Othello/Reversi Final Report

Daniel Ginzburg
Isaac Wodele
AJ King
ginzb014
wodel008
king1517
12/18/2024

1 Abstract

This study evaluates the performance of three AI algorithms, minimax, alpha-beta pruning, and Monte Carlo Tree Search on the strategy game Reversi. Each algorithm was tested against a random agent to assess win rates and computational efficiency. The minimax algorithm achieved a remarkable win rate of 96% with a depth cutoff of 3 but had an average runtime of 60.29 seconds per game. The alpha-beta agent, also with a depth cutoff of 3, demonstrated comparable performance with a 95% win rate while significantly reducing runtime to 10.31 seconds due to its pruning mechanism. These high win rates are thanks to the heuristics formula they used, which took into account the difference in chips, choices, and corner chips for each state. In contrast, the Monte Carlo agent with 200 simulations achieved a lower win rate of 73% while averaging a runtime of 75.34 seconds per game. These results highlight the trade-offs between win rate and computational efficiency. Alpha-Beta Pruning shows itself as the most efficient algorithm for deterministic environments like Reversi. While Monte Carlo Tree Search excels in complex and non-deterministic environments, its performance in Reversi is limited by high computational cost and reduced accuracy in games with well-defined optimal moves. This analysis offers insight into the strengths and limitations of AI in strategic gameplay.

2 Introduction

The field of AI has increasingly turned its attention to strategy games as an effective testing space for newer algorithms and techniques. Among these, Reversi sticks out for its simplicity of learning and its difficulty in mastering. Despite its relatively straightforward rules, the game poses challenges for computer agents due to the sheer amount of possible moves and the need for long-term strategic planning. Reversi's uniqueness makes it a great candidate for studying adversarial search algorithms and other AI methodologies designed for decision-making problems in a game environment. This project takes a look at the application/effectiveness of various game-playing agents that use different algorithms all in an attempt to optimize the gameplay of Reversi. The primary objective is to develop, test, and compare various agents that leverage well-known techniques such as Minimax, Alpha-Beta Pruning, and Monte Carlo Tree Search. Each of these algorithms offers distinct advantages and trade-offs in terms of win rate, computation time, and overall efficiency in complex game states. While implementing and analyzing these methods within the context of Reversi, the goal is to better understand their strengths and limitations while

contributing ideas that may aid broader applications. In this project, the focus is on understanding how different factors such as how an agent evaluates moves, how deeply it thinks ahead, and how efficiently it uses resources. Testing these algorithms in various game situations will hopefully clarify which approach works best and why. The following sections review not only Reversi itself but also the algorithms experimented on, as well as how their performance is tested followed by the results of the experiment.

3 Project Analysis

3.1 Problem Brief

Reversi, also known as Othello, is a two-player strategy game played on an 8x8 board. Each player takes turns placing their discs on the board, aiming to flank and capture their opponent's pieces by enclosing them between their own. The game features a simple rule set but demands deep strategic thinking, as players must anticipate their opponent's moves and consider the long-term implications of their placements. The goal is to have the majority of discs in your color on the board when no more moves can be made. With each move, players must balance offense and defense, employing various tactics and adaptations to outsmart their opponent and secure victory. To maximize the score, certain algorithms can be modified to produce the most optimal strategy.

This project will attempt to replicate, improve, and compare algorithms for optimizing Reversi. Reversi is a complex game that forces the players to think many moves ahead. Unlike games with a fixed, deterministic outcome, Reversi requires dynamic, real-time decision-making, making it a challenging problem for algorithmic solutions. This project can contribute by comparing the optimization and efficiency of different algorithms such as Minimax, Alpha-Beta Pruning, and Monte Carlo Tree Search.

3.2 Reversi

3.2.1 The Game

Reversi, also known as Othello is a strategy board game for two players, typically played on an 8x8 board. Out of 64 black and white colored disks, the goal is to have the most disks in the player's color by the end of the game. A paper by Anton Leouski titled Learning of Position Evaluation in the Game of Reversi focuses on training an agent to play Reversi intermediately using another method and also describes the game state details of Reversi very well: "Like chess, it is a deterministic, perfect information, zero-sum game of strategy between two players, black and white"[13]. Players take turns placing disks on the board. A valid move must flank one or more of the opponent's discs between the placed disc and another disc of the same color(in a straight line horizontally, vertically, or diagonally). Reversi is Japanese-based and is renowned for its simplicity in a way that it is easy to learn, but almost impossible to master. This is because it is difficult for humans to envision dramatic changes even just a couple of moves in advance[13]. Computers have a clear advantage in this aspect. A known strategy within the game of Reversi is to control the corners as corners cannot be outflanked. Avoid moves that grant the opponent access to the corners. Plan in an attempt to trap the opponent's pieces and keep as many as possible. There are many other strategies within the game of Reversi, but most require a player to view multiple moves in advance which proves challenging for most.

