A Comparative Study of OLETS and other Minimax Algorithms in Standard American Checkers

Authored by Stefanie Tischner

***Abstract*—Zero sum board games are a simple and effective way to test the effectiveness of tree search algorithms. Although their structure is quite plain, that straightforwardness lends itself to testing these algorithms at their most fundamental level. This paper will measure Upper Confidence bound applied to Trees (UCT) and Open-Loop Expectimax Tree Search (OLETS) against each other in the realm of Simplified American Checkers. With both algorithms rooted in MINMAX and being derivatives of MCTS, they share many commonalities. Although UCT is considered a pretty standard version of MCTS, OLETS has yet to become mainstream. This paper seeks to determine how OLETS performs compared to UCT in a variety of tests and depth searches. Both algorithms - along with a third to be used as a control - will be built using the Unity game engine and then measured against each other in a series of controlled environments. The tests examine how well each algorithm arrives at a 'correct' answer with depth acting as a sliding parameter. The winning algorithm is the one that requires least computation time for the most correct answers. Unfortunately, after extensive testing the results showed a very minimal difference between the 2 algorithms. UCT tended to perform slightly better overall, but not at a high enough margin to make the difference significant numerically.**

I. Introduction

ZERO-sum board games have been a popular topic in artificial intelligence since its conception[1].They revolve around the concept of equal exchange, where one party's loss is another's win. Creating algorithms capable of solving these games has been a long-standing topic in the field. For checkers, the journey ended in 2007 with a solution that tied in a move database with a backward and forward search algorithm[2]. Over the years, this solution has become more sophisticated and solvable with a singular forward search algorithm[3]. This paper is conducting a comparative study of two such algorithms: Open-loop Expectimax Tree Search (OLETS)[4], and Upper Confidence bound applied to Trees (UCT)[5]. The 2 algorithms share a lot of similarities, from the algorithms that inspired them to the methods they utilize to calculate their best move. With OLETS being the newer of the 2 algorithms by about 8 years, it has yet to be extensively tested in the same contexts that UCT is often employed in. The research that follows attempts to rectify that by measuring them against each other by performing various tests to determine which algorithm is the most cost efficient in the process of reaching the 'right' answer. In order to conduct this research appropriately, MINIMAX[6], a simple deep search algorithm, was implemented as a control algorithm to give the experiment some perspective.

II. Related work

This literature review will first review how checkers initially solved, after the discussion will focus on the history, functionality, and intention behind each algorithm used in this study.

A. How checkers was solved

Checkers, or draughts, was a long-standing work in progress in solvable board games. Before a solution was found, computers already had the ability to beat human players for several years. Initially, this was thanks to Samuel's, which used a rudimentary version of Minimax [7]. However, it was achieved more effectively a few years later by CHINOOK, a program containing an opening database [8], a deep search algorithm, and every possible endgame position in the game [9]. The project was first started in 1989 as a way to further understanding of heuristic search algorithms. The program starts with opening a library containing a vast collection of opening moves by grandmasters. Once the program has found an appropriate move set and navigates its way out of the opening phase of the game, it then performs a deep search using alpha-beta search with the history heuristic [8]. The program's strength lies in its extensive database of end games that it can pull and derive strategy from. While initially starting with only all 6-piece boards [8], it was eventually able to expand to hold all 8-piece solutions [2]. The database holds all endgame positions as well as whether they are likely to result in a win, loss or a draw. With the use of a backward search, CHINOOK can accurately manoeuvrer its way to a favourable ending.

Then, in 2007, a solution was found. It came in three parts: An endgame database, a proof tree manager, and a proof solver [2]. The database was adapted from CHINOOK and was boasting an impressive 39 trillion different ending positions, improving the 8 piece database to a 10 piece one. As the game commences, the forward search algorithm receives the game's current state and then proofs it against the endgame positions. If no matches are found, the program then conducts a minimax search while using CHINOOK's move database to evaluate the heuristics of every possible move [2].

B. Minimax

Minimax is a tree search algorithm that assigns itself two players. Max, whose goal is to maximise her outcome, and Min, whose goal is to minimise Max's outcome [11]. It is

```
MinMax (GamePosition game) {
 return MaxMove (game);
}

MaxMove (GamePosition game) {
 if (GameEnded(game)) {
  return EvalGameState(game);
 }
 else {
  best_move < - {};
  moves <- GenerateMoves(game);
  ForEach moves {
    move <- MinMove(ApplyMove(game));
    if (Value(move) > Value(best_move)) {
      best_move < - move;
    }
  }
  return best_move;
 }
}

MinMove (GamePosition game) {
 best_move <- {};
 moves <- GenerateMoves(game);
 ForEach moves {
   move <- MaxMove(ApplyMove(game));
   if (Value(move) > Value(best_move)) {
     best_move < - move;
   }
 }

 return best_move;
}
```

Fig. 1: Minimax Pseudocode, from [10]

important to preface that Minimax as an algorithm is defined by its heuristic analysis. It is characterised by the method is uses to determine its next move. The algorithm will perform a deep search to a desired depth and calculate the heuristic value of each end node. The depth of the search is determined by the user or by how many moves are required until the game's end. The algorithm will then perform a back search from the assumed breadth and alternate the nodes between min and max, each evaluating the node and then returning the minimised or maximised result. This is repeated until the algorithm analyses every possible option upon which a move is decided, and the computer advances its piece (figure 1) [6].

