Variations of Minimax to Solve Connect 4

Zach Siew
siew0053@umn.edu

December 21, 2020

Abstract

Connect 4 is a 2 player game that belongs to the classification of an adversarial namely a zero sum game with perfect information for both players. Connect 4 was first solved by using a knowledge based approach with solutions revolving multiple strategies; however, it was able to be solved using the brute force approach a few years later. Artificial Intelligence algorithms that were capable to strongly solve Connect 4 was minimax and negamax. To add on, both minimax and negamax algorithm can adopt the implementation of alpha-beta pruning to further optimize the run-time of the algorithm. After alpha-beta pruning was first introduced to act as an optimization to minimax and negamax algorithm, different variations of alpha-beta pruning that could outperform alpha-beta pruning in certain cases were gradually being introduced as well. Some of the variations which will be discussed in this paper are Palphabeta and SCOUT algorithm. By comparing these variations to alpha-beta pruning, we will be able to grasp a clearer picture of which algorithm is more efficient in solving Connect 4.

Introduction

A 2 player non-trivial game such as Tic-Tac-Toe, Othello, and Connect 4 has existed for a long time now and almost all of them had been solved by AI with a specific algorithm attached to it. However in the point of view of a normal human, people normally would find it hard to come up with a strategy that will always result in a win against anyone. Normally most 2 player non-trivial game is a perfect-information game, meaning each player has information of all events that occurred previously when making a decision, normally in a sequential game. Right now, there exist a certain tactics that each player can adopt to always win or draw the game. For instance, the first player will almost always will be guranteed a win if their first move were to drop at the middle column if they play the rest of the game perfectly. If the first player did not place their first piece at the middle column, then the second player can force a win by playing perfectly as well. However for this paper, I will not be discussing those static tactics or solutions as those involves zero to none implementation of AI. I will be focusing more on an AI approach to solve Connect 4. With that, we can observe a specific algorithm known as minimax that can be used to solve this type of game given any "board position" (state). In this case, the game which we will be referring to is Connect 4. Another reason on why minimax would be a good fit for using to solve Connect 4 is that it is a zero sum game. Meaning that a player's advantage is equivalent to another player's disadvantage. And minimax is a great algorithm to solve zero sum game. A general description of how Connect 4 works is that, two players each with a different color will take turns dropping colored discs into one of the 7 column until one of the player manage to connect 4 of their respective colored disks. By applying the minimax algorithm to Connect 4, we can easily achieve the best possible action to take at any given point of the game to result in a win for ourselves.

Literature Review / Related Work

When dealing with classic board games such as Connect 4, chess, and so on, they all have something in common which is that they are all two-player deterministic game with perfect information available to both players. And there exist two similar yet different algorithms that are widely used to solve these type of games. The two algorithms are called the minimax algorithm and Monte Carlo Tree Search (MCTS). The main difference between these two algorithms is the way they perform their search. In the case for minimax algorithm, every possible moves are searched from a given state to the terminal state until one is able to determine which move will result in the best possible outcome. On the other hand, MCTS utilises random play-outs to obtain an Upper Confidence Bound (UCB) value and the UCB value associated with every move will determine the next action [4]. Because minimax searches every possible moves, the branching factor of the game tree will be enormous if the game that is being searched on is a complicated one with a wide range of possible actions and outcomes. As most two player games that people are more interested in using AI on are extremely complicated such as chess and Go, MCTS is a much preferable algorithm to use because MCTS does not try to evaluate every moves but rather kind of "guess" the next best move with random playouts. Not only that, MCTS is also more popular and practical than minimax because MCTS does not an explicit evaluation function while minimax does. Most of the time, it is extremely difficult to come up with a good evaluation function for games with high complexity and if the evaluation function does not provide an accurate value, that will cause the whole search to crumble as no good answer will be computed. Hence, MCTS is more practical as one does not need to come out with an evaluation function which can be hard [4]. Since the 2 player game that are of interest to us is Connect 4, it is better for us to use the minimax algorithm as Connect 4 is a fairly simple game and the evaluation function for it can be easily defined and computed as well. Not only that, the maximum possible moves for Connect 4 is the number of columns in the board, which is 7. Hence, we will have a maximum branching factor of 7, which is reasonable to use minimax. The branching factor can also be further reduced with methods such as alpha beta pruning which will be discussed next.

