to the Fox-game, which is an old Scandinavian board game where one player controls the hens whose goal is to move from one end of the board to the other, while the other player controls the foxes and aims to stop the hens from achieving their goal by jumping over them and thus removing them from the board.

1.1 Problem Statement and Purpose

This report will analyze the performance in regards to computations needed for the minimax algorithm when applied to the Fox-game with different implementations regarding:

With and without alpha-beta pruning (α-β pruning). How does α-β pruning affect the performance of the minimax algorithm?

Parallelization, how does running the algorithm in parallel affect the performance?

Lastly we will look at alternating the search depth. How much more computation does deeper searches need and how often is the chosen move different when searching deeper in comparison to shallower searches?

We will look at all of these factors in order to get a better understanding of how to optimize the algorithm further.

1.2 Scope and Limitations

Due to time constraints we chose a simpler approach to parallelizing the α-β pruned algorithm, using local α and β values that are thread specific instead of synchronized values. This should not effect the results significantly, but the limitation is noteworthy since it differs from other research in the same area, see section 2.4.

Chapter 2

Background

2.1 Game Theory in Artificial Intelligence

The area this thesis focuses on is a branch of mathematics called Game Theory. Game Theory is used to model the strategic interaction between different players in games or other context with predefined rules and outcomes [1].

Artificial Intelligence can be applied to these models in order to help humans solve games. In complex games with many different outcomes, it can be hard for humans to calculate or predict outcomes. This is where the help of computers comes in [1].

There are many categories of games out there. The type of game focused on in this thesis is called 2-player, zero-sum game. The meaning of zero-sum is that one person's gain is equivalent to an opponents loss, in other words the net change of benefit is zero [2].

One example of another type of strategy game for two players is called Real-time strategy game. Here, one persons game is not necessarily at the cost of the opponent, instead both players can gain power or wealth simultaneously [3].

Two games from different categories require different methods for planning a good strategy and different implementations of AI is needed in order to calculate the winning game plan [1].

2.2 The Minimax Algorithm

The minimax algorithm is a common way to evaluate game states in games like chess, checkers and go [4]. It is however computationally expensive for games with a large number of possible moves or states, since the game tree may grow rapidly. The algorithm has a time complexity of $O(bm)$, where b is the game trees branching factor and m is the maximum depth [5].

The minimax algorithm is a recursive algorithm used in game theory and artificial intelligence to calculate the optimal move for a player in a two-player, zero-sum game. It involves searching through a tree of nodes, where a node is a possible game-board and pieces. The player controlling the foxes wants to minimize the amount of points, the hens want to maximize the points. Therefore the algorithm is named “minimax”. The algorithm assumes that both players will always make the best move for themselves [6].

2.2.1 α-β Pruning

One technique that may be used to speed up the minimax algorithm is α-β pruning, since it allows for some parts of the tree to be ignored and therefore reducing the required computation. It accomplishes this by eliminating branches of the tree if they are guaranteed to result in a worse state than any previously evaluated moves. In order to keep track of previously evaluated moves, it stores the best move that has been found so far for player A and player B in the variables α and β respectively [7].

If the algorithm evaluates a node and finds that its value is worse than the α value, then it means that player A will never choose that node because there is already a better option available. Therefore, the algorithm can prune the rest of the branch that the node belongs to because it is guaranteed to be worse than the α value. Similarly, if the algorithm evaluates a node and finds that its value is better than the β value, then it means that player B will never choose that node because there is already a move that is better for player B. Therefore, the algorithm can prune the rest of the branch that comes after that node, since it will not be played [7].

By pruning these branches, the algorithm can avoid evaluating states that are guaranteed to not be played, and therefore reducing the total number of nodes that needs to be evaluated, improving the algorithm’s efficiency [7].

2.2.2 Variable Search Depth

Varying the search depth may both decrease the computation needed, as well as allowing the algorithm to look further ahead in the tree. By allowing the search depth to be dynamically adjusted based on the current game state, the optimal depth can be chosen, allowing for the most efficient and accurate search.

If the game is in a state that has a lot of possible next states, searching too deep into the tree would require immense computational resources, therefore a shallower search depth may be required to limit the computation time. On the other hand, if the game is in a state where there are only a few possible next states, a deeper search would be computationally possible and would allow the algorithm to give more accurate results.