Minimax has since proved to be a staple in the world of zero-sum games [5]. It has been improved and iterated upon several times, often losing its deep search but retaining the core of its decision making. Some popular examples include Negamax [11] and Monte Carlo Tree Search (MCTS) [12].

C. MCTS and UCT

MCTS is a variation on Minimax in which, instead of proceeding with a standard deep search with alpha-beta pruning, the algorithm will perform several rollouts. It will randomly select a node, and then randomly pick the next node from the selected nodes children. This repeats until that simulated game reaches its end, upon which the results from the game get fed up the ladder and the algorithm starts a new game. MCTS will simulate several games to their very end and then calculate the best result based on which node yielded the most wins. The flaw in MCTS oftem comes from the concept of exploration vs exploitation [5], where the algorithm will often start exploiting a solution with a lack of knowledge about what the other branches have to offer.

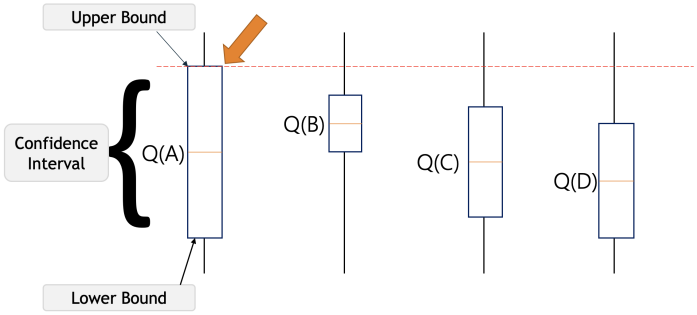

Fig. 2: Visual aid for Upper Confidence Bound, from [13]

UCT tackles this problem with an approach formed from the multi-armed bandit problem[5]. The multi-armed bandit problem is often presented through a gambler standing before a row of slots at a casino. He's bound to randomly pick options until a machine stands out as a 'winner'. If he decides on this winning machine early, he will then exploit it without bothering to explore his other options. Of course, chance always stands that he *did* discover the best machine, but statistically it is much more likely that he'd find a more optimal one if he expanded his search[14].

UCT employs an almost identical search method as MCTS, except with two key differences: Its heuristics and Exploration. UCT never stops exploring. It will continue running random simulations throughout the exploitation phase, favouring nodes that have gone completely unexplored over well performing ones. Marking the explored nodes and randomly picking between the unexplored ones within the given time frame. When done with its search, it will pick the best performing node based on how well each of its children performed. This means that hypothetically, if given enough time, UCT could eventually turn into MiniMax since it would exhaust all its options and return after having completed a deep search of the whole game.

Along with using the minimax method and evaluating the value through the board-given method, it scores each node by how many times it has been visited and if that node results in a win [5]. Out of the explored paths, it will pick the one with the highest reward outcome, as well as choosing to prioritise reward rather than minimising damage(figure 2) (figure 3). This method of calculating the heuristic value of a node through the context provided by the game and its performance in the algorithm established UCT as a popular jumping-off point for the development of similarly proposed algorithms [4] [15].

```
function UCTSearch(node)
{
        make move list out of node

        function rollout
        {
                while (moves < depth)
                {
                        random move(move list)

                        if (result)
                        {
                                return result
                        }
                }

                player 1 = result board.evaluate score
                player 2 = result board.evaluate score

                if player 1 > 2
                {
                        return player 1
                }
                else if player 1 < 2
                {
                        return player 2
                }
                else
                {
                        return draw
                }
        }
}

function compare
{
        new score = UCT.winner.evaluate score
        old score = old best performing branch
        if (new score > old score)
        {
                new score = old score
        }
}
```

Fig. 3: UCT Pseudocode, utalising systems from [16][3]