Even though we had settled on using the minimax algorithm to solve Connect 4, there are a lot of variations that the minimax algorithm can take on to solve the same problem. All of them with their respective usage. In this section, I am going to explain the different approaches to the minimax algorithm as well as providing an explanation for the change.

By using the regular minimax algorithm to solve Connect 4, it might pose an issue and will return a move that might not be the correct one. Researches in [7] has shown that the algorithm can result in a situation called as a game tree pathology where inaccurate results will be produced as the depth of the search increases. This is mainly due to the fact that the evaluation function used to calculate how good a state is is an estimate measurement of that respective state rather than the exact measurement. And when this evaluation function is applied to the leaf node of a game tree that has a large depth, the errors from estimating the state a state by the evaluation function will be amplified as they propagate up the tree, resulting in an incorrect move being made [7]. In [1], a slight modification to the regular minimax algorithm has been proposed to eliminate this game tree pathology; that is, the Althofer's negamax algorithm. The algorithm essentially makes use of negamax which is a slight variation to the minimax algorithm where it consists of only max nodes and the negative value of the maximum is carried upwards. In the same paper, it has also been shown that the algorithm is capable of applying alpha beta pruning to improve the runtime of the algorithm by eliminating unnecessary searches. Hence, I will be implementing both the minimax algorithm and the Althofer's negamx algorithm on Connect 4 and perform an analysis on both algorithms as explained in the Analysis section.

Other than encountering with game tree pathology, the minimax algorithm also can encounter with something called the "horizon effect" [2]. In [2], the horizon effect is explained to be a problem which can occur when a game tree is searched to a fixed depth, game losing decision and threats beyond the search depth will remain undetected. So, a state may have a good score from the evaluation function at that given point, but it does not account to the future of that given state. And by the time that the player must deal with the threat, it would be too late as the player's position might become too degraded to be salvageable. To counter that, an algorithm called the Quiescence search is introduced. The way Quiescence search works is just like how human players are able to decide whether to ignore a potentially bad looking move and search a promising move to a great depth. Quiescence search utilises two main parameters which are called "volatile" and "quiet" where "volatile" positions are search to a great depths to make sure there are no threats lying down the game tree to get a better estimate of the evaluation function [5]. The determination of "volatile" and "quiet" position can then be determined by the impact that specific move has on the game such as capturing enemy pieces in a game like chess. However, even though we know that such problem exists, we will not be implementing this algorithm for our Connect 4 game. For the same reason as to why we does not use MCTS, Connect 4 is not one of those highly unstable game like chess where performing a specific move may be beneficial at first but might create a hidden, unnoticeable threat later on. Other than that, it is also hard to come up with the "volatile" and "quiet" position for a game like Connect 4 as there does not exist actions that could greatly impact the game like capturing enemy pieces in chess [5].

