COMP 424 Final Project Report
Reversi Othello AI Agent Analysis

Nicholas Belev ◇ Hamza Rashid
McGill School of Computer Science
3480 Rue University, Montréal, QC H3A 2A7
{firstname.lastname@mail.mcgill.ca}

December 4, 2024

Contents

1 Overview and Motivation

1.1 The Strongest Algorithm for Reversi

Careful and extensive research was conducted, along with experimentation and comparison of game-playing approaches. Among these approaches were multiple versions of an Alpha-Beta pruning Minimax algorithm and an elementary Monte Carlo move-selection. Consequently, the strongest algorithm identified for playing Reversi Othello is a depth-limited, Iterative Deepening Search (IDS) with Alpha-Beta pruning, coupled with an arsenal of weighted heuristic evaluation functions. The approach is complemented by a Zobrist key hashing for efficient game state memoization in a transposition table. This eliminates redundant computations across branches of the game tree, giving more time to depth exploration.

1.2 Overview of the Algorithm

This algorithm's behavior is inspired by professional-player Reversi strategy while benefiting from efficient search across the depth and breadth of possible board states. The core components are as follows:

1.2.1 Iterative Deepening Search (IDS)

IDS ensures that the search algorithm always finds at least some strategic move to play, under time constraints. The depths for IDS are initialized for each board size ($N \times N$ — $N \in \{6, 8, 10, 12\}$) to near-guarantee that an Alpha-Beta pruning tree search up to the starting depth will always complete, yielding a fair evaluation of all "terminal" states at that depth. If time runs out, in the IDS scope or the Alpha-Beta function's scope, the highest-valued move is played. IDS explores incrementally deeper with its remaining time to gain certainty of promising future states.

1.2.2 Alpha-Beta Pruning

This specific traversal of a Minimax tree optimizes the search by eliminating branches of the game tree that lead to states too highly-valued by the minimizer or maximizer to ever be permitted by the algorithm (for 'us' or for the 'opponent'). Coupled with a move ordering via a lightweight heuristic (to search moves from best to worst), it maximizes the probability of pruning (and at the shallowest depth possible), significantly reduces the number of nodes explored, and allows for deeper searches within the same space and time limits.

1.2.3 Heuristic Evaluation Function

A multi-component evaluation function determines the values of 'terminal' future game states. This determines the minimizing and maximizing decisions that the Agent simulates through the search process by assessing board states (nodes). The heuristic is a weighted average score of several factors:

- **Piece Difference**: The relative number of tiles controlled by the player compared to their opponent.
- **Corner Occupancy**: Control of corners, which are the most valuable locations due to being unflippable to the opponent's side and implicitly stable.
- **Corner Closeness**: Penalizing proximity to empty corners, as they create risk for the opponent capturing corners. These are the three squares horizontally, vertically, and diagonally adjacent to a corner (the first two types are called C-squares, and the last type is called an X-square).
- **Mobility**: The number of legal moves available to the player minus those available to the opponent. Attempt to maximize the player's mobility and minimize their opponents'.
- **Frontier Tiles**: Penalizing positions adjacent to empty squares, which are usually at higher risk of being flanked.
- **Disk Squares**: Using pre-computed positional weights for each square on the board, to determine the dominant side.
- **Stability**: States with high stability have a large amount of player tiles which can no longer be flipped.

1.2.4 Zobrist Hashing and Transposition Table

Zobrist hashing provides a space-efficient and computationally quick way to generate keys—representation of board states, enabling fast lookups and updates of board state information. This is incorporated into a transposition table, which ensures that previously evaluated state data is stored and quick to look-up for future computations that might encounter the same state (be it at the same or a different depth).

