Colin Song & Isabelle Ho
261049870
261094492

COMP424 Final Project:
Othello AI Bot - Yogurt Cup

Introduction

Othello is a turn-based game where 2 players place discs on a board. This board is traditionally 8x8, but our bot is adjusted to be able to play on any numbered board from 6x6 to 12x12. This is a territorial game where players attempt to capture their opponent's pieces as their own and claim the territory. To make a move, the player must capture their opponent's piece(s), or else their turn gets passed. Othello's rules are simple, but the strategies can become complex. To build the most effective Othello-playing AI bot, we analyzed a selection of algorithms and ran numerous tests for evaluation.

The strongest algorithm we found

Throughout the creation of our Othello AI Bot, Yogurt Cup, we wrote different algorithms and a hybrid mix of these algorithms. Ultimately, we concluded that a hybrid algorithm of the Monte Carlo Tree Search (MCTS) and the minimax algorithm with alpha-beta pruning is the strongest algorithm to play Othello.

We initially created a Greedy Algorithm bot that prioritized capturing the most number of tiles at every point of the game as we thought that the quantity of captured tiles equates to one's territory. This was before we solidified our understanding of Othello. After trialing this bot against random_agent and gpt_greedy_corners_agent, we soon found that this greedy algorithm is extremely short-sighted and naive as it only maximized immediate gain without future planning. We soon learned that short-sighted agents taught us that certain agents do well early on but once we reach the end game, they cannot prioritize the win over the immediate captures. These types of strategies showed us that we need to broaden the evaluating abilities of our agents to not be shortsighted and look at the game in multiple stages.

After both engaging in gameplay against an AI bot online and creating a minimax with alpha-beta pruning agent to play, we saw the importance of separating the stages of the game as it was clear that there are significantly fewer ways the endgame can play out[1]. The minimax agent essentially worked as an endgame solver, where its exhaustive search algorithm works the best in the final leg of the Othello game as the optimal play is easier to find within fewer states. This type of opponent showed us that no matter how we approached the early stages of the game, we must switch to deterministic strategies in the endgame. That is how we decided to incorporate minimax with alpha-beta pruning in our endgame.

With our endgame algorithm decided, we circled back to our approach for the early to mid-game. Even though we realized that the early captures do not manifest ultimate wins, early moves still matter as keeping key positioning can be game-winning. However, there are way more possibilities in the early to mid-game that it would be computationally expensive with minimax with alpha-beta pruning to properly simulate such large spaces within the given time limit (2 seconds). Minimax with alpha-beta pruning

[1] Egilsson, Einar. "Reversi." *Reversi | Play It Online!*, Rauðás Games ehf, cardgames.io/reversi/.

alone would not be sufficient as it depends on its evaluation function for finding good moves but there is too much uncertainty within the early game. By uncertainty, we are referring to the possibilities of a good move turning bad or vice versa. The search space is too large to be focused on only exploiting one move that cannot promise to be beneficial all the way through. Minimax does not balance exploitation and exploration well enough in these uncertain environments as it would just exploit the current best move and not explore the unknown moves that might eventually become even better than the current best move. This is why we looked for an algorithm that is known for being flexible in exploration and exploitation while able to handle large uncertain search spaces without exhausting all computational power, the Monte Carlo Tree Search.

The most effective approach we found to optimizing our bot is to split up the early-mid game and the endgame for two different algorithms to handle. We believe that we achieved a high-level play quality from this approach. The MCTS algorithm is ideal for the early to mid-game stages as it excels in exploring large spaces of uncertainty while balancing exploration with exploitation. We then switch to using the minimax algorithm with alpha-beta pruning for the late game because there are fewer available moves and game states left so we can use exhaustive searches to ensure optimal play. We chose MCTS for its strength in balancing exploration and exploitation and minimax with alpha-beta pruning for its deterministic precision which is ideal within smaller search spaces. Our hybrid approach looks at the stage of the game and uses the most fitting algorithm for the strength of the chosen type of search.

Explanation of our agent design

Our agent design, as mentioned above, is focused on using a hybrid mix of two algorithms, the Monte Carlo Tree Search and Minimax with alpha-beta pruning. These two algorithms are also supported by heuristics and domain knowledge so that our agent can adapt dynamically. Our agent would be directed on which algorithm to use based on the current stage of the game.