2.2.3 Parallelizing the Minimax Algorithm

In the context of computer programming, algorithms can be executed either sequentially or in parallel. With the advent of multi-core processors, parallel computing has become increasingly relevant, allowing programs to run faster by processing multiple tasks concurrently [8].

The minimax algorithm is a great candidate for parallelization since it needs to do a lot of calculations to find the optimal move. However, parallelizing an α-β pruned version of the minimax algorithm comes with a lot of challenges since α-β pruning is inherently sequential because the decision to prune is based on the values obtained from previously explored nodes. This means that the values of α and β, which determine the bounds for pruning, are continually updated as each node is evaluated. As a result, the pruning decision at any given node is dependent on the outcomes from preceding nodes, making it difficult to distribute the work among multiple processes [8].

2.3 Fox-Game

2.3.1 The Game

Fox-game is an old scandinavian 2-player board game, where one player plays as the foxes, and the other as the hens.

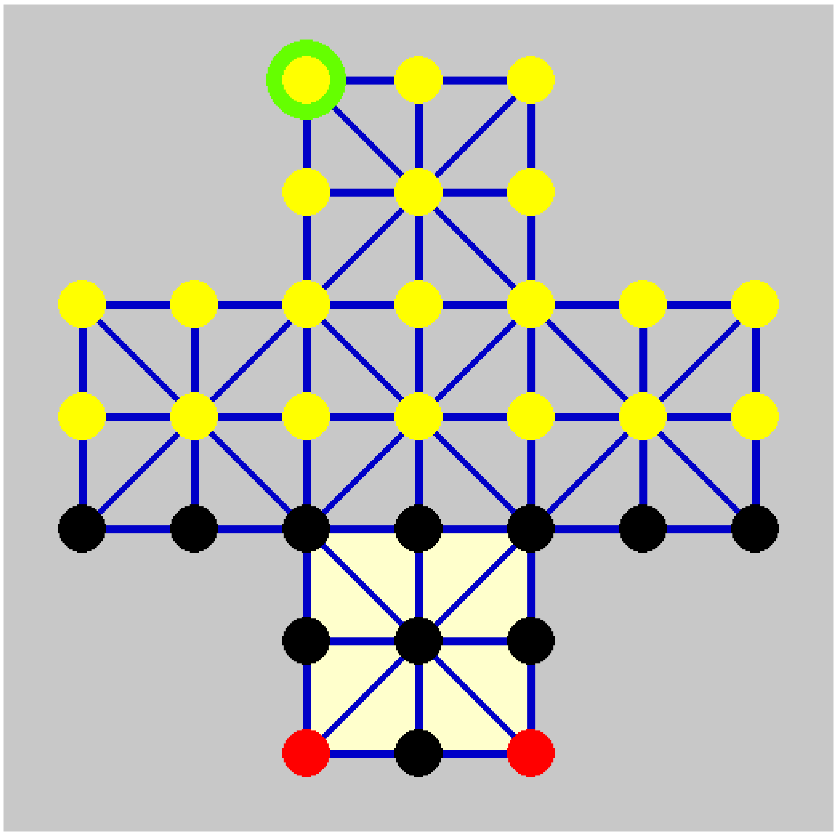

Figure 2.1: The game board

The board layout consists of five three by three squares, where the edge points are connected either vertically or horizontally to its closest edge point neighbor and the middle point is connected to all other points in the square. These squares are then combined to form a plus-sign shaped board layout. One of these squares make up the coop.

There are two fox pieces and 20 hen pieces, and the objective for the hens is to reach their coop on the other side of the board whilst the foxes try to stop them. The foxes start off from the two bottom corners of the coop, whilst the hens start from the opposite side of the board, filling up the top 4 rows. The hens are only allowed to move forward or sideways whilst the foxes may move in any direction.

If a hen is placed in between a fox and an empty slot, the fox may choose to capture that hen by jumping over it. The hens can also capture the foxes by positioning themselves such that the foxes are unable to move.

The hens win if they manage to fill the whole coop or remove both foxes and the foxes win if they manage to deny the hens to do so, meaning only 8 hens remain.

2.4 Related Work

There have been many studies over the years that have analysed the minimax algorithm and different implementations of it.

One study analyzed minimax with and without α-β pruning when applied to the game *Connect-4*. They concluded that at depth 4 their minimax algorithm evaluated 5.9 times more nodes than their α-β pruning variant. At depth 8 their α-β pruning managed to perform even better and performed 81.5 times better than their non-pruning version [9].