1.3 Rationale for the Approach

Reversi is a game of dynamic balance and evolving goals, where early-game mobility, proper distribution of pieces, and control of corners manifests into stable late game positions. The best performing strategy captures these nuanced objectives through a multi-faceted evaluation function and strategic depth adjustment during different phases of the game. The methodology (weighting scheme, depth, memoization, etc.) was refined based on human observation, experimental results, and insights from existing literature on Reversi strategies. For instance:

- **Math/Algorithm Analysis**: With effective move ordering, Alpha-Beta pruning theoretically reduces the search space from b^d to $b^{d/2}$, avoiding the equivalent of half the depth of the search tree (via pruning); for an average branching factor b and depth d.
- **Iterative Design**: Through experimentation, the depth limit and heuristic weights were tuned to optimize gameplay on standard boards (6×6, 8×8, 10×10, 12×12) with respect to a [1 to 2] second processing time limit.
- **Reading Sources**: Published strategies and computer programs on positional board scoring, stability, and other heuristics were consulted to achieve the design of the evaluation function.

1.4 Results and Conclusions

To revisit the simplistic Monte Carlo algorithm that was tested, it operates like a Monte Carlo Tree Search without adapting its exploration or exploitation to focus on more unexplored or promising moves. Instead, it looks at all valid moves from the current state, evenly splits 2 seconds among these moves and carries out as many random games per move as time permits. It selects the move with the highest wins-to-games ratio. It performed quite poorly—evident from its near-random gameplay in the middle-game where breadth of available moves is the greatest, and thus is spread too thin to calculate reliable win rates. This supported the decision to steer towards deterministic algorithms over stochastic options.

This culminated in the agent hereby named “ZMIDAB” (Zobrist-Memoized, Iterative Deepening, Alpha-Beta Pruning algorithm), which exhibits consistent performance across varied board sizes and opponents. Its ability to account for numerous dimensions of “good” gameplay give it a strategic advantage over opponents that rely on short-sighted move selection or shallow heuristics (such as Chat-GPT’s 1-step-lookahead Greedy Corner agent). Iterative deepening combined with Alpha-Beta pruning achieves high-quality optimal-choice gameplay while respecting the 2-second time and 500 MB memory constraints. ZMIDAB is certainly the strongest AI agent among the Alpha-Beta pruning Minimax algorithms we developed, due to memoization.

2 Agent Design and Theory

This agent was developed in a file entitled `student_agent.py`, which is an agent registered within the Othello Reversi code environment for this project. The game logic, graphic user interface, turn-taking, and error response, etc., are vastly handled by the auxiliary and helper scripts of the environment. The Artificial Intelligence comes from the functionality within the `StudentAgent` class of `student_agent.py`. The game logic calls `step(chess_board, player, opponent)` in order to receive the agent’s chosen (valid) move.

Enumerated below are the specifics of how a move is chosen by `StudentAgent`:

2.1 Initialization

The process associated with `student_agent.py` begins when the `StudentAgent`’s first move is requested by the game logic (controlled via `simulator.py`). Upon this first call to `step()`, the class data for this game is not yet initialized, so `initialize(chess_board)` is invoked to ensure that all board-size-dependent parameters and data structures—initially left unspecified by `__init__(self)`—are ready for use throughout the game. These include:

- **Board Size**: Derived from the dimensions of the provided `chess_board`.
- **Heuristic Weighting Scheme**: The heuristic weights for each evaluation metric (e.g., `piece_difference`, `corner_occupancy`) are set based on the board size from `weight_options`. This selection tailors the importance of different heuristics to the board size, e.g., larger boards increase the branching factor and make mobility more critical. Smaller boards emphasize tactical stability and a rush for corner control. Board size greatly affects game length, outcome-entropy, player-mobility, and position-stability.
- **Corner Positions**: Stores the indices of corner squares (e.g., (0,0) for the top-left corner). These positions are the most stable and often most valuable.
- **Initial Search Depth for IDS**: Uses the `init_depth` dictionary to determine the starting depth for Iterative Deepening Search (IDS) based on the board size, balancing computational complexity and search efficacy.
- **Square Weights**: Generates a positional weighting matrix (`base_weights`) that assigns strategic importance to specific squares on the board, such as corners and edges, enhancing heuristic evaluation.
- **Zobrist Hashing Table**: Constructs the `zobrist_table`, randomly assigning a 64-bit integer to every possible (row, column, square_value (one of 0, 1, 2 representing an empty square, a black piece, or a white piece)) combination. This table is used in-game to XOR together all cells into one long-integer, compactly representing a game state as a key for the transposition table.
- **Transposition Table**: Initializes an empty dictionary (`transposition_table`) to cache evaluations of previously encountered board states (and the depth at which they were encountered, indicating with a flag—exact, lower, or upper—indicating whether the evaluation is definitive or provides a bound for pruning), avoiding re-computing the value of the same state in different IDS iterations.
- **Caching Structures**: Additional caches (`move_cache`, `move_ordering_cache`) to store valid moves and prioritize promising ones, for pruning efficiency.