While regular minimax algorithm is great in solving 2 player perfect information games and can be further enhanced with the help of alpha-beta pruning as it eliminates unnecessary searches that will only increase the program's run-time without any benefits. Not only that, there exist also other variations to alpha beta pruning that allows the minimax algorithm to potentially compute faster. In the process of optimizing alpha beta, a new concept called minimal window search was introduced as well where it allows one to make cutoffs that an alpha beta full window search was not able to [3]. One of the variations of alpha beta pruning that uses minimal window serach is called the principle-variation alpha beta (Palphabeta). It makes use of minimal window search in a way that each time a minimal window search on a subtree fails high, the search is repeated with a wider window. However, there is still some risk in using this approach because if the tree is poorly ordered, Palphabeta will visit more leaf node than normal alpha beta pruning does [3]. There exist also another variation of alpha beta pruning which is a further generalization of Palphabeta and it is called the SCOUT algorithm. In [8], a rough description of how SCOUT algorithm should be implemented is explained. The only difference that the SCOUT algorithm has while comparing to Palphabeta is that a recursive call is made to the SCOUT function itself and not to alpha beta when the minimal window search fails high. That being said, there will be 3 different approaches which I will implement to solve Connect 4. That is, alpha beta pruning, Palphabeta, and SCOUT algorithm. All of which will be implemented on a negamax algorith instead of minimax algorithm.

Approach To Solve Connect 4

Before I start discussing the approaches that I will take to solve Connect 4, there is somethings that first need to be addressed. For this project, I will be modifying a publicly available code from [6]. The code will be in a public github repository and consists of python coded connect 4 game written with a python module named pygame so that GUI based Connect 4 will be presented, as well as a minimax algorithm to help solve the problem. From there, I will be modifying mainly on the subsection of minimax algorithm provided in the code. The GUI based Connect 4 code will stay the same as its only job is to help visualise the board after every player's action. There will be several ways in which I will take to solve connect 4, as explained in the Literature Review section. First of all, I will be conducting a comparison between the regular minimax algorithm with Althofer's negamax algorithm to investigate if the effect of game tree pathology applies to Connect 4. The only way in which this can be done is by having two bots playing against each other, where one bot is running on the regular minimax algorithm and another running on Althofer's negamax algorithm. An example of this would be running a bot with minimax algorithm in one machine and running another bot with negamax algorithm in another machine and depending on what each bot decides to move, I will carrying their respective moves to their opponents. For instance, if bot 1 decided to drop a piece on column 1, then I will be personally performing a move (pretending to be bot 1) in bot 2's game and vice versa. Since the purpose of this specific experiment is to test the effect of game tree pathology and game tree pathology will normally only occur at a much deeper depth; as a result, I will be conducting this experiment with varying depths. The minimax algorithm that I will be using is taken directly from [6] whereas the negamax algorithm that I will be using is from [3] in which they had provided us the pseudocode for it.

Next, I will be diving deeper into what the regular minimax algorithm can be modified into to be more efficient and faster. One of it would be the implementation of alpha-beta pruning, Hence, I will be adding an extension to the regular minimax algorithm called alpha-beta pruning. Then I will begin comparing the run-time between the regular minimax algorithm and the minimax algorithm with alpha-beta pruning. Specifically what is the time it takes for the bot with their respective algorithm to come up with an action. The run-time that will be noted down will be the average of several actions in order to get a more consistent and accurate comparison. To better illustrate the usefulness of alpha-beta pruning, I will be conducting this experiment with varying depths for the search as well so that a clearer correlation can be represented. Not only that since Connect 4 is a 2 player game, to obtain a more accurate result from the run-time, the player's move that are given to both bots with different algotithms will always be the same. The pseudocode in which I used to modify the regular minimax algorithm is from [3].