Algorithm 1 OLETS

procedure OLETS(s, T)
  $\mathcal{T} \leftarrow$root ▷ initialize the tree
  while elapsed time $< T$ **do**
    RUNSIMULATION($\mathcal{T}$, s)
  return action $= \arg\max_{a \in C(root)} n_s(a)$

procedure RUNSIMULATION($\mathcal{T}$, s_0)
  $s \leftarrow s_0$ ▷ set initial state
  $n \leftarrow$root($\mathcal{T}$) ▷ start by pointing at the root
  $Exit \leftarrow$False
  while $\neg Final(s) \wedge \neg Exit$ **do** ▷ Navigating the tree
    if n has unexplored actions **then**
      $a \leftarrow$Random unexplored action
      $s \leftarrow$ForwardModel(s,a)
      $n \leftarrow$NewNode(a,Score(s))
      $Exit \leftarrow$True
    else ▷ use node scoring to select a branch
      $a \leftarrow \arg\max_{a \in C(n)}$OLE($n$,$a$)
      $n \leftarrow a$
      $s \leftarrow$ForwardModel(s,a)
  $n_e(n) \leftarrow n_e(n) + 1$
  $R_e(n) \leftarrow R_e(n) + Score(s)$
  while $\neg P(n) = \emptyset$ **do** ▷ update the tree
    $n_s(n) \leftarrow n_s(n) + 1$
    $r_M(n) \leftarrow \frac{R_e(n)}{n_s(n)} + \frac{(1-n_e(n))}{n_s(n)} \max_{c \in C(n)} r_M(c)$
    $n \leftarrow P(n)$

procedure OLE(n,a)
  return score $= r_M(a) + \sqrt{\frac{ln(n_s(n))}{n_s(a)}}$

Fig. 4: OLETS Pseudocode, from [4]

D. OLETS

Most modern algorithms derived from Minimax tend to perform some random search in their decision-making process. In OLETS, this is done by simulating plays in a static world space which results in the algorithm being significantly lighter than its counterparts.[4]. OLETS iterates randomly through all unexplored nodes during its given time, updating each nodes score along the way. Instead of saving a certain pathway to follow once the computation is over, the root nodes are updated with a score based on the performance of their children, which is then used to determine which path to explore (figure 4). It's method was derived from HOLOP, another algorithm meant to address the bandit problem MCTS was facing [17]. OLETS also forgoes the UCT scoring system that exists in most Minimax derivatives. Instead, it presents a system where it calculates a score based on the average reward of that node and the average reward of all its direct children [4].

E. Implications

UCT is a common standard in the world of AI for zero-sum games; numerous projects and papers exist discussing the topic and outlining the methodology and how to execute the algorithm for the game of checkers [18][16], as well as several publicly available projects implementing the algorithm for their own version of the game.. OLETS, on the other hand, has seen very little application outside of GVGAI competitions [4] [15].

OLETS directly contrasts UCT in several fundamental ways. They are both ideas founded in MCTS and the bandit problem, and they employ their own heuristical system tied to the nodes, but they approach this topic differently. The features that ultimately led to the success of UCT are being directly challenged by what OLETS proposes. They possess very similar approaches to the tree search aspect as well as the Heuristics. However, their methods do still differ in several distinct places. OLETS's most attractive quality lies in its computational simplicity. It was designed to be a lightweight algorithm that still effectively manages to solve problems it presented with [4][15]. However, these are isolated incidents in an evironment that actively enocourages the testing of new algorithms as well as the rejection of the old and safe. With UCT being 8 years its senior, OLETS hasn't been granted the same level of attention or research that UCT has amassed. This dissertation mainly aims to draw a comparison between the 2 in order to determine if OLETS is a viable

option outside of the niche its built. By using Minimax as the control group, this paper intends to directly measure it up against UCT and put its effectiveness to the test.

III. Research Question

RQ: How does OLETS compare against UCT for correctness over computation time in calculating moves for American Standardised Checkers?

H1: UCT will yeild better results while requiring less computation time.
H2: OLETS will yield better results while requiring less computation time.
HNULL: There is no significant difference between the 2 algorithms when it comes to correctness over computation time.

IV. Artefact Design

The artefact used to conduct this research exists in 2 main parts: The game and the algorithms. Also for the simplicities sake, white will always be playing the max player. This is partially to allow for simpler code but largely due to the fact that white will always have the advantage due to the first move. Black will always be retaliating to what white did which makes its position as the permanent mini player guaranteed.

A. The Game

The creation of the base game for the project was pretty straight forward. Having been created using the Unity game engine, the base of the artefact would be supported thanks to the nature of the software. The 2 main components that make up the base are the Board and the Pieces.

The board exists in 2 ways: as a visual component in the game, and as a 2D array. The visual component is quite straight forward. It's a sprite of a checker board that's been equipped with a 2D collider. The collider registers whenever the board is being directly clicked and code in the attached component calculates the clicks location in world space and then converts it into 'game space'. This tells the board what cell was clicked in order to move the piece to the desired location. The visual component however, is primarily there to aid the player in comprehending the state of the game. Most of the logic involves the 2D array of the board which is an 8x8 array used to represent the state of the game. Every 'cell' in the 2D array represents a square on the board and either holds a reference to a piece on it, or is null. This 2D array is also what gets passed on to all the algorithms to give them an understanding of the state of the game, as well as what decides is displayed visually on the screen. Several functions work together to translate the information provided by the 2D array to accordingly adjust the visual representation of the game.