3.2.2 AI Application

With Reversi being a simple yet popular two-person adversarial game, it is a great candidate for creating and testing different types of computer-playing agents. For example, in

Yiwei Han's paper, he applies a few common approaches to agents for the game of Reversi[8]. These approaches include Minimax, fixed depth alpha-beta pruning with a heuristic, and reinforcement learning where he combines Monte Carlo Tree Search with a Q-learning function and Neural Network[8]. Han concludes that all three methods work well as agents for Reversi and that the level of agent success is determined by its resource consumption through depth limits[8]. The more resources the agent is allowed, the better it will perform[8]. Another paper by Barber, Pretorius, Qureshi, and Kumar details similar strategies and also applies them to the game of Reversi. In their paper, they test Reversi agents using classic Minimax, Alpha-Beta Pruning, NegaScout (Principle Variation Search), and Central Neural Networks[1]. They found that adversarial methods like Minimax and Alpha-beta pruning outperformed the Neural Networks in the game of Reversi but are greatly affected by the heuristic used[1]. They also noted how Alpha-Beta and NegaScout visited significantly fewer nodes than Minimax making them more efficient[1]. The reason they suspect for the Neural Networks' relatively poor performance is because the network cannot see how good or bad a move is[1]. It only knows what previous players have done in a given scenario based on the data sets it has been trained on[1].

3.3 Software

The software used in this experiment will be an open-source Python implementation of Reversi created by JRChow[10]. It allows for not only human vs agent games but also agent vs agent games which will be used primarily for the experiment. The games will be played via the terminal as that will allow for the quickest experimentation as compared to the GUI version. Aside from the game itself, the agents and their subsequent strategies will also be implemented in Python. The random agent and the alpha-beta pruning agent have already been implemented so the Monte Carlo Tree Search agent and Minimax agent will be added in addition.

3.4 Importance

Experimentation with AI agents in games like Reversi offers valuable insights into decision-making processes that are directly applicable to real-world problems. In Reversi, agents are required to weigh different move options, predict the actions of their opponents, and adjust their strategies in real time. This mimics the types of challenges faced in more complex domains such as supply chain management or autonomous vehicle navigation. By experimenting with algorithms in this controlled environment, researchers can test and refine strategies in a simpler and more manageable setting before applying them to situations with greater uncertainty and complexity. This allows for the improvement of decision-making algorithms, providing a foundation for solving real-world problems with a higher degree of confidence and accuracy. Lastly, the outcomes of these experiments can highlight strengths and weaknesses in various strategies. These offer valuable lessons that can inform future advancements in fields that require dynamic and strategic decision-making [16].

4 Literature Review

To gain a complete understanding of adversarial search algorithms and their applications, reviewing past work around adversarial two-player games and their compatibility with the game Reversi is a necessity. According to an article about the application of reinforcement learning to the game of Othello by Nees Jan van Eck and Michiel van Wezel, games like Reversi/Othello are sequential decision-making problems where performance can be easily measured. This makes experimenting

and testing different agents with performance measures a fairly straightforward procedure. To identify a search strategy in an agent that outperforms the alternatives, analyzing what has already been accomplished is crucial for understanding the task. First, gaining a complete understanding of the game of Reversi itself can help outline potential performance measures and aid the reader in understanding the objective. Second, going over AI applications to the game, including how easy it is to test different search strategies and what makes Reversi a good candidate game to experiment with is equally essential. Then, examining the algorithms themselves outside Reversi will clarify how they work and what they accomplish. After learning about the algorithm itself, the next step is to analyze previous applications of the algorithms for Reversi. This will help answer questions such as how efficiently each algorithm performs playing the game, and what aspects of the algorithm would give it an edge over other search strategies. The main search strategies focused on in this literature review include Minimax, Alpha-Beta Pruning, and Monte Carlo. Finally, it's important to touch on other approaches to adversarial problem solving, such as Evolutionary Neural Networks or other search algorithms not previously discussed. This is important as there are various ways to create game-playing agents and with the recent advancements in Machine Learning, the field will grow even larger.