Several studies have also researched ways to parallelize the α-β algorithm. One study predicted a speedup of $k^{\frac{1}{2}}$ by having moves ordered from best to worst, as well as having a dynamic tree splitting algorithm, that re-assigns finished processes onto parts of the tree that have not yet been evaluated [10].

Another study, apart from arriving at similar conclusions also highlighted the importance of transposition tables. Meaning that if the same position reoccurs in different places in the tree, they should be given the same heuristic. Implementing this comes with it's own sets of problems, but can yield great rewards in terms of processing power if done correctly [11].

One study implemented the α-β pruning algorithm on chess specifically. One performance boosting method they analyzed particularly used what they called "*scouting*". This method implements a dumber version of heuristics towards the end of the depths analysed causing the algorithm to be faster when searching at these deeper depths, but potentially missing complicated tactical possibilities. Employing these seekers in addition to having their own tree splitting algorithm resulted in a $5.7 - fold$ speedup with 9 processors [12].

Chapter 3

Method

3.1 The Fox-game

All game specific variables are stored in the main game class. Including hens alive, foxes alive, whose turn it is, in addition to the current state of the board. When later creating a node, as described in section 3.2.2, all these variables are cloned and stored in that node.

3.2 Minimax Algorithm

This algorithm employs the foxes as the minimisers and the hens as the maximisers.

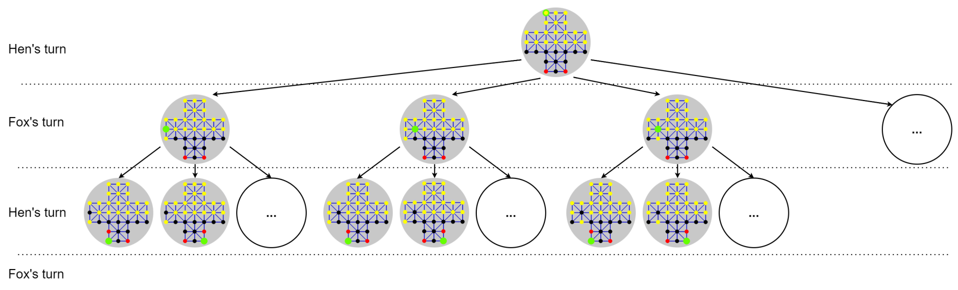

Figure 3.1: Algorithm tree

As seen in fig. 3.1, we use yellow circles to represent hens, red circles to represent foxes and a green circle to represent the selected position on the board.

The algorithm works by recursively calling itself to evaluate each possible move the current player can make until it either reaches a position where one player has won or it has looked at as many steps into the future as it was provided in its depth parameter, where it then will assign the current position a score based on given heuristics, see section 3.2.1. The algorithm then chooses the best move in regards to which side it is controlling, whilst assuming the opponent will play the most optimal moves for them in the upcoming turns.

3.2.1 Heuristics

Each board state is given a score based upon certain heuristics. Since the hens wants to maximize points, each time we award points to hens, we add that number to the total score. Whilst on the other hand if points are awarded to the foxes, said points are subtracted from the total. These heuristics should not affect the results of this report, since we only analyze the performance of the algorithms, not if it plays ”good”. The reasoning to have any type of heuristic for this analysis is for the computer to be able to differentiate between moves.

The most important factor is if someone has won. Winning a game is awarded one thousand points. For hens to win they need to fill up certain spots on the board, we call these positions: *hen's coop* . Three points are awarded for each hen in the coop. Getting hens closer to their coop is also important, therefore one point is awarded for moving a hen towards the coop. Hens are also rewarded for staying closer to the edges.

Both players are rewarded for keeping their own pieces alive and capturing the other player's pieces. One point is awarded for each hen alive and ten points for each fox alive.

Item	Points
Hens alive	1
Foxes alive	-10
Hen in coop	3
Hen on row: R	R
Hen on column C	$\lfloor \lvert 3.5 - C \rvert \rfloor$

3.2.2 Nodes

A node has information about a specific state of the game including the board. Each node also contains a reference to all its own children, where a child is a node based on a game board one valid move away from the current node's game board. See fig. 3.1.