The pieces also exist in 2 parts, the visual and the component. Just like the board, the pieces are equipped with a sprite corresponding to their state and a collider to be manually moved by the player. Their components contain their position within the board, as well as the logic to determine whether a move is illegal or not. Other functions housed within the piece are the ones used to determine whether a piece is a promoted and to check for any forced moves it might be forced to make.

The amalgamation of these 2 components as well as a game manager used to house communication between them results in a playable version of standard American checkers where both sides are controlled by the player.

B. Minimax

The minimax algorithm is quite simple and straight forward compared to the other 2. The main function returns a key value pair containing the evaluation of a board and the board state that led to it. The result is calculated using simple recursion, where a list is made of all possible moves a side could make which then runs the main loop using the child as the root board with one less depth and expanding the nodes on the opposite side. Once the depth reaches 0 the algorithm will return whatever node turned out most favourable for the playing side.

C. UCT

The UCT script works by turning the receiving board into a node class to keep track of the following variables: The parent node of the current one, a list of all its children, the 2D array containing all the board information, what colour its playing, how many times it's been visited, and how many wins the node or its children have been a part of. The main loop expands a node into its children and will randomly choose one but prioritise nodes that have fewer visits. That node gets expanded as well and a new random child is chosen. This loop continues onward until no more moves are able to be made on that simulation and a winner is decided based on the state of the board. This information gets back propagated through all the parent nodes as they get updated with the information of their potential. This process then gets repeated with a different node, until the computing time is up. The algorithm then sends the winning board to be applied to the game.

D. OLETS

The logic loop for OLETS its almost identical to UCT's for a few key exceptions: the data isn't being stored in any semi permanent fashion which allows it to be lighter and slightly more computing efficient. The other main difference is the equation used to determine how likely a node is to be visited again. The equation can be seen in figure 4.

V. Research Methodology

The research is conducted by recreating the game of checkers in unity and then altering the rules to mimic those of American standardised checkers. The game also contains the code base for all 3 of the algorithms which was written to adhere to the details described above.

I will evaluate the algorithms using four criteria derived from 'A Comparative Study of Game Tree Searching Methods' [10]. The algorithms will be measured on **completeness**, **complexity**, **maximum complexity**, and **optimality**. These parameters should be able to provide an apt profile on each of the games completed, allowing for further study and quantification of performance.

Completeness: if the algorithm finds a solution in the case where one exists [10].

Complexity: the number of nodes generated [10].

Maximum Complexity: the maximum number of nodes in memory during the search [10].

Optimality: whether the algorithm always finds the best solution [10].

The algorithms will be tested in 2 main stages and 2 sub-stages: problem-solving and a series of match-ups between algorithms, both of which will be tested on a variety of performance levels.

In the problem-solving stage, the algorithms will be presented with several boards set up with different problems they are expected to solve. These problems are all expected to be solvable within a certain number of moves [19]. The algorithm will be tested on their **completeness** and **optimality** in these tests.

The second test will be sets of pitting the algorithms against each other. Since in a perfect game of checkers, the game will end at a draw, or alternatively, white will win due to the starting advantage; the algorithms will be presented with an already played board, where one side will have an advantage over the other. The sides will be switched for a second test to evaluate the algorithms against each other on **optimality**, **maximum complexity**, **completeness**, and **complexity**.

VI. Artefact Expectations

The artefact itself presents a game of Standard American checkers. The board is 8 by 8 squares large, and each side has 3 rows of 4 pieces. The rules are forced capture, no flying kings, and allowed double jump. Taking the rules into consideration, as well as the planned tests, these are the requirements expected from the artefact:

Loads the board and the Pieces correctly; constantly updating the visuals to give the player a clear understanding on the current state of the board

Implement the rules accurately and accordingly.

Implement all of the 3 AIs; capable of being easily attached to either player.

Working user interaction and the ability to play against any AI on any colour.
Save and Load boards into the game.

Export statistics at the end of every game detailing how the algorithms performed.

The board's data gets stored in 2D arrays, each piece in a spot that coordinates with their in-game position. These arrays then get passed down to each of the algorithms, where copies upon copies are made, each depicting a slightly different board with a different move set displayed on them. Once the respective algorithms have evaluated their board and returned the one with the highest result, the array gets passed on to the original board where it adapts the pieces as needed.

The game also employs a heuristic evaluation of every piece's position: points earned, points lost, number of friendly evolved pieces, and the amount of enemy evolved pieces . Certain board positions and formations will also be rewarded. The central 4 tiles are the strongest but also very prone to attacks, which is why a defensive position is highly favoured as well. The algorithms will also gain extra points for any adjacent friendly pieces since it makes them more protected (figure 5).

```
board eval pseudo code

int boardEval(color)

        int result

        for each tile in centre
                if occupied by color(mine)
                result += 3

        if piece(color) in behind adjacent tile
                result += 1

        for each piece(color) in array
                if exists += 1
                if king += 2

        for each piece(!color) in array
                if king -= 2

        remaining = 12 - count piece(!color)
        result += remaining
```

Fig. 5: Evaluation Pseudocode

VII. Ethical Consideration

The project is a comparative study on different tree search algorithms in the game of checkers. There is no involvement of human participants as well as no collection of any data pertaining to people outside of the artefact. This project exists solely in the artefact hence it will have no impact on the

environment or on society in any way. Due to these criteria this project is low risk. All software used and created are publicly available and used in accordance with the provided licenses. The research conducted and the work produced falls in line with academic integrity with all external work being properly marked and credited. There are no possible ethical risks that come to mind in terms of this project.