4.1 Minimax

4.1.1 The Algorithm

The Minimax search algorithm is a widely used search strategy used for making competitive game agents. In an article about minimax, and different heuristic functions within Connect-4 titled Research on Different Heuristics for Minimax Algorithm: Insight from Connect-4 Game by Xiyu Kang, Yiqi Wang, and Yanrui Hu, they discuss the background of minimax: "Minimax algorithm comes from the "Minimax theorem", which was proposed by John von Neurmann in 1928[11]." The algorithm has been updated and extended since then, optimizing it for more players and making it more efficient. It uses a recursive technique to evaluate an entire game tree. At each given leaf node, minimax will apply a heuristic function to calculate a game score and determine the desirability of that given position. Minimax works its way up the game tree, alternating between minimizing and maximizing players to choose the best path for the player assuming the opponent will always make the best move possible according to an article about alpha-beta pruning by Carl Felstiner which also mentions how minimax evaluates[4]. In a typical game environment, the max nodes represent the player as the player wants to maximize score. Similarly, the opponent represents the min nodes as minimax wants to minimize the opponent's options and therefore score. This paper by Carl Felstiner discusses the minimax algorithm itself and also applies it to simpler games such as Tic-Tac-Toe and Hex. The minimax search algorithm is excellent for smaller game trees when it has the computing time to search the entire tree. However, when it comes to larger game trees, minimax tends to fall short. This is why minimax is often used as a base search strategy for other algorithms such as alpha-beta pruning where it can simplify the game tree by determining which branches of the game tree do not need to be explored.

4.1.2 Minimax with Reversi

Minimax can be used to evaluate Reversi game trees as the game has a perfect information structure, where both players have full visibility of the board and the rules are deterministic. Reversi has a relatively manageable state space especially when compared to games like chess or Go. Granted it might be more optimal to cut down the game tree

with some of the search algorithms derived from minimax that prune. Minimax does a well enough job at evaluating potential moves which makes it a worthy algorithm to experiment with according to an article written by Vaishnavi Sannidhanam and Muthukaruppan Annamalai titled An Analysis of Heuristics in Othello[17]. The paper was made by a couple of people from the University of Washington and discusses the different search strategies used on Reversi agents. They focus on the amount of new research that has been put into AI game-playing agents. They also look at the different searching strategies within the game of Reversi. And point out the different types of heuristic functions used along with these strategies. The minimax algorithm systematically explores all possible moves and countermoves in an attempt to maximize the player's advantage while minimizing the opponent's best response. In the case of Reversi, heuristic functions can be manipulated to prioritize certain positions on the board. An example of this could be setting the corners to be a more desirable spot on the board for the agent. Or even little things like just because a move could get our agent a lot of pieces to change color, it might set up the opponent for an even bigger move two moves later[17]. Minimax would do a good enough job of catching these sorts of things. One of the only downsides of Minimax is there is not really a depth cut off and the entire tree is searched regardless of branches that could've been pruned otherwise. Another article titled Hierarchical Reinforcement Learning for the Game of Othello by Timothy Chang focuses on mostly reinforcement learning agents within the game of Reversi. However, they have a massive literature review on all the search strategies used as well. When talking about minimax, they explain how it is more optimal to limit minimax. In their thesis, they implement a minimax algorithm with a certain depth cutoff. At a certain depth, if minimax has not already terminated, he would essentially terminate it himself which could be helpful for minimax's long computing time and might make things easier to experiment with[3]. At the end of the day, it is expected that other search strategies derived from minimax should end up being more efficient than minimax but it is still believed to be a worthy search algorithm to test with.

4.2 Alpha-Beta

4.2.1 The Algorithm

Alpha-beta pruning is an optimization technique used in the minimax algorithm, primarily used for decision-making in two-player games such as chess or tic-tac-toe. As stated earlier, the minimax algorithm evaluates possible moves in a game tree by simulating all possible future game states and selecting the best move. However, it has been found that this search can be computationally expensive, especially with larger trees. Alpha-beta pruning improves efficiency by eliminating branches of the game tree that do not need to be explored. It maintains two values, alpha and beta, representing the best score achievable by maximizing and minimizing players, respectively. If during the search, it is determined that a branch cannot lead to a better outcome than an already explored branch, the algorithm "prunes" that branch, avoiding unnecessary evaluations. The pruning process can lead to a significant reduction in the number of nodes examined, allowing the algorithm to explore deeper levels of the tree in the same amount of time, without affecting the final decision[6]. The branching factor of a tree refers to the average number of nodes each node generates within a game tree. Experts say that under ideal conditions, alpha-beta pruning can explore half the nodes in a tree and exponentially reduce the runtime of the search[15]. While alpha-beta pruning might have conditions where it is optimal, it still has scenarios where is struggles. Researchers came up with the idea of multi-cut pruning to try to solve these weaknesses. It

allows the algorithm to cut more than one sub-tree or node by implementing additional comparisons. This should increase pruning efficiency and eliminate large portions of the game tree effectively. It was shown to have the biggest increase in games with large branching factors that a traditional alpha-beta search might struggle more with[2].