3.2.3 Parallelizing the Minimax Algorithm

As described in section 2.2 the algorithm treats the game as a tree where the nodes are the possible moves a player can make from the previous position, see fig. 3.1. To parallelize the algorithm we use a python library called multiprocessing. This library allows us to start processes that can execute the minimax algorithm in parallel. A new process is started for each child in the first layer of the minimax tree and each process calculates the evaluation for each child. The move that corresponds to the best evaluation is then chosen. Only as many new processes as there are branches from the first node are started in order to ensure that the cost of starting these processes and the inter-process communication does not outweigh the benefits gained from parallelization.

Parallelizing the α-β pruned version of the minimax algorithm is more difficult, see section 2.2.3. Therefore, local α and β values for each process are used, and each process prunes the subtrees independently from each other. This approach works under the premise that the time saved from parallelization outweighs the inefficiency introduced by potentially exploring pruned branches due to the lack of global α and β values.

3.3 Tests

This report conducted multiple tests on the minimax algorithm when used on the Fox game in order to collect data and measure its performance. The tests were conducted by simulating games and performing independent tests on every different game state.

3.3.1 Testing for Computation Required

To gather data of the performance impact of α-β pruning, a single game was simulated. This game consisted of 91 different game states, all of which were tested independently of each other, with depths one to four, with and without α-β pruning. The amount of game nodes required to be computed for each different variant was stored and displayed in fig. 4.1 to fig. 4.4.

3.3.2 Testing for Time Required

The same game as in section 3.3.1 was used to gather time data for each different variant of the minimax algorithm. The total time required for each variant to find the optimal move was measured with Pythons time module [13]. The tested variants were; depths one to four, with and without α-β pruning and with sequential and parallel implementations. The results are displayed in fig. 4.5 to fig. 4.8.

3.3.3 Testing for Optimal Move Selection Rate

To collect data on how the optimal move selection rate is impacted by the depth variable of the minimax algorithm, nine games were simulated, resulting in 888 game states that were tested with depths one to five. From the 888 game states, a mean for each comparison between each depth could be calculated as well as the standard deviation. This data is displayed in table 4.1.

3.4 Hardware

All tests were run on an AMD Ryzen 5 7600X CPU, with 6 cores and 12 threads, boosting to 5.4GHz [14]. Since most of the tests only measure the total amount of computation needed, the hardware used does not matter. However, the test that measured time, section 3.3.2, is highly dependent on the hardware and different results might be found if run on other hardware.

Chapter 4

Results and Analysis

This chapter presents data acquired from tests conducted on the minimax algorithm, highlighting various features of the algorithm. section 4.1 presents data collected from a single game, with 91 game states, highlighting the computation and time required for the minimax algorithm to find the optimal move for different depths. section 4.2 presents data collected from 9 games, with a total of 888 game states, showing the changes in optimal move selection rate between different depths.

4.1 Computation and Time Required

The following data was acquired from a single game, with 91 game states, where the look-ahead played itself with a depth of two. Section 4.1.1 highlights the differences in computation required for the minimax algorithm when using no pruning compared to α-β pruning. Section 4.1.2 presents the differences in computation time for the minimax algorithm when run sequentially and in parallel on a modern consumer CPU, see section 3.4.

4.1.1 α-β Pruning Compared to No Pruning

The following results were taken from the experiment described in section 3.3.1, where all states of a single game were analyzed independently, on how much computation they required to find the optimal move for depths one to four and with or without α-β pruning.

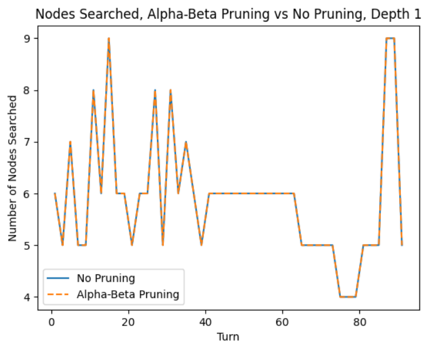

Figure 4.1: Nodes Searched, α-β pruning vs No Pruning, Depth 1.

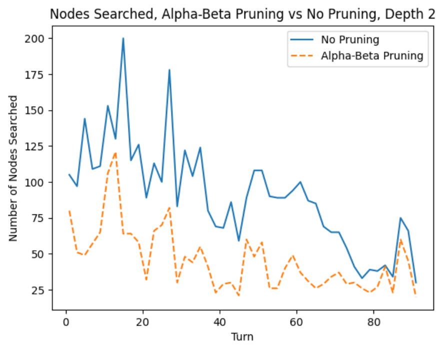

Figure 4.2: Nodes Searched, α-β pruning vs No Pruning, Depth 2.