VIII. Validity and Verification

The majority of the project relied on the creation of a solid frame work to begin with, which led me to decide that a waterfall workflow would be best suited for this assignment. The waterfall workflow can easily lead to feature creep and instances of redundant or repetitive code. Being acute][ly aware of this I was very careful to plan out most sections of the code beforehand as well as making the vast majority adaptable.

Although little refactoring happened during the creation process, the quality of the work was ensured using Unit testing. Due to a large part of this project being the creation of a working game, almost every aspect of it can be proofed using unit tests. Snippets of testing will be included in the addendum .

IX. Problem Solving - Data Collection

The resulting data was collected by creating scenarios manually in game and saving the game states to be reused for further tests. the results were then noted down by hand and loaded into the R code file that produced the graphs displaying the data. Raw data snippets will be displayed in the addendum for readers to get a better idea of the results as well as an insight to how these algorithms were being measured in real time.

The first set of tests compared how each algorithm solved checkers problems. Unfortunately due to the nature of the field, this ended up being a very limited sample pool. The algorithms were tested in 3 separate problems (figures 6, 7 and 8). The color at the bottom of the board is what was being played and their opponent was played by their respective algorithm.

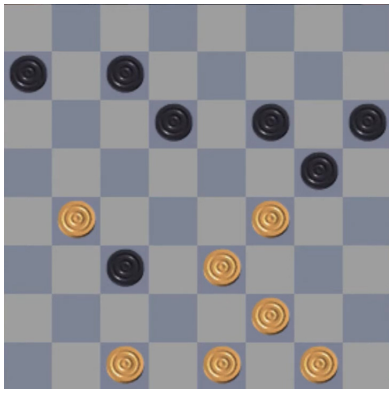

Fig. 6: problem 1 taken from [19]

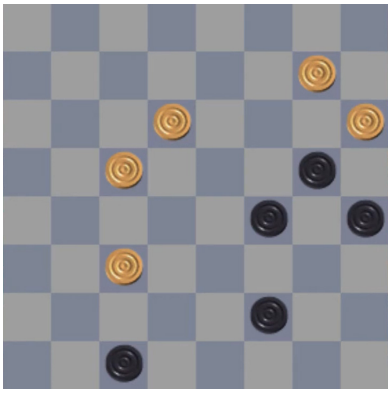

Fig. 7: problem 2 taken from [19]

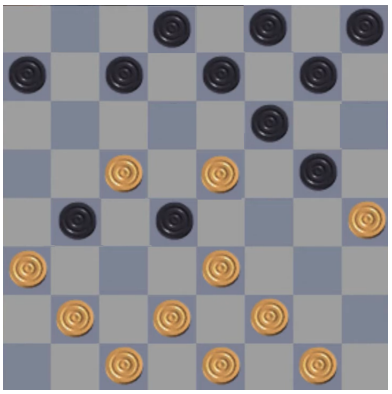

Fig. 8: problem 3 taken from [19]

X. Results - Test 1 - 3

The first set of tests showed no large gap between any one of the algorithms. The test was carried out by gradually increasing the allowed depth of the algorithm and seeing whether they'd arrive to the expected goal. In the graphs that represent this data a 1 is used for when the algorithm doesn't make the correct choice, while a 2 is used for a correct move. Unfortunately this data isn't particularly interesting since in most tests each algorithm would find the correct move by the 3rd depth increment (as seen in fig.9).

XI. Results - Match ups

the second set of tests pit all 3 algorithms against each other in a series of match ups. The same Depth metrics were used as in the previous experiment, where a shallow search allowed MINMAX to search to a depth of 16 and allowed UCT and OLETS 0.5 seconds of computing time, a medium search 32 depth and 1 seconds, and a deep search 64 depth and 2 seconds.

The results this time ended up a lot more varied than in the previous tests. As is standard, white has the advantage in any game so half the games every algorithm played were white while the other were black. In the Bar graphs presented, each algorithm's wins were averaged out and turned into a numerical number. 1 was used for wins and 0 was used for losses.