4.2.2 Alpha-Beta with Reversi

In Reversi, each player attempts to outflank and capture the opponent's pieces by strategically placing their pieces on the board. The game can be represented as a tree of possible moves, where each node corresponds to a game state, and each branch represents a potential move. Using alpha-beta pruning, the minimax algorithm evaluates these game states by simulating future moves and selecting the optimal strategy. Alpha-beta pruning works by recursively traversing the game tree, pruning branches where it is determined that no better outcome can be achieved than what has already been explored. For instance, if a move leads to a game state that is worse than an already evaluated state for the current player, the algorithm prunes that branch, saving computational resources. This pruning dramatically reduces the number of game states the algorithm needs to evaluate, allowing the AI to search deeper into the game tree and make stronger decisions faster, without needing to explore every possible move. In Reversi, where the game tree can grow exponentially with each turn, alpha-beta pruning significantly enhances the AI's ability to play at a higher level by focusing on the most promising branches of the tree[12, 14].

4.2.3 Heuristics for Reversi

As mentioned previously[1, 17], the success of methods like minimax and alpha-beta pruning heavily depends on the algorithm's heuristic. A heuristic is the formula used to determine how good a given state is. In a paper by Jacopo Festa and Stanislao Davino, multiple heuristic algorithms are applied to variations of alpha-beta pruning[5]. The heuristics in question are Score (HS), Mobility (HM), Mobility and Corners (HMC), Mobility, Corners, and Edges (HMCE), Mobility, Corners, Edges, and Stability (HMCES), and Mobility, Corners, Edges, and Stability, Time-variant (HMCEST)[5]. Aside from HS, which simply counts the number of tiles for each side, the majority of the other heuristics focus on potential scenarios beneficial to the agent such as corner pieces and how many moves an agent can make[5]. These heuristics were tested with alpha-beta pruning at various depth levels[5]. The results showed that as depth increased, the success of the agent increased as well at the expense of computing time[5]. As for the heuristics, HMC performed the best when given the max depth of 7, suggesting that a slightly simpler heuristic is better with a greater depth[5]. In general, however, the best-performing heuristic was HMCES, which had the highest average win percentage across the depth ranges[5].

4.3 Monte Carlo

4.3.1 The Algorithm

The Monte Carlo Tree Search strategy is a prominent approach to adversarial games and subsequent agents. As mentioned in a paper by Świechowski et al, the algorithm was originally proposed in 2006 to make computer players for the game Go[18]. By default, Monte Carlo doesn't make use of heuristics like other adversarial strategies such as alpha-beta but can function just by knowing the rules of the game[18]. The paper discusses not only the application of base Monte Carlo but also modified versions of the algorithm that improve upon it in one way or another[18]. Base Monte Carlo Tree Search uses a four-step process when it comes to its strategy: Selection, Expansion, Simulation, and Backpropagation[18]. Selection in-

volves choosing the next node according to the selection policy. Then, unless the selection node is in a terminal state, at least one new node is expanded to the tree. From this new node, random simulations are performed until a terminal state is reached and a payoff is recorded. Finally, the payoffs are backpropagated up the tree which updates the nodes[18]. The algorithm is often modified depending on the specifics of the problem being solved. For example, with games involving perfect information such as the game of Reversi, Monte Carlo's can be modified with Action Reduction, Early Termination, or UCT alternatives[18]. Action reduction seeks to eliminate possible actions, specifically bad ones, from decision states [18]. Early termination sets cut-off depth to the random layouts to enhance algorithm efficiency[18]. UCT alternatives involve modifying the Upper Confidence Bound for the Monte Carlo Tree[18].