- **Monte Carlo Tree Search**

MCTS is a probabilistic search technique that implements heuristics in its decisions. In the early to mid-game, MCTS is chosen as it excels in exploring large, uncertain search spaces while also balancing its decision between exploration and exploitation[2]. At the start, there are simply too many possible game states. To counter this uncertainty, MCTS's main purpose, when "given a game state to choose the most promising next move", will help condense the high computational complexity into only the promising moves[3]. Essentially, MCTS simulates numerous scenarios of the game to predict the next best move from the current game state. This idea of simulating until it finds the next most promising move is not as simple as it seems, there are steps involved in how MCTS determines a move "promising". MCTS can be broken down into 4 distinct phases; selection, expansion, simulation, and backpropagation.

---

[2] Meger, David. (Fall 2024). *Monte Carlo Tree Search* [Course Presentation]. https://us.edstem.org/api/resources/42314/download/L8-MCTS.pdf

[3] Czarnogorski, Kamil. "Monte Carlo Tree Search - Beginners Guide." *Int8.Io*, 26 Apr. 2021, int8.io/monte-carlo-tree-search-beginners-guide/.

- **Selection**: In the selection phase, MCTS navigates from the root node to a leaf node while using the UCT (Upper Confidence bound applied to Trees) formula[4].

$$\frac{w_i}{n_i} + c\sqrt{\frac{lnN}{n_i}}$$

Where: $w_i = number\ of\ wins\ that\ came\ from\ the\ node$

$n_i = total\ number\ of\ simulations\ of\ this\ node\ (visit\ count\ of\ this\ child)$

$N = parent\ node's\ total\ number\ of\ simulations\ (visit\ count\ of\ parent\ node)$

$c = exploration\ parameter$

$\left(\frac{w_i}{n_i}\right)$ = the section of UCT that goes through the "exploitation"

$\left(c\sqrt{\frac{lnN}{n_i}}\right)$ = the section of UCT that focuses on "exploration"

In the case of Othello, the root node would refer to the current game state and the leaf node would be a partially explored state or terminal game states. Each node in the MCTS tree represents a single board state. Exploration encourages the discovery of moves that lead to new board states to make sure that potentially promising moves are not overlooked. Exploitation refers to focusing on the moves with higher rewards and prioritizing these promising moves. MCTS during the selection phase uses the **Tree Policy**, guided by the UCT formula. This policy within MCTS would first start at the current game state and evaluate all its future possible game states using UCT then choose the highest value UCT to continue its exploration. This process within the Tree Policy continues recursively until it reaches an untried/new game state or the end of the game. Using the Tree Policy in the selection phase, our MCTS would be able to balance exploration and exploitation.

- **Expansion**: After the selection phase selects a child node, the expansion phase adds new *unexplored* child nodes to the selected node. In the case of Othello, these new child nodes all represent the possible game states that can happen from the selected child node. The possible game states would be determined dynamically based on the current game state as a new node is created for each new unexplored move.
- **Simulation**: Within the selected child node, MCTS simulates the game either until a set depth or until the very end. In our case, we set a maximum depth limit of 10 for MCTS. Usually during this phase, MCTS would use some type of **Default Policy** to select which move is best or to just select a move, such as using heuristics or a random choice. For our MCTS's judgement, we used domain knowledge. With this method, we gave every position on the Othello board a value and compared the values of each possible move within the simulation. The Default Policy focuses on playing out the games with a given method of judgement (in this case, a domain knowledge heuristic) to provide an estimated value for each game state.

[4] https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/

- **Backpropagation**: Once a terminal node is reached, that means we have reached the final game state in Othello (game over, no more moves can be played). This final step updates the values of the nodes in the path it took to reach that terminal node. These values would change depending on whether the terminal state resulted in a win, loss, or draw. The visit counts for the nodes are also incremented within this step. After each backpropagation, the MCTS algorithm will gain more knowledge of the potential moves.

- **Minimax with Alpha-Beta Pruning**

Minimax is a recursive search algorithm that finds the best strategy while assuming the opponent plays optimally[5]. There are min and max nodes that alternate between each depth of the tree, which in the case of Othello would be the board states after a move has been played. Max would represent our agent and Min would represent whoever is our agent's opponent. This search algorithm expands the complete search tree until it reaches the terminal game states[6]. At the end of the games, it would compute the utilities and then backtrack while comparing the values. At a max node, we keep the highest value among the node's children while at a min node, we keep the lowest value of its children. Similarly to MCTS, we have a depth limit of 15 for minimax.

➔ **Alpha-Beta Pruning:** The algorithm "prunes" the branches of the search tree that do not bear fruit. This means that it would eliminate all the possible losses led to by certain moves and those moves are the ones we "prune". Alpha represents the highest score achieved by our agent while Beta represents the highest score achieved by the opponent. When the score (node value) has a positive value, we know that max is winning and when the score is negative, we know that min is winning. Now with alpha-beta pruning, minimax's efficiency greatly improves as alpha-beta pruning would eliminate the unfavorable game states.