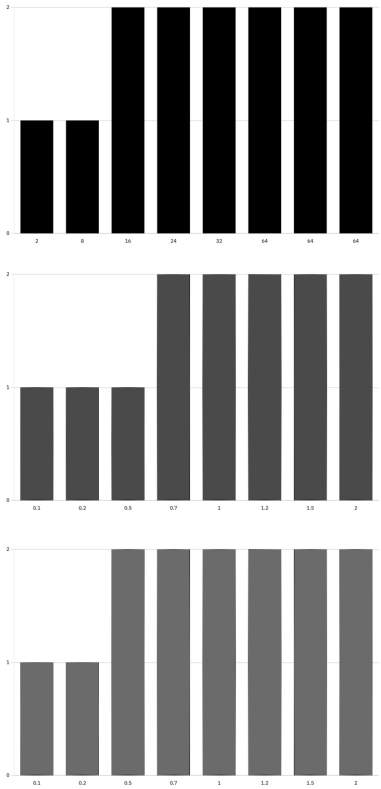

Fig. 9: Test 2 table. MINMAX, UCT, OLETS in order

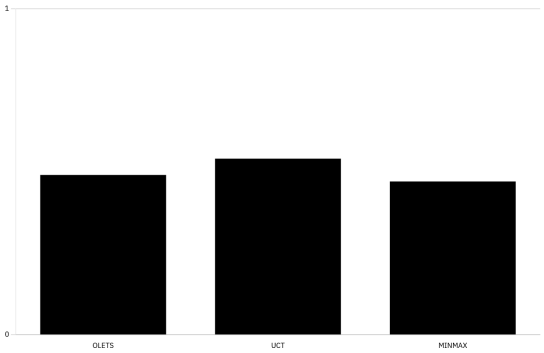

Fig. 10: Shallow Search, OLETS, UCT, MINIMAX

In the round containing the shallow search parameters the results were still rather even across; with OLETS scoring a 51, UCT scoring a 54, and MINIMAX scoring a 45 (fig.10)

The medium search round is where the difference in performance between the 2 advanced algorithms and MINMAX becomes more clear. While OLETS manages to keep up with UCT, MINIMAX has fallen behind by a considerable amount. OLETS scores a 63, UCT a 68, and MINIMAX a 31.

Once we reach the deep search MINIMAX has completely fallen out of the picture. By the end of the testing period MINIMAX only managed to score 3 points while OLETS scores 71 and UCT scores 76. (fig.12)

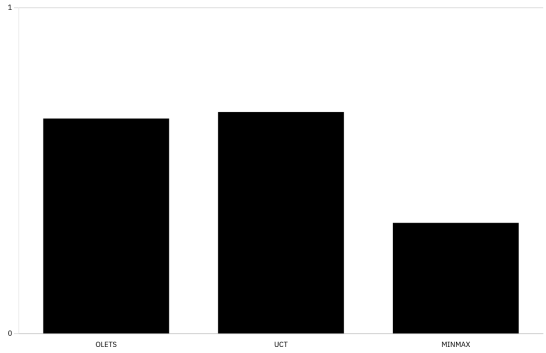

Fig. 11: Medium Search, OLETS, UCT, MINIMAX

Laying MINMAX to rest, I've condensed the deep search graph to only include OLETS and UCT in order to display a better comparison between the 2 (fig.13).

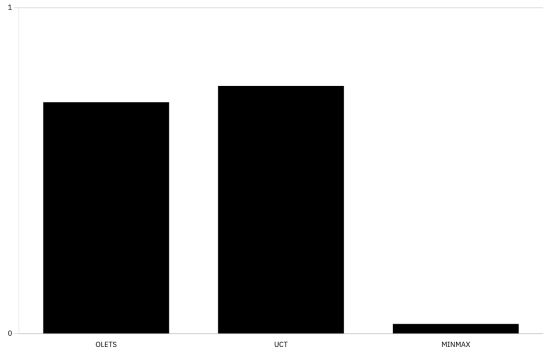

Fig. 12: Deep Search, OLETS, UCT, MINIMAX

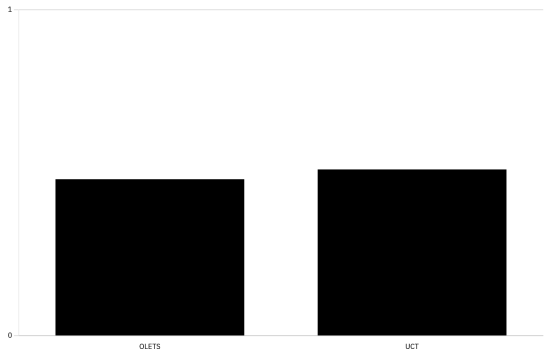

Fig. 13: Deep Search, OLETS, UCT

XII. Analysis and Discussion

After concluding the testing phase it was quite clear that there exists a gap between UCT and OLETS, although not a particularly substantial one. The performance difference between the 2 hangs consistently at around 2 to 5 percent, with the margin being even smaller if we were to solely consider the nodes each algorithm manages to explore each search. While

the results didn't show any difference significant enough to confidently declare a 'winner', the research still managed to show how well OLETS held up against UCT. While the 2 algorithms may measure to be quite equal at the moment, it is the nature of algorithms to be iterated upon and improved upon endlessly. With OLETS remaining fairly new and its infinite unexplored, it hasnt had the chance to grow as UCT did. Future research might put OLETS far ahead of UCT or vice versa. That's the beauty behind the simplicity of this field; the iteration and constant attempts at improvement never truly end.