4.3.2 Monte Carlo with Reversi

As Reversi is a classic perfect information game, Monte Carlo Tree Search can therefore be applied. In an article by Roy Hung, he discusses his implementation of Monte Carlo Tree Search and tests it against randomized agents in the game of Reversi[9]. Two variations of the algorithm are pitted against a pseudo-random algorithm: one with no domain knowledge and one with domain knowledge[9]. The pseudo-random agent has a benchmark win rate of 46% as black[9]. Both variations are tested using a different number of simulations and a different number of iterations. Both variations easily beat the pseudo-random agent. The variation without domain knowledge sees a consistent win rate of greater than 95% at around 20 iterations with even the number of simulations being as low as 20[9]. The variation with domain knowledge improves win rates with fewer iterations and simulations[9]. However, as the number of iterations increases, the algorithm performs slightly worse than the one without domain knowledge[9]. This suggests that the heuristic might hinder the algorithm's performance in late-game[9]. While the algorithm succeeds against random agents, its performance against human opponents isn't great as it struggles to beat players that have at least some experience with the game of Reversi[9]. By default, Monte Carlo Tree Search isn't the most formidable game agent for Reversi, but with enhancements like increased simulations/iterations paired with domain knowledge through the use of heuristics, its performance can be drastically improved[9].

4.4 Other Approaches

4.4.1 Evolutionary Neural Networks

Aside from the more classical approaches to adversarial problems, there are other ways to create agents for the game of Reversi. In a paper by Gunuwan et al, they discuss a Reversi Agent that uses an evolutionary neural network[7]. A neural network involves a a series of nodes sorted into three layers, an input layer, a hidden layer, and an output layer[7]. This graph system is combined with a genetic algorithm that allows the network's nodes to update and change[7]. For this to happen, 20 individuals are created with a mutation factor that allows variation among them[7]. Then, these individuals are played against each other to determine the most fit agent[7]. After the best individual is selected, it is tested against a negamax algorithm using a greedy heuristic[7]. The evolutionary neural network performed very well against negamax winning approximately 77% of the games[7]. This provides evidence that with proper training, evolutionary neural networks can be used to create a successful Reversi agent.

5 Experiment/Results

5.1 Approach

5.1.1 Minimax

The first algorithm tested in the Reversi experiment is a modified version of the classic minimax algorithm. The base version of the algorithm can be seen in Figure 1. The minimax algorithm involves two main functions, Max-Value and Min-Value. These functions combine to compute the utilities of different results in the game tree to determine the optimal move. It also assumes that the player is trying to maximize utility while the opponent is trying to minimize it. The algorithm makes use of a heuristic function to evaluate game states as well. Due to reversi's relatively big game tree, modifications to this algorithm are made for the experiment. To speed up the runtime, a depth limit of 3 is imposed on the algorithm. This means that once that level is reached in the game tree, the algorithm uses the heuristics function to evaluate the current state instead of traversing to a terminal state[16].

```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: Minimax Pseudo code [16]

5.1.2 Alpha-Beta Pruning

The second algorithm tested in the Reversi experiment is a modified version of the alpha-beta pruning algorithm. Figure 2 shows the pseudo-code of the algorithm. Much like minimax, alpha-beta also makes use of the Max and Min functions. However, the algorithm also tracks two additional values as it traverses the game tree: alpha and beta. These two values allow for the algorithm to prune certain branches of the game tree that are deemed as obsolete. This significantly improves performance with time and space complexity both being enhanced. To allow for consistent comparison, the algorithm will be modified with a depth limit of 3 like minimax[16].

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

Figure 2: Alpha-Beta Pruning Pseudo code [16]

5.1.3 Heuristics

The heuristics evaluation function used in both the minimax and alpha-beta algorithms mentioned above is made up of three scoring calculations[10]. The first is coin difference, which simply compares the number of chips or coins each player has. The second is choice difference, which calculates how many choices each player has at a given state. The third is corner difference, which tracks the difference between the number of corners captured by each player. The corner spots are especially

important as once they are captured by one player, they can not be flipped back. The results of these three calculations are weighted and combined into a single evaluation function that is used as the heuristic[10].

$$\text{Coin Difference} = \frac{\text{Coins of Max - Coins of Min}}{\text{Coins of Max + Coins of Min}} \times 100$$

$$\text{Choice Difference} = \frac{\text{Choice of Max - Choice of Min}}{\text{Choice of Max + Choice of Min}} \times 100$$

$$\text{Corner Difference} = \frac{\text{Corner Captured By Max - Corner Captured By Min}}{\text{Corner Captured By Max + Corner Captured By Min}} \times 100$$

Figure 3: Heuristics Formulas [10]

5.1.4 Monte Carlo Tree Search

The final algorithm tested in the Reversi experiment is a modified version of the Monte Carlo Tree Search algorithm. The figure below shows the pseudo code for the algorithm. The Monte Carlo Tree Search algorithm builds a search tree using iterations of random simulations. These simulations evaluate the potential outcome of the actions in search of an optimal strategy. For each explored node, the simulation will play out the game using random actions until a goal state is reached. This search begins at the root node of the tree and uses a combination of unknown options as well as the best-known options in order to explore other simulations. These results are then propagated back up the tree, updating the nodes in its path. In order to achieve the most optimal runtime and win percentage, a simulation limit of 200 was placed on the algorithm. This means that once the Monte Carlo Tree Search algorithm has run through 200 simulations, it will pick the move it deems most optimal[16].

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
  tree ← NODE(state)
  while IS-TIME-REMAINING() do
    leaf ← SELECT(tree)
    child ← EXPAND(leaf)
    result ← SIMULATE(child)
    BACK-PROPAGATE(result, child)
  return the move in ACTIONS(state) whose node has highest number of playouts
```