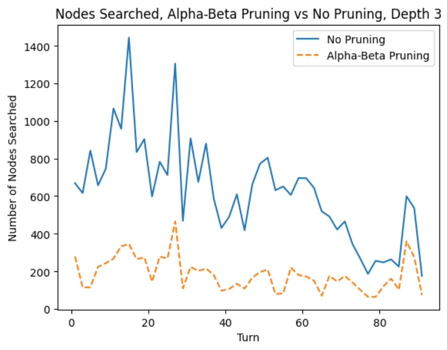

Figure 4.3: Nodes Searched, α-β pruning vs No Pruning, Depth 3.

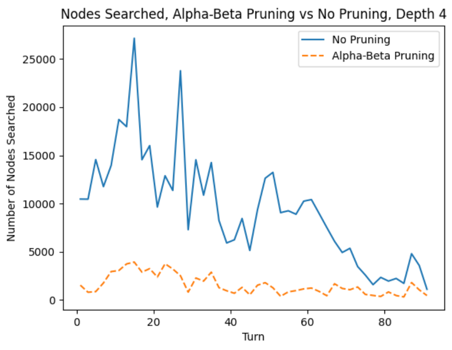

Figure 4.4: Nodes Searched, α-β pruning vs No Pruning, Depth 4.

As seen in fig. 4.1, there is no difference in the amount of nodes needed to be calculated for depth one when using α-β pruning, compared to not using it. This is due to the fact that α-β pruning relies on the comparison of values between two successive depths in the tree to determine whether further exploration of a particular subtree is necessary or not. In a tree with only one depth, there are no deeper levels to compare the values against, which means that α-β pruning cannot be applied effectively in this case.

Looking at fig. 4.2, fig. 4.3 and fig. 4.4, the α-β pruned version of the minimax algorithm requires far less nodes to be visited compared to the non-pruned version. This is to be expected since the purpose of α-β pruning is to remove branches of the game trees that are guaranteed to not result in an optimal move, and therefore reducing the amount of nodes that needs to be calculated. The difference between the two versions becomes greater with the level of depth. This is due to α-β pruning being able to prune of greater parts of the tree since the tree is bigger, and therefore the efficiency of α-β pruning is higher.

4.1.2 Sequential Compared to Parallel

The following results were taken from the experiment described in section 3.3.2, where all states of a single game were analyzed independently, on how much time they required to find the optimal move for depths one to four, with or without α-β pruning and using sequential or parallel computing.

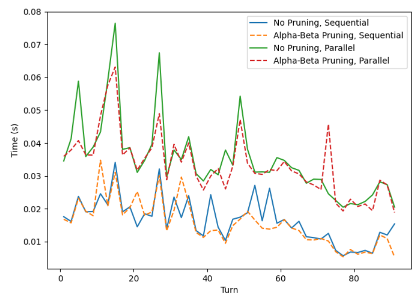

Figure 4.5: Time to find optimal move, Sequential vs Parallel, Depth 1.

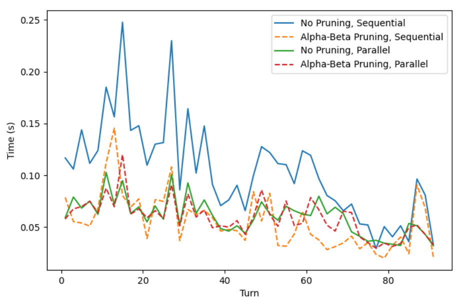

Figure 4.6: Time to find optimal move, Sequential vs Parallel, Depth 2.

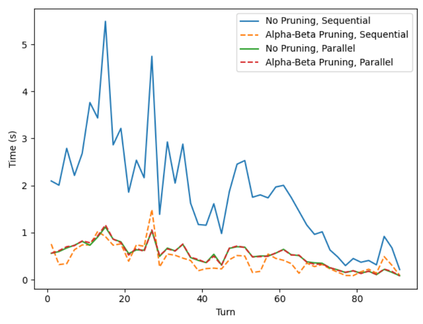

Figure 4.7: Time to find optimal move, Sequential vs Parallel, Depth 3.