- **Combining the algorithms**

We combined these two algorithms dynamically based on the current game phase. MCTS is used during the early-mid game to navigate the large branching factors with the help of the UCT formula, and the Tree and Default Policies within the 4 phases. To understand MCTS in Othello in the 4 phases, first, we select a valid move, then we expand to see all the new board states that the move leads towards and simulate the board states until the end of the games. Following the simulations, we update the path of the tree with the outcomes of our games and gain more knowledge as to which moves lead to the best results. Our agent will continue to use MCTS until the game reaches a point where there are 15 remaining moves. Minimax is then used for the end game, where a smaller branching factor can let our agent search and evaluate more exhaustively. Now that the search space is smaller in the end game, we can ensure that alpha-beta pruning is being put to use, whereas, in the early game, it would be difficult to get to alpha-beta pruning as there is a time limit to each move. For our heuristics, we use domain knowledge through value boards which we use arrays to represent. These value boards carry the assigned positional weights in each tile for difity. Stabiferent board sizes. We created these value boards with the ideas of other potential heuristics in mind, such as stability and mobility refers to the stable discs that cannot be flipped on the board due to their superior positioning. Mobility is the difference in the number of available moves between our agent

---

[5] Meger, David. (Fall 2024). *Alpha-Beta Game Search* [Course Presentation]. https://us.edstem.org/api/resources/42175/download/L7-alpha_beta.pdf

[6] Meger, David. (Fall 2024). *Game Playing* [Course Presentation]. https://us.edstem.org/api/resources/42098/download/L6-Games1.pdf

and opponent. The idea of these heuristics were vital in creating our value boards. For example, corner discs are crucial in their stability and potential for the end game, therefore, they receive a higher positional weight on the board. By using our domain knowledge to make informed moves, we guarantee that by the end game, we can capitalize on our positioning through the deterministic minimax with alpha-beta search.

Analysis of our agent's quantitative performance

1. Depth Level:

Given the differences between MCTS and Minimax and the stages at which we utilize the algorithms, we have set a search depth to optimize performance and decision-making efficiency. For MCTS, we concluded that a maximum search depth of 10, offers an optimal balance—it is sufficiently deep to evaluate promising game states while remaining shallow enough to ensure broad exploration of viable moves, thereby enhancing the agent's strategic versatility and overall effectiveness. We found this max depth level by simply adjusting the depth level that MCTS should search up to and analyzing the performance of our agent during the early and middle stages of the game. For Minimax, we determined that a maximum search depth of 15 is optimal. Depths beyond 15 exceed the 2-second time constraint for identifying the best moves, while shallower depths reduce the effectiveness of Minimax, limiting its ability to evaluate sufficient moves to secure a decisive advantage in the game. Similar to how we found the depth level for MCTS, we adjusted the depth level that Minimax should search up to such that our agent would effectively make decisions that led to our win.

2. Breadth

During the early stages of the game (defined as when up to 20% of the board is occupied), we use MCTS. The breadth in this phase is relatively small, as the number of valid moves typically ranges from 6 to 13. The breadth increases as more tiles are placed on the board and our simulation handles more valid moves for the player. In the mid-game (when approximately 20–70% of the board is filled), the number of valid moves increases significantly, averaging around 20, as more pieces on the board create additional options. This phase achieves the maximum breadth, as the search space is at its largest. As the game progresses into the endgame, the number of valid moves diminishes due to fewer open tiles, resulting in a smaller search space. At this point we use Minimax and begin to see the breadth at each level continue to diminish until we reach the breadth of 1 (last open tile on the board) or 2 (last 2 open tiles on the board)

3. Impact of board size:

We observed that our agent performs better when playing on smaller boards (6x6, 7x7, and 8x8). This is because each move has a greater impact on smaller boards, where the limited space amplifies the outcome of each decision. In contrast, on larger boards, the agent's moves tend to be less impactful overall, as the increased size dilutes the significance of most tiles, except for those with exceptionally high strategic value (edge, corner, X and centerpieces).

4. Heuristics and Move Ordering Approaches We Tried:

When developing our bot, we experimented with several heuristics to evaluate move execution and assess whether a given board state favoured the player. These heuristics included:

- ➔ Mobility: Evaluating the number of valid moves available to the bot compared to the opponent for a given board state. This was used to prevent the bot from flipping too many pieces during the early and mid-game stages.
- ➔ Stability: Measuring the number of stable discs (discs that cannot be flipped) compared to the opponent. This was used to ensure the bot established stable discs during the mid-game to create a favourable state for the late game.
- ➔ Frontier Discs: Counting the number of frontier discs (discs adjacent to empty tiles) compared to the opponent. This helped minimize the bot's frontier discs during the early and mid-game stages.
- ➔ Corner Occupancy: Tracking the number of corners controlled by the bot compared to the opponent. This ensured the bot prioritized decisions that led to securing more corners than the opponent.
- ➔ Inner Control: Favoring moves that flipped pieces to control the central four tiles of the board, particularly during the early game, to control the center.