Figure 4: Monte Carlo Pseudo code [16]

5.2 Experiment

5.2.1 Testing Phase

In order to set up a proper experiment, there were many conditions that needed to be met. These include things such as randomization to eliminate bias, a control or baseline group to compare to, a sufficient sample size, as well as code that can be replicable. To meet these criteria, the decision was made to play each algorithm versus a random agent to create a baseline at a sufficient sample size of 100 games using replicable open-source code. Similar steps were taken into consideration when deciding on the head-to-head match-ups and how they would be approached. In the head-to-head experiments, it was decided that the algorithms would face off against each other 50 times each, for a total of 100 additional games per algorithm. The depth limit and simulation count were also decided during this phase. Whatever values were able to produce the best win rate within the most reasonable time was selected.

5.2.2 Agent vs Random

To evaluate and compare the effectiveness of the algorithms, each one would play 100 matches of Reversi against a random agent, with 50 games as the black player and 50 games as the white player. As mentioned before, the minimax and alpha-beta agents were set to a depth limit of 3 while the Monte Carlo Tree Search agent was constrained by a simulation limit of 200. After completing the 100 games, the program would output the number of wins for each algorithm and the average runtime in seconds. This approach al-

lows for a direct comparison of the algorithms, shedding light on their strengths and computational demands.

5.2.3 Agent vs Agent

The next phase of the experiment is to have each algorithm play against the other algorithms (ex: Minimax vs Alpha-Beta Pruning). As there are three agents, each agent played 50 matches versus the other two. Of those 50 games, 25 games were played as the black player and the other 25 were the white player. The agents were each using their previously defined limitations for both depth limit in minimax and alpha-beta and simulation amount in Monte Carlo. After the 50 games had been completed, record the amount of games won, and track the average runtime.

5.3 Results

5.3.1 Minimax vs Random

The first algorithm tested was the minimax agent against a random agent. Even with the lower depth cutoff of 3, minimax was able to achieve the highest win rate of 96% against the random agent. This, however, came at a cost as the algorithm had an average game time of just over a minute with a time of 60.29 seconds.

5.3.2 Alpha-Beta vs Random

Following the minimax agent, the alpha-beta agent was tested against the random agent with a depth cutoff of 3 as well. The agent was put against a random agent in 100 games. The Alpha-beta agent managed to win 95% of its game vs the random agent which nearly equaled the minimax algorithm. Had more games been run, their performance in terms of win percentage would be identical. The thing that sets alpha-beta apart, however, is its runtime, averaging only 10.31 seconds per game. This difference can be attributed to the alpha-beta pruning mechanism, which is constantly eliminating branches in the search tree that cannot influence the agent's final decision.

5.3.3 Monte Carlo vs Random

Finally, Monte Carlo was tested against a random agent throughout 100 games. The agent achieved a win rate of 73% against the random agent in a configuration with 200 simulations. This performance was noticeably worse compared to the alpha-beta and minimax agents. Not only that but the Monte Carlo agent also performed slower than minimax, averaging 75.34 seconds per game. This extended runtime reflects the agent's need to repeatedly simulate game states to estimate the value of its next move. These results highlighted the inherent trade-offs when it comes to using a Monte Carlo-based agent, especially when a relatively lower win rate comes at a hefty price.

5.3.4 Minimax vs Alpha-beta

The first head-to-head matchup is between the minimax and alpha-beta pruning algorithms. The agents played 50 matches against each other using a depth cutoff of 3 and the same heuristic function. The outcome of these two functions was a 50/50 split and an average runtime of 62.35 seconds. This result is not surprising as alpha-beta pruning is simply a more efficient version of minimax.