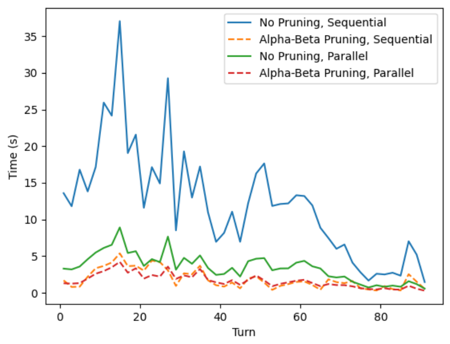

Figure 4.8: Time to find optimal move, Sequential vs Parallel, Depth 4.

Figure 4.5 shows that the parallel versions are slower than the sequential ones, this is not surprising as it is due to the extra overhead required when creating the multithreading environment. In fig. 4.6, fig. 4.7 and fig. 4.8, the two parallel versions perform about the same as the sequential α-β pruned version. Figure 4.8 does however show that the non pruned parallel version is slower then the α-β pruned versions, indicating that for greater depths, α-β pruning becomes efficient enough that more computing power can not outperform it.

4.2 Data From Multiple Games

The following data was collected from nine games, with 888 different game states, where the look-ahead played itself with different depths. Table 4.1 displays the average probability that a certain depth chooses the same next move as a higher depth, for depths one to four compared to depths two to five.

Depth	vs Depth 2	vs Depth 3	vs Depth 4	vs Depth 5
1	0.741(0.438)	0.610(0.488)	0.507(0.500)	0.408(0.491)
2	-	0.635(0.481)	0.502(0.500)	0.421(0.494)
3	-	-	0.592(0.491)	0.516(0.500)
4	-	-	-	0.610(0.488)

Table 4.1: Average probability of depth 1 to 4 choosing the same next move as depth 2 to 5, (standard deviation).

Table 4.1 indicates that a higher depth results in a higher optimal move pick rate. When comparing depth 1 to depth 5, the rate of the same next move is around 40.8%, and for depth 2, it is 42.1%. However, for depth 3, this percentage grows to 51.6%, and for depth 4, the percentage grows to 61.0%. These observations suggest that although the growth is generally modest with each added depth level, the increase becomes more pronounced when the difference in the compared depths is smaller.

Chapter 5

Discussion

5.1 Discussing the Results

5.1.1 Looking at Node Count

Comparing fig. 4.3 and fig. 4.4 shows that adding one more depth to the standard minimax algorithm requires upwards of 20 times more computation. The α-β pruning variant however, only requires about two times more computation for one more depth compared to the standard variant and about 8 times more than the α-β pruned version. This means that using α-β pruning allows the algorithm to search one depth deeper for almost no extra computational cost and even if the algorithm already uses α-β pruning, adding one more depth requires about half of the extra computation required compared to adding one more search depth to non-pruned versions of the minimax algorithm.

α-β pruning was most effective during the early stages of the game, when there are many possible moves to be made. This is clearly shown in fig. 4.4, where the difference is large in the beginning but the two versions almost converge in the last moves. This is due to the fact that α-β pruning is highly effective at reducing the number of nodes that need to be evaluated in the early stages of the game when the game tree is wide, allowing for a more efficient search. As the game progresses and the tree becomes narrower, the standard minimax algorithm naturally needs to explores fewer nodes, while the pruning efficiency of the α-β pruned version decreases. This results in the observed behavior where the node count remains almost constant for the α-β pruning version, while it drastically decreases for the standard version as the game progresses.

The standard version of the minimax algorithm exhibits greater node count fluctuation due to its exhaustive exploration of all possible branches and nodes, which can vary greatly in complexity from move to move. The α-β pruned version on the other hand maintains a more constant node count, as it consistently eliminates unnecessary branches, ensuring a more efficient and stable search pattern throughout the game.

5.1.2 Parallelization

The implementation of parallelization substantially boosted the efficiency of the standard minimax algorithm. As depicted in fig. 4.8, the parallelized variant demonstrated a three to four-fold increase in speed at depth 4 when compared to the sequential version. This significant leap in speed can be attributed to the concurrent evaluation of nodes in the game tree, thereby exploiting the inherent parallelism of the minimax algorithm. The simultaneous exploration of multiple game states resulted in faster decision-making, thus improving the overall performance.