We ultimately decided not to use every single heuristic. The bot struggled to effectively evaluate the quality of moves or the favorability of a game state, as the numerous heuristics added complexity and cluttered the decision-making process, reducing its effectiveness.

5. Prediction of Winrates:

i) Random agent: 191/200 = 95.5%
ii) Dave (an average human player): 17/20 = 85%
iii) Classmates' agents: 85%

Summary of Advantages and Disadvantages of Our Approach

- **Advantages:**
  - ➔ Usage of 2 algorithms (MCTS and Minimax): In the early and mid-game, we use MCTS, as the search space is too large for exhaustive exploration. MCTS selectively explores branches and estimates the value of moves using heuristics. In the late game, we transition to Minimax, where the reduced search space allows us to compute the game's outcome (win or loss) and identify the optimal move.
  - ➔ Depth limit: Implementing a depth limit for both MCTS and Minimax ensures efficient exploration of the search space, focusing on identifying the best moves to achieve favourable outcomes for our agent. This limitation also optimizes computational resources, avoiding unnecessary processing of deeper, less relevant states.
  - ➔ Domain knowledge: By giving the agent prior domain knowledge of the value of moves in Othello, the agent makes more informed decisions, leading to favourable board states.
- **Disadvantages:**
  - ➔ Lack of heuristics: Our agent relies on domain knowledge and does not incorporate a wide range of heuristics that could enhance its ability to evaluate moves and board states more effectively. As a result, the agent may occasionally overlook favourable moves that are not explicitly informed by the domain knowledge provided.

Colin Song & Isabelle Ho
261049870
261094492

- ➔ Depth limit: While the depth limit improves computational efficiency, it also constrains the agent's ability to fully evaluate the long-term impact of moves. Simulating to the end of the game, where a winner is determined, provides the most accurate assessment of a move's value, but this level of analysis is curtailed by the imposed depth restriction.

How Could We Improve

The next step for our bot is to involve incorporating additional heuristics to enable more accurate identification of favourable moves and game states. By integrating a broader range of heuristics, the agent can increase its chances of selecting the best moves without necessitating an increase in the depth limit. With more detailed information, the bot will be better equipped to evaluate and prioritize optimal strategies. Another improvement for our bot is to save the MCTS tree generated for subsequent turns. By reusing this tree, the bot can save computational resources by avoiding redundant simulations and searches for moves that have already been explored. This approach not only reduces computational overhead but also facilitates more informed searches by building upon the insights gained in previous turns. We can also improve our hybrid algorithm to transition from MCTS to minimax with alpha-beta dynamically between different board sizes. Instead of switching algorithms strictly at the 15 remaining moves mark, we could make this transition cater for each board size's end game. For example, the bigger board sizes (10x10 or 12x12), would benefit from an earlier switch to the late-game minimax. This is because starting the minimax early can allow the agent to fully exploit the advantages of stable discs, which are more vital in bigger boards. The transition to end-game minimax search being later is also less punishing on the smaller boards (6x6) as the number of possible moves shrinks quicker, so the cost is less for an exhaustive search. A reasonable range to switch to the minimax end game for the larger boards would be 20-30 remaining moves. After evaluating our agent, we believe that in the future, we could incorporate more heuristics, reuse the generated trees to save computational resources, and dynamically change the point at which we switch algorithms based on the board size.

Colin Song & Isabelle Ho
261049870
261094492

Works Cited:

Egilsson, Einar. “Reversi.” *Reversi | Play It Online!*, Rauðás Games ehf, cardgames.io/reversi/.

Meger, David. (Fall 2024). *Alpha-Beta Game Search* [Course Presentation]. https://us.edstem.org/api/resources/42175/download/L7-alpha_beta.pdf

Meger, David. (Fall 2024). *Monte Carlo Tree Search* [Course Presentation]. https://us.edstem.org/api/resources/42314/download/L8-MCTS.pdf

Czarnogorski, Kamil. “Monte Carlo Tree Search - Beginners Guide.” *Int8.Io*, 26 Apr. 2021, int8.io/monte-carlo-tree-search-beginners-guide/.

https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/

Meger, David. (Fall 2024). *Game Playing* [Course Presentation]. https://us.edstem.org/api/resources/42098/download/L6-Games1.pdf