5.3.5 Alpha-beta vs Monte Carlo

Within a direct competition between Alpha-beta and Monte Carlo, Alpha-beta had a clear win over Monte Carlo with a win rate of 96%. With Monte Carlo once again using 200 simulations and Alpha-beta using a depth limit of 3, it's no surprise Monte Carlo couldn't win often. On top of that, the average time per game was 61.62 seconds per game, which is likely due to Monte Carlo's poor computation time.

5.3.6 Monte Carlo vs Minimax

Lastly, Monte Carlo against minimax for the last 100 games yielded some predictable win rates. Minimax won against Monte Carlo 92% of the time which can be concluded as essentially the same win rate as alpha-beta against Monte Carlo. However, the computation time significantly differed at 151.41 seconds which is due to the poor runtime of minimax and the even worse runtime of Monte Carlo.

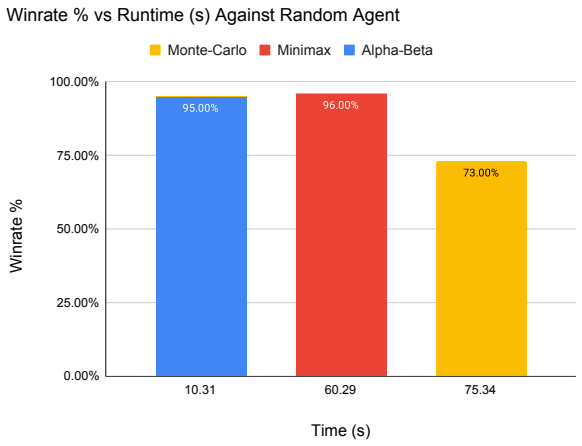

Figure 5: Win Percentage vs Runtime

	Percent % Lost			
Percent % Won	Algorithm	Alpha-Beta	Minimax	Monte Carlo
	Alpha-Beta	NULL	50.00%	96.00%
	Minimax	50.00%	NULL	92.00%
	Monte Carlo	4.00%	8.00%	NULL

Figure 6: Head to Head Win rates of Algorithms

6 Analysis

6.1 VS the Random Agent

After performing the experiment and collecting the results of the algorithms' performance vs the random agent, it is pretty clear which algorithm performed the best. When comparing the win percentages of the three agents, alpha-beta and minimax nearly tied, both achieving a win percentage 95%. This success can be partially attributed to the algorithm's base functionality but it also can be attributed to its evaluation function/heuristic. Both algorithms made use of a combination heuristic that took into account the coin/chip difference, the choice difference, and the corner difference for any given state. By using this heuristic, minimax and alpha-beta were able to play moves strategically compared to the random agent, beating it nearly every game. The Monte Carlo agent still performed well against the random agent, managing to win 70% of its games. While lacking the benefit of a heuristic function, the Monte Carlo agent made up for it by using random simulations to determine the best move. For the runtime, however, one agent outperforms the rest. That algorithm is the alpha-beta agent, which had an average runtime of 10.21 seconds per game. Minimax and Monte Carlo both came in at over a minute with Minimax taking on average 60.29 seconds and Monte Carlo taking on average 75.34 seconds.

6.2 VS each other

After analyzing the three agents a random agent, the agents were then pitted against each other. In the first matchup, minimax and alpha beta played 50 games against each other, each playing 25 as black and 25 as white. After the 50 games, minimax and alpha beta had beaten each other for the same amount of games, each winning 25. This could have something to do with them each using the same heuristic but it goes to show that

from a move-making perspective, they are essentially equal. Next up was minimax vs Monte Carlo. In this matchup, minimax won 92% of the games it played meaning Monte Carlo only won 8%. This pattern was similar to the alpha-beta and Monte Carlo matchup in which alpha-beta managed to win 96% of its games against Monte Carlo. This result isn't surprising as minimax and alpha-beta both performed extremely well against the random agent in comparison to Monte Carlo both win-wise and time-wise. This result is mainly due to Monte Carlo's lack of heuristic usage when making moves. While alpha-beta and minimax can take advantage of a heuristic to strategically pick moves based on the specifics of the game, Monte Carlo is limited to random playouts which while effective against a random agent, struggle to compete against a strategic agent.