Lastly, I will be exploring on the different variations that the concept of alpha-beta pruning can take on. As explained in the Literature Review section, I will be focusing on Palphabeta and SCOUT algorithm. I will be conducting a comparison of run-time between three different algorithms, i.e. alpha-beta pruning, Palphabeta, as well as the SCOUT algorithm. In the Palphabeta algorithm that I am going to implement, the pseudocode in [3] has a falphabeta function in it. Falphabeta functions just like normal alpha-beta pruning except it adopts the concept of fail-soft. Not only that, in the SCOUT algorithm that will be implemented, a new function called TEST will also need to be implemented alongside. The purpose of TEST is just to test if the value of that specific branch is greater than or lesser than a bound v. There are 2 variations to TEST, which is $TEST_{GREATER}()$ for max node and $TEST_{LESSER}()$ for min node. If TEST returns true, then it means there are better values at that specific branch. And if TEST returns false, that means to ignore that specific branch as no better values exist inside it. The pseudocode for TEST can be found in [8]. It is also important to note that these algorithms will be implemented on a negamax algorithm. For this experiment, just like in the case with the second experiment mentioned earlier where I compare the run-time between minimax and minimax with alpha-beta pruning, The same player move will be used across all three algorithms to ensure accuracy. In this experiment, I will specifically focus more on the SCOUT algorithm as the SCOUT algorithm is just a further generalization of Palphabeta. The pseudocode that were used are from [3]. All of the algorithms that were touched on will be provided down below.

Algorithm 1 Althofer's Negamax Algorithm

```
1: procedure NEGAMAX($x, \alpha, \beta$)
2:     $h \leftarrow staticevaluation(x)$
3:     if $xisaleaf$ then
4:         $return(h)$
5:     end if
6:     Generate children $x_1, ..., x_n$ of x
7:     $m \leftarrow a$
8:     for $i := 1$ to n do do
9:         $t \leftarrow -Negamax(x_i, -\beta, -m)$
10:        if $t > m$ then
11:            $m \leftarrow t$
12:        end if
13:        if $m \geq \beta$ then
14:            $return(m)$
15:        end if
16:    end for
17:    $return(m)$
18: end procedure
```

Algorithm 2 Alpha-Beta Pruning

```
procedure ALPHABETA($p, \alpha, \beta$)
    if $terminal(p)$ then
        $return(staticvalue(p))$
    end if
    $w = generate(p)$
    $m = \alpha$
    for $i := 1$ to w do do
        $t = -alphabeta(p_i, -\beta, -m)$
        if $t > m$ then
            $m = t$
        end if
        if $m \geq \beta$ then
            $return(m)$
        end if
    end for
end procedure
```

Algorithm 3 Palphabeta

```
procedure PALPHABETA($p$)
    if $terminal(p)$ then
        $return(staticvalue(p))$
    end if
    $w = generate(p)$
    $m = -palphabeta(p_1)$
    for $i = 2$ to w do do
        $t = -falphabeta(p_i, -m - 1, -m)$
        if $t > m$ then
            $m = -alphabeta(p_1, -\infty, -t)$
        end if
    end for
    $return(m)$
end procedure
```

Algorithm 4 SCOUT

```
procedure SCOUT($p$)
    $w = generate(p)$
    if $b = 0$ then
        $return(staticevaluation(p))$
    else$v = scout(p_1)$
    end if
    if $p$ is max node then
        for $i := 2$ to b do do
            if $TEST_{GREATER}(p_i, v)$ is True then
                $v = scout(p_i)$
            end if
        end for
    end if
    if $p$ is min node then
        for $i := 2$ to b do do
            if $TEST_{LESSER}(p_i, v)$ is True then
                $v = scout(p_i)$
            end if
        end for
    end if
    $return(v)$
end procedure
```

Analysis of Result

This section will consists of tables that corresponds to their respective experiments and an analysis of those tables will be provided.

Table 1: Regular Minimax Algorithm (Player 1) vs Althofer's Negamax Algorithm (Player 2)

	Game 1	Game 2	Game 3
Depth 1	Player 1 win	Player 2 win	Player 1 win
Depth 2	Player 2 win	Player 1 win	Player 1 win
Depth 3	Player 2 win	Player 1 win	Player 2 win
Depth 4	Player 1 win	Player 1 win	Player 1 win
Depth 5	Player 1 win	Player 1 win	Player 1 win
Depth 6	Player 1 win	Player 1 win	Player 1 win