On the contrary, parallelization did not significantly enhance the performance of the α-β pruned variant of the algorithm. As reflected in fig. 4.7 and fig. 4.8, the parallel and sequential versions of the α-β pruned variant demonstrated similar performance. This is primarily because the α-β pruning process is fundamentally sequential. The efficiency of pruning relies on knowledge of prior nodes to decide whether to evaluate a specific branch or not. Due to this sequential nature, the benefits of parallelization are somewhat limited in the context of α-β pruning. Our results are somewhat limited by the fact that we used local α and β values for each thread. Therefor each thread prunes a subtree independently from all other threads, which is more inefficient than using synchronized α and β values. We do however believe that global α and β values would not improve performance significantly since the nature of α-β pruning is still very sequential and thereby limits the potential of parallelism.

While parallelization led to improvements, it also introduced some overhead costs due to the requirement of setting up threads for parallel computation. As demonstrated in fig. 4.5, the parallel versions of the algorithm were slower than the sequential ones, due to the time penalty of setting up threads. This extra time penalty is only notable at lower depths, where the cost of setting up threads outpaces the benefits of parallel processing. This overhead cost becomes negligible at higher depths, as the exponential increase in the number of nodes effectively drowns out the thread setup time. This indicates that the benefits of parallelization tend to significantly outweigh its initial costs in complex scenarios, affirming its practical relevance in enhancing computational efficiency.

5.1.3 Information Gain of Each Added Depth

As seen in table 4.1, the closer the tested depth is to the evaluation depth, the greater the chance of them having the same next move. It also seems to be that a certain depth chooses the same next move as one depth deeper 60% of the time, with the exception for the comparison between depth 1 and depth 2, where it is almost 75%.

Although the data reveals some trends, there is still considerable variability, and we cannot make definitive conclusions on the marginal utility of additional depth levels in the minimax algorithm. The standard deviations of the probabilities (presented in parentheses in the table) indicate there is significant variation in the results. Future research may benefit from considering a wider range of depths and a larger data set, which would provide a more nuanced understanding of the benefits of increased depth in the minimax algorithm.

5.1.4 Analysis of Variable Depth

Variable depth might make sense depending on which variant of the minimax algorithm is used and in what scenario. In table 4.1, we see that the no-pruning variant requires a lot less computation after turn 63, which would make it possible to add another depth level after that point, making the algorithm more accurate in selecting the optimal move whilst maybe not requiring that much more computation than it required in the earlier turns. The α-β pruned version does however not have the same drop off in amount of nodes searched in the later stages of the game and does therefore not have the same headroom to allow for a deeper depth.

Variable depth does only make sense if the computation required falls off enough to make room for adding another level of depth, which in this case only is true for the non-pruned version of the minimax algorithm. And even in this case, it would be better to use α-β pruning and add one depth level for the whole game, instead of using a non-pruned version and only raising the depth in the latter stages.

5.2 How Our Methods Affect the Results

In this thesis we have chosen to separate three aspects of the minimax algorithm, α-β pruning, parallelization along with the impact of different depths and if variable depth is desirable to implement.

There are many other methods to achieve even greater improvements for the algorithm. Some of them are mentioned in section 2.4.

Previous research have ordered the moves, so that the best moves are evaluated first [11]. In one thesis they did not implement this feature on any particular game, instead they simply created a tree of nodes with points, to simulate a game. This meant they could order the moves however they wanted. With how α-β pruning works, this does make sense in theory. However in practise it can get a little trickier. How can one know which move is best? If they knew that, why would they need minimax or any other algorithm to figure out the best move? On the other hand, there are some moves that are inherently good, and in most cases are the best move. In the fox game, capturing hens as foxes are most likely the best move, and moving towards hens are also good. So there are some cases where some moves are probably better. In another paper, they implement this feature in the last couple of analyzed depths for their chess minimax algorithm [12]. Just how effective this method is would be hard to evaluate, as it probably is highly dependent on what game the minimax algorithm is applied to.

Other research managed to achieve greater success with their parallelization of the algorithm. In order to do this they implement a master-follower setup where the master algorithm is in charge of employing and redistributing the other processes when they are idle. In order to do this they also have to split the remainder of the tree to be evaluated [10]–[12]. We did not implement this type of setup due to time restraints.