XIII. Conclusion

This paper outlines measuring an emerging algorithm, OLETS, against a long-standing standard within the tree search field. Overall, both algorithms showed signs of success, and while UCT won more games by a slight margin, the data pool is still quite small. The tests were done on a single game and neither algorithm were given any changes to possibly improve their performance. I intend to expand on it in the future and conduct a more varied series of tests that evaluates both algorithms to their full extent.

References

[1] H. J. van den Herik, J. W. H. M. Uiterwijk, and J. van Rijswijck, "Games solved: Now and in the future," *Artificial Intelligence*, vol. 134, no. 1, pp. 277–311, 2002.

[2] J. Schaeffer, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, "Checkers is solved," *Science*, vol. 317, pp. 1518–1522, 2007.

[3] Y. Wang, Z. Yang, H. Qiu, and X. Liu, "Application and improvement of UCT in computer checkers," in *2018 5th IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS)*, 2018.

[4] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couëtoux, J. Lee, C.-U. Lim, and T. Thompson, "The 2014 general video game playing competition," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 3, pp. 229–243, 2016.

[5] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*, 2006.

[6] T. H. Kjeldsen, "John von neumann's conception of the minimax theorem: A journey through different mathematical contexts," *Archive for History of Exact Sciences*, vol. 56, no. 1, pp. 39–68, 2001.

[7] A. Samuel, "Some studies in machine learning using the game of checkers (reprinted from journal of research and development, vol 3, 1959)," *Ibm Journal of Research and Development*, vol. 44, pp. 207–226, 2000.

[8] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron, "A world championship caliber checkers program," *Artificial Intelligence*, vol. 53, no. 2, pp. 273–289, 1992.

[9] J. Schaeffer, R. Lake, P. Lu, and M. Bryant, "CHINOOK the world man-machine checkers champion," *AI Magazine*, vol. 17, no. 1, pp. 21–21, 1996.

[10] A. Elnaggar, M. Gadallah, M. Mostafa, and H. Eldeeb, "A comparative study of game tree searching methods," *International Journal of Advanced Computer Science and Applications*, vol. 5, pp. 68–77, 2014.

[11] M. Campbell and T. A. Marsland, "A comparison of minimax tree search algorithms," *Artificial Intelligence*, vol. 20, pp. 347–367, 1983.

[12] T. Vodopivec, S. Samothrakis, and B. Ster, "On monte carlo tree search and reinforcement learning," *Journal of Artificial Intelligence Research*, vol. 60, pp. 881–936, 2017.

[13] samishawl, "Upper confidence bound algorithm in reinforcement learning." Available at https://www.geeksforgeeks.org/upper-confidence-bound-algorithm-in-reinforcement-learning/ (2020/02/08).

[14] A. Mahajan and D. Teneketzis, "Multi-armed bandit problems," in *Foundations and Applications of Sensor Management*, Springer US, 2008.

[15] R. D. Gaina, A. Couëtoux, D. J. N. J. Soemers, M. H. M. Winands, T. Vodopivec, F. Kirchgeßner, J. Liu, S. M. Lucas, and D. Perez-Liebana, "The 2016 two-player GVGAI competition," *IEEE Transactions on Games*, vol. 10, no. 2, pp. 209–220, 2018.

[16] D. Anderson, "dvander/boardgames." Available at https://github.com/dvander/boardgames (2011/06/21).

[17] A. Weinstein and M. Littman, "Bandit-based planning and learning in continuous-action markov decision processes," *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 22, pp. 306–314, 2012.

[18] F. Speare, "FaydSpeare/checkers." Available at https://github.com/FaydSpeare/Checkers (2018/12/06).

[19] azcheckers, "azcheckers - YouTube." Available at https://www.youtube.com/@azcheckers/videos (2023/12/21).

Appendix A
Reflection

I am quite new to this subject matter. Both in programming tree search algorithms and board games. I've looked at and referenced several similar algorithms over the course of making this artefact and ended with what I assumed was a very competent skeleton to continue building the artefact from. However, this proved to be wrong later down the road. Once I was at the a point of implementing the algorithms I found that the structure I built my artefact on was slowly becoming more and more incompatible with the algorithms I was writing. Several times, I contemplated rebuilding the whole artefact to suit my purposes better. Due to time constraints, I ended up instead cutting the code down to more flexible functions and classes and crafting the algorithms in a way that complimented my initial design instead of fighting it. One of the choices I made that I was most grateful for was storing the pieces in a 2D array. Although the process of handling both the Board space and the World space was tedious at first, it paled in comparison to what would have awaited me if I hadn't. It was truly the best way to handle the data, and if I were to revisit the project in the future, I would craft the algorithms around the board system more carefully. I would also change the structure of my workflow. Instead of fully completing an algorithm, I would slowly work on all 3 simultaneously. This would not only help deepen my understanding of how these algorithms function differently in a real time technical perspective, but also aid with the algorithms having more of a cohesive style. Due to me learning a lot during this process, some of the newer code benefits from certain degrees of sophistication that older code lacks. It would have also prevented me from having the create a similar feature several times in order to suit the styles of the different algorithms.