The purpose of this experiment is to test if game tree pathology applies to the specific algorithm that is being used to solve Connect 4. If game tree pathology applies to that algorithm, then the player that is using that algorithm will ultimately lose the game as the algorithm is not returning the correct solution. For this experiment, player 1 will be the player that moves first while player 2 will be the player that goes second. Based on table 1, we can see that no correlation can be found for depth 1 to depth 3 in regards to which algorithm is more dominant. The player that wins the game seems to be random. This is most likely because of the depth in which they are running on. Since both algorithms are running on a small depth, the solutions generated by both algorithms will not be a good one. With that, both algorithms from depth 1 to depth 3 are playing poorly against each other, which explains the randomness of victory. However, starting from depth 4 on wards, player 1 attains victory for every game. This is highly due to the fact that a nearly perfect solution for Connect 4 can be produced by both algorithms with a fairly small depth. This is because Connect 4 is not a complicated 2 player game like chess and GO. The game tree produced to solve Connect

4 is very small comparing to game trees produced for chess and GO. Hence, an optimal move will most likely exist starting from depth 4 on wards. For example, if there is an enemy piece on the board, the algorithm with depth less than 3 will not be able to search for the case when the enemy win by connecting 4 pieces together. However with depth of more than 3, these cases can be easily predicted and prevented to secure victory. Going back to the experiment, since we know that player 1 starting from depth 4 is playing near perfectly for every single moves, player can always force a victory that way. The only way that player 1 with regular minimax algorithm would lose is if it is being affected by game tree pathology. However, such a case cannot be observed in this experiment. A possible explanation for this is due to the fact that the effects of game tree pathology only kicks in when a game tree is extremely large with an evaluation function that is not perfect. Since the game tree produced to solve Connect 4 is fairly small compare to other 2 player games and the evaluation function to evaluate the state of a Connect 4 game is very straightforward and easy, it is safe to say that game tree pathology does not apply to algorithms that are used to solve Connect 4.

Table 2: Regular Minimax vs Minimax with Alpha-Beta Pruning

	Regular Minimax Run-time (s)	Minimax with Alpha-Beta Pruning Run-time (s)
Depth 1	0.003	0.003
Depth 2	0.017	0.009
Depth 3	0.128	0.051
Depth 4	0.886	0.150
Depth 5	6.706	0.832
Depth 6	48.412	2.620

Based on table 2, it can be easily observed that minimax with alpha-beta pruning is has a faster run-time than regular minimax algorithm at every depths except for depth 1. There exist a special case such as with depth 1 where the run-time for both algorithms are the same is because of how alpha-beta pruning works. Alpha-beta pruning works by eliminating unnecessary searches by comparing the existing value with the min/max value that were found by searching down the game tree. In order to accomplish that, one will need to keep track of bestMin and bestMax value. If the bestMax value is more than the bestMin value, then one can prune that part of the sub-tree, meaning to stop searching down that path. In the case of depth 1, there will hardly be any sub-tree for us to prune since the value that we found by searching down the tree will be automatically moved up to the root node to compare. In other words, the concept of finding the bestMax and bestMin value for comparing does not apply to depth 1 since the value at depth 1 can straight away be used to compare with the value at the root node; hence, alpha-beta pruning is unnecessary here. Disregarding depth 1, there is a general pattern of minimax with alpha-beta pruning having a faster run-time than regular minimax. This is fairly obvious as we already know that alpha-beta pruning allows one to avoid searching redundant paths in the game tree. The use of alpha-beta pruning starts to become apparent at depth 4 onward. Starting from depth 4, the difference between both algorithms starts to grow by a large amount. This is because without the help of alpha-beta pruning, the regular minimax algorithm will need to search the whole game tree which the run-time will grow exponentially as the depth increases. However, with the implementation of alpha-beta pruning, a huge part of the sub-tree can potentially get removed hence reduce the run-time. There exist also some other cases where alpha-beta pruning will search the same amount of nodes as regular minimax even though it does not appear in our experiment. That will happen when we search from the left and the best move occurs at the right most of the game tree. Such a phenomena is called the worst ordering.