Another feature is having a transposition table. Depending on the game, the same node may appear in multiple positions throughout the game tree. Storing the evaluations for these nodes in a giant hash table eliminates the need to reevaluate the same position multiple time [11].

There are certainly other aspects as well, aside from features implemented that affect the performance of the minimax algorithm. The way nodes are represented, the programming language used, which game the algorithm is applied to, what hardware is used etc.

There are numerous factors that affect the outcome of the performance. We chose to focus on the three aspects mentioned above. The results gathered from this thesis are therefore impacted by how we implemented them and the methods we used to measured and gathered our data.

5.3 Future Works

Although minimax is an old algorithm and has been studied immensely throughout the years, meaning it can be tricky to find interesting aspects to study that have not already been studied before. There are many exciting avenues to explore for future works.

Our original plan was to implement a self learning AI which would then be tested against our minimax AI, to see which algorithm would be better at the game. This avenue could be expanded where more algorithms could be tested and on different games.

Analyzing and finding better ways to parallelize the algorithm, or putting more effort towards finding the theoretical limit of parallelization of the algorithm might be another path.

Finding ways to implement the algorithm on different types of games and having ways of evaluating it's effectiveness is another approach, games where the algorithm might have been overlooked because it has been seen as difficult to implement or finding real world scenarios that an effective minimax algorithm may be applicable.

Chapter 6

Conclusion

In conclusion, this thesis applied the minimax algorithm with several enhancements to the Fox-game. We utilized α-β pruning to improve efficiency, which yielded up to 20 times better performance, especially at the start of games, when the piece count was high. The impact of parallelization varied, proving beneficial for deeper searches in the non-pruned version but yielded no improvements for the α-β pruned version. We also examined dynamic search depth, which only proved effective for the standard minimax algorithm. Ultimately, α-β pruning was the most impactful enhancement, and future research may refine parallelization and dynamic search depth within the pruned minimax algorithm for further optimization.

References

[1] P. Paolo Ippolitio, “Game theory in artificial intelligence,” *Towards Data Science*, Sep. 2019. [Online]. Available: https://towardsdatascience.com/game-theory-in-artificial-intelligence-57a7937e1b88.

[2] W. Kenton, “Zero-sum game definition in finance, with example,” *investopedia*, Aug. 2022. [Online]. Available: https://www.investopedia.com/terms/z/zero-sumgame.asp.

[3] A. Dahlbom, “An adaptive ai for real-time strategy games,” M.S. thesis, Högskolan i Skövde, 2004. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:his:diva-908.

[4] *Chess engine*, Oct. 2020. [Online]. Available: https://www.chess.com/terms/chess-engine.

[5] D. N. Jeevanandam, “Understanding the minmax algorithm in ai,” *INDAai*, Jan. 2023. [Online]. Available: https://indiaai.gov.in/article/understanding-the-minmax-algorithm-in-ai.

[6] *Algorithms - solving problems by searching*, 2003. [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/blind.html#.

[7] *Algorithms - alpha-beta pruning*, 2003. [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/alphabeta.html.

[8] M. Bio, “A brief history of the multi-core desktop cpu,” *TECHSPOT*, Dec. 2021. [Online]. Available: https://www.techspot.com/article/2363-multi-core-cpu/.

[9] R. Nasa, R. Didwania, S. Maji, and V. Kumar, "Alpha-beta pruning in mini-max algorithm–an optimized approach for a connect-4 game," *Int. Res. J. Eng. Technol*, pp. 1637–1641, 2018.

[10] R. A. Finkel and J. P. Fishburn, "Parallelism in alpha-beta search," *Artificial Intelligence*, vol. 19, no. 1, pp. 89–106, 1982. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0004370282900224.

[11] T. A. Marsland and M. Campbell, "Parallel search of strongly ordered game trees," *ACM Computing Surveys (CSUR)*, vol. 14, no. 4, pp. 533–551, 1982. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/356893.356895.

[12] J. Schaeffer, "Improved parallel alpha-beta search," in *Proceedings of 1986 ACM Fall joint computer conference*, 1986, pp. 519–527.

[13] *Pythontime*, May 2023. [Online]. Available: https://docs.python.org/3/library/time.html.

[14] *Amd ryzen 5 7600x*, Sep. 2022. [Online]. Available: https://www.amd.com/en/products/cpu/amd-ryzen-5-7600x.

TRITA-EECS-EX-2023:286