Minimax and UCT specifically would benefit from this greatly. Since OLETS and UCT both get their information through run throughs, a lot of code was able to be recycled, and they were able to communicate with each other a lot easier when needed. However, since minimax was created first an on top of that also has a different computing flow from the other two, integration was a lot more difficult.

Overall, if i were to approach this topic once more, or if i were to revisit it, I would probably end up rebuilding a lot of it from the ground up with the knowledge I have

now. The product is solid as it is, but it lacks a lot regarding optimality and cohesion. With the knowledge I've gained, I would be able to cut down on a lot of code as well and create something more efficient and cleaner.

APPENDIX B
MINIMAX MAIN LOGIC LOOP

```
if (maxPlayer)
{
    // max score
    float MaxEval = -100000;

    // best node
    Pieces[,] bestMove = null;

    // gets all possible moves and stores them in
    a board

    List<Pieces[,]> allMoves =
    getAllMoves(true, board, true);

    // for every move runs a minmax eval
    for (int i = 0; i < allMoves.Count; i++)
    {
        // runs an eval with 1 less depth

        // it will loop through this function
        again and gather all possible moves
        for the boards that have been changed
        once already

        // but it will hand the search and
        evaluation over to mini

        // does this until depth = 0

        float evaluation =
        minimax(allMoves[i],
        depth - 1, false, false).Key;


        // compares board against
        current max

        MaxEval =
        Mathf.Max(MaxEval, evaluation);

        if (MaxEval == evaluation)
        {
            bestMove = allMoves[i];
        }
    }

    // returns best board
    return new KeyValuePair<float,
    Pieces[,]>(MaxEval, bestMove);
}
```

APPENDIX C
UCT MAIN LOGIC LOOP

```
 // goes through all children of a node
 below the threshold

while (true)
{
    if(node.children == null)
    {
        if(node.visits >= maturityThreshold)
        {
            // gets all possible children
            expand(node, board);

            // if no more children
            if(node.children == null)
            {
                // record state
                whitewin = true;
                history[depth++] = node;
                break;
            }
            continue;
        }

        // check whos won
        whitewin = playOut(board);
        break;
    }

.
.
```

APPENDIX D
OLETS STRUCT

```
//used to store all node data

public class Agent
{
    public int visits;
    public float score;
    public List<Agent> children;
    public Pieces[,] state;
    public Agent parent;
    public Pieces piece;

    public Agent
    (Pieces[,] state, Agent parent)
    {
        this.visits = 0;
        this.score = 0f;
```

```
        this.children = new List<Agent>();
        this.state = state;
        this.parent = parent;
        this.piece = new Pieces();
    }
}
```

```
my_data <- read.csv("MINMAX_T1.txt")

library(ggplot2)

#bar plot
ggplot(my_data, aes(x = Category, y = Count)) +
  geom_bar(stat = "identity") +
  theme_minimal()
```

APPENDIX E
UNIT TESTS

```
    //test
    public void doRandom()
    {
        Pieces = gameManager.findAllMoves(gameManager.whiteTurn, Pieces)[0];
        refreshBoard(Pieces);
        UpdateBoard();
        gameManager.whiteTurn = !gameManager.whiteTurn;
    }

    //test
    public void doMinMax()
    {
        KeyValuePair<float, CSS_Piece[,]> minMaxTest = minmax.Minmax(Pieces, minmaxDepth, game
        Pieces = minMaxTest.Value;
        refreshBoard(Pieces);
        UpdateBoard();
        gameManager.whiteTurn = !gameManager.whiteTurn;
    }

    //test
    public void doUCT()
    {
        Pieces = UCT.UCTMainLoop(Pieces, gameManager.whiteTurn, UCTThinkTime);
        refreshBoard(Pieces);
        UpdateBoard();
        gameManager.whiteTurn = !gameManager.whiteTurn;
    }

    //test
    public void doOLETS()
    {
        CSS_Piece[,] nextBoard = CSS_OLETS.OLETSMainLoop(Pieces, gameManager.whiteTurn);
        Pieces = nextBoard;
        refreshBoard(Pieces);
        UpdateBoard();
        gameManager.whiteTurn = !gameManager.whiteTurn;
    }
```

APPENDIX F

R code snippet

```
#bar graph for first series of tests
setwd("C:/Users/Stefanie/DataResults")
# file path

# read data
```