Table 3: Alpha-Beta Pruning vs Palphabeta vs SCOUT

	Alpha-Beta Pruning Run-time (s)	Palphabeta Run-time (s)	SCOUT Run-time (s)
Depth 1	0.003	0.003	0.003
Depth 2	0.009	0.019	0.021
Depth 3	0.051	0.077	0.073
Depth 4	0.150	0.137	0.149
Depth 5	0.832	0.492	0.682
Depth 6	2.620	1.542	1.792

Based on table 3, it can be observed that Palphabeta and SCOUT generally are faster than alpha-beta pruning except for some cases, for example depth 2 and depth 3. In the case of depth 2, Palaphabeta took 0.010 seconds more to take an action while SCOUT took 0.012 seconds more than alpha-beta pruning. A possible explanation for Palphabeta and SCOUT to be slower than both alpha-beta pruning and SCOUT is that the tree built for the search was poorly ordered. A poorly ordered tree will cause most minimal window search performed by both algorithms on a sub-tree to fail high meaning it returns a value that is larger than or equal to its upper bound β. Resulting in repeated search with a wider window. And with a wider window, it is highly likely that Palphabeta and SCOUT will end up searching more terminal nodes than alpha-beta pruning. Not only that at depth 3, the results had shown that Palphabeta and SCOUT are slower than alpha-beta pruning as well. Just as the same reason as before, both Palphabeta and SCOUT is most likely repeating their minimal window search on failure high. And since the depth of the tree is fairly small, alpha-beta pruning will almost always outperform both Palphabeta and SCOUT. However, as the depth of the tree starts increasing, alpha-beta pruning will start to run slower than the other 2 algorithms. For instance starting from depth 4, Palphabeta and SCOUT are slowly taking over alpha-beta in terms of run-time. This is because of the idea of using minimal window search starts to become more apparent as the depth increases. With the concept of minimal window search, a tighter bound can be achieved and that can eliminate more unnecessary searches than the full window search that alpha-beta pruning adopts [3].

Other than adopting the idea of minimal window search, another reason that SCOUT is faster than alpha-beta pruning is that SCOUT may visit lesser nodes than alpha-beta pruning. As can be seen from figure 1 below, assuming that $TEST_{GREATER}(P, 5)$ is called by the root after the first branch, it will then proceeds to call $TEST_{GREATER}(K, 5)$ which ship K's second branch. Then as a result, $TEST_{GREATER}(P, 5)$ will fail and SCOUT would not need to be called for the branch rooted P. On the other hand with alpha-beta pruning, every node will still need to be searched on. Another observation that could be made is that Palphabeta is generally faster than SCOUT. This is mainly because after a search fails high, Palphabeta examines all terminal nodes of the sub-tree once more while SCOUT performs twice more searches when a node is better than its earlier siblings [3].