7 Conclusion

This study highlights the strengths and weaknesses of applying AI algorithms to the game of Reversi. The algorithms used were minimax, alpha-beta pruning, and Monte Carlo tree search. The minimax algorithm ended with the highest win rate of 96% but suffered from its lengthy runtime, making it less practical for real-time applications. The alpha-beta pruning algorithm proved to be the most efficient, putting up a nearly identical win rate of 95% while significantly reducing the computational time through its elimination process. These high win rates are thanks to the heuristics formula they used, which took into account the difference in chips, choices, and corner chips for each state. On the other hand, the Monte Carlo Tree Search algorithm showed its limitations in deterministic environments like Reversi, in which clearly defined optimal moves exist. Regardless of its moderate win rate of 73%, its runtime was the highest, showing the cost of repeated simulations. Overall, the alpha-beta pruning algorithm emerged as the most balanced approach, showing strong performances in win rate and efficiency. These findings reinforce the importance of aligning the choice of algorithm with the characteristics of the game environment and resources available when designing AI for games with complexity.

8 Contributions

AJ wrote the Introduction, The Game, Importance, Literature Review, Minimax, and Results.

Daniel wrote the Problem Brief, Importance, Literature Review, Alpha-Beta, Approach, Experiment, Graph/Charts, Data Collection.

Isaac wrote the AI application, Software, Literature Review, Heuristics, Monte Carlo Tree Search, Approach, Graph/Charts, Data Collection, Analysis.

[1]

References

[1] Barber, A., Pretorius, W., Qureshi, U., and Kumar, D. An analysis of othello ai strategies. CS7IS2 Project (2019/2020), 2020. barberal@tcd.ie, pretoriw@tcd.ie, mqureshi@tcd.ie, kumardh@tcd.ie.

[2] Björnsson, Y., and Marsland, T. Multi-cut pruning in alpha-beta search. *Artificial Intelligence 101*, 1-2 (1998), 1–17.

[3] Chang, T. Hierarchical reinforcement learning for the game of othello. *Masters of Science Thesis* (2023), 102.

[1] We used generative AI for grammar checking, bibliography citation formatting, and help with structuring/outlining our ideas

[4] Felstiner, C. Reinforcement learning for playing othello. *Whitman College Mathematics Department* (2019).

[5] Festa, J., and Davino, S. Iago vs othello: An artificial intelligence agent playing reversi. `mailto:festajacopo@gmail.com,woods88@hotmail.it`, 2024. Accessed: 2024-11-22.

[6] Fuller, S. H., Gaschnig, J. G., and Gillogly, J. J. Analysis of the alpha-beta pruning algorithm. Technical report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, July 1973.

[7] Gunawan, Armanto, H., Santoso, J., Giovanni, D., Kurniawan, F., Yudianto, R., and Steven. Evolutionary neural network for othello game. *Procedia - Social and Behavioral Sciences 57* (2012), 419–425. International Conference on Asia Pacific Business Innovation and Technology Management.

[8] Han, Y. Application of different artificial intelligence methods on reversi. In *Highlights in Science, Engineering and Technology: Proceedings of CMLAI 2023* (2023), vol. 39, University of Rochester, Rochester, USA, p. 1338. yhan32@u.rochester.edu.

[9] Hung, R. A reversi playing agent and the monte carlo tree search algorithm. *Medium Article*, September 2019.

[10] JRChow. Reversi. GitHub repository, 2023. Accessed: 2024-11-04.

[11] Kang, X., Wang, Y., and Hu, Y. Research on different heuristics for minimax algorithm: Insight from connect-4 game. *Journal of Intelligent Learning Systems and Applications* (2019), 17.

[12] Korf, R. E. Multi-player alpha-beta pruning. *Artificial Intelligence 39*, 1 (June 1990), 37–58.

[13] Leouski, A. Learning of position evaluation in the game of othello. *Department of Computer Science* (1995), 15. Technical Report UM-CS-1995-023.

[14] Olivito, J., Gonzalez, C., and Resano, J. Fpga implementation of a strong reversi player. *IEEE Access 9* (2021), 123456–123467. DIIS-I3A, University of Zaragoza, Zaragoza, Spain; Architecture Department, Complutense University of Madrid, Madrid, Spain.

[15] Pearl, J. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Artificial Intelligence 19*, 2 (1982), 47–58.

[16] Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2020.

[17] Sannidhanam, V., and Annamalai, M. An analysis of heuristics in othello, 2023. Department of Computer Science and Engineering, Paul G. Allen Center, University of Washington.

[18] Świechowski, M., Godlewski, K., Sawicki, B., et al. Monte carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review 56*, 3 (2023), 2497–2562.