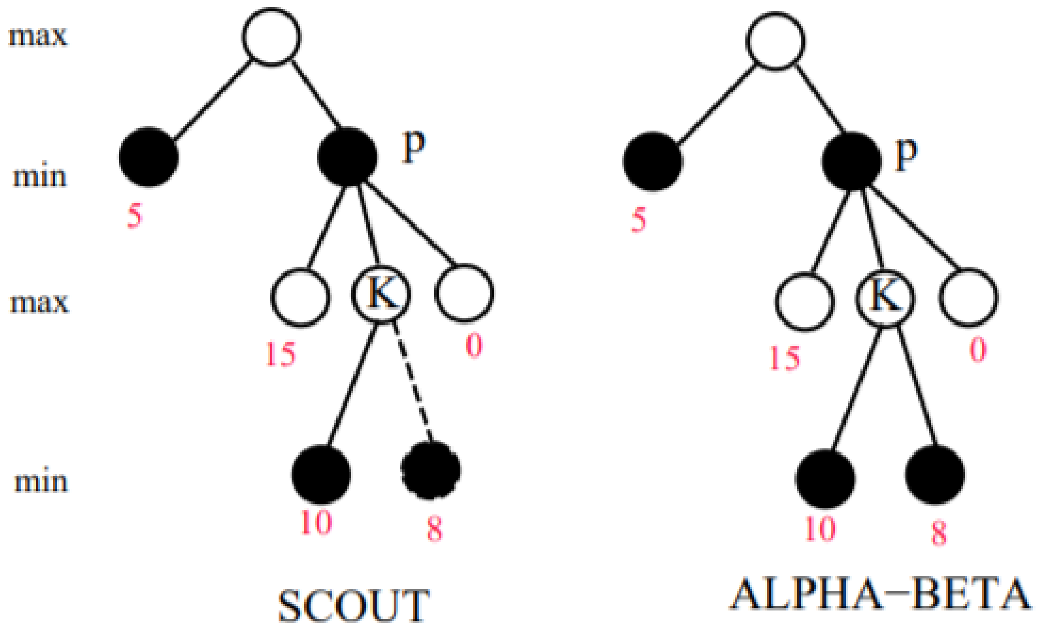

Figure 1: Case when SCOUT visit lesser node than alpha-beta pruning

Conclusion

Connect 4 is a fairly simple 2 player game which is also one of the reason on why game tree pathology does not work on the algorithms that are used to solve Connect 4, specifically minimax algorithm. The ability for a minimax algorithm to obtain an optimal solution with a relatively small depth is what makes it immune to the pathology. Even though the regular minimax algorithm with no modification is enough to come up with an optimal solution, the time required to obtain such solution is very long as the depth increases. Which is why a popular modification to minimax algorithm is introduced. That is, the alpha-beta pruning. The job of alpha-beta pruning is to prune the part of sub-tree that are deemed useless early on so that there will not be any unnecessary searches. As long as the game tree produced is not in a worst ordering, alpha-beta pruning will surely be able to reduce the run-time by a large amount. With that, more variations of alpha-beta pruning such as Palphabeta and SCOUT are also introduced to enhance the original alpha-beta pruning. The idea of minimal window search also came along and is being implemented in both Palphabeta and SCOUT. As alpha-beta pruning uses the concept of full-window search while Palphabeta and SCOUT uses minimal window search, Palphabeta and SCOUT will ultimately outperform alpha-beta pruning as they are searching with a tighter bound. Other than Palphabeta and SCOUT, there exist also other algorithms that might potentially outperform both of them such as fail-soft alpha beta (falphabeta) and SSS* algorithm that are not discussed in this paper. With these algorithms being as efficient as they are already, one way to further improve the performance of such algorithms is the creation of the game tree. If the respective algorithm is able to produce an ideal ordering of a game tree and prevent a worst ordering to ever happen, then the performance of such algorithm will be greatly increased and be consistent as well.

References

[1] M. A. Abdelbar. Alpha-beta pruning and althofer's pathology-free negamax algorithm. 5(4):521–528, 2012.

[2] H. J. Berliner. Some necessary conditions for a master chess program. In N. J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 77–85. William Kaufmann, 1973.

[3] M. Campbell and T. A. Marsland. A comparison of minimax tree search algorithms. *Artificial Intelligence*, 20:347–367, 07 1983.

[4] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai.

[5] L. Harris. *The Heuristic Search and The Game of Chess. A Study of Quiescence, Sacrifices, and Plan Oriented Play*, pages 136–142. Springer New York, New York, NY, 1988.

[6] G. Keith. Connect-4 python. `https://github.com/KeithGalli/Connect4-Python`, 2019.

[7] L. Li and T. A. Marsland. On minimax game tree search pathology and node-value dependence. pages 1–26, 1990.

[8] J. Pearl. Scout. a simple game-searching algorithm with proven optimal properties.