Once `self.initialized` is set to `True`, the above initialization task is skipped for the remainder of the game.

2.2 Iterative Deepening and Step Function

The game logic calls the agent's `step()` method to return a move. `step()` does the following:

2.2.1 Iterative Deepening Search (IDS)

- The agent progressively searches deeper into the game tree, starting with an initial depth determined by the board size (`init_depth`).
- For each depth, the agent calls the `alpha_beta_search()` function to compute the best move given the current board state and heuristic evaluation.

2.2.2 Alpha-Beta Pruning

- Performs DFS on Minimax tree (generated by simulating future moves for both players along the current branch and backtracking).
- Does not search deeper than the current maximum depth. Stores this terminal state's evaluation.
- Narrows the range of possible values (using alpha and beta bounds) to prune branches that have an overlap ($\alpha \geq \beta$), as they are guaranteed to result in a sub-optimal, worse outcome for the current player than a previously explored branch.
- Significant reduction in the effective branching factor allows deeper searches within the same computational limits.

2.2.3 Time Management

- To abide by the game's time limit, track remaining time (`time_limit - elapsed_time`) and halt further depth exploration once time runs out.
- If the search for a particular depth exceeds the time limit, the agent defaults to the best move found during the last completed depth.

2.3 Move Selection: Alpha-Beta Search

The `alpha_beta_search()` implementation is used to return the best next move based on evaluating the long-term consequences (terminal states) resulting from available moves. The following elements are central to its implementation:

- **Transposition Table Lookup**: Before evaluating a board state, the agent checks the transposition table using the Zobrist hash of the current board. If the state exists in the table:
  - The cached depth is compared to the current search depth:
    - If the cached depth is greater than or equal to the current depth, the cached value can be used to avoid further computation.
  - The flag determines how the cached value influences the search:
    - `exact`: The value is returned directly as the evaluation for the node.
    - `lower`: The cached value updates the alpha bound, potentially pruning worse branches for maximizing nodes.
    - `upper`: The cached value updates the beta bound, potentially pruning worse branches for minimizing nodes.
  - If the bounds ($\alpha \geq \beta$) are updated such that $\alpha \geq \beta$, further exploration of the branch is pruned, and the cached result is used.
- **Legal Moves**: The agent uses `get_cached_valid_moves()` to retrieve valid moves for the current player, caching results to minimize redundant computations.
- **Recursive Search**:
  - For each legal move, the agent simulates the move using `make_move()` and recursively calls `alpha_beta_search()` for the opponent.
  - After the recursion, the move is retraced using `undo_move()` to restore the prior board state.
- **Pruning and Bound Updates**: The algorithm updates alpha or beta based on the evaluation results, pruning branches where further exploration cannot improve the outcome for the side making the move.
- **Move Ordering**: Moves are sorted via the lightweight stability heuristic (`compute_stability()`), from highest to lowest, to increase pruning probability.
- **Result Caching**: Upon evaluating a state, the result is stored in the transposition table with that state's Zobrist hash as the key:

```
self.transposition_table[hash_key] = {
    "depth": depth,
    "value": value,
    "flag": flag,
    "move": best_move}
```

2.4 Evaluation Function

The agent uses a weighted evaluation function (`evaluate_board()`) to assign scores to board states. The heuristics are as follows:

2.4.1 Don't Lose

If the search encounters a future state where the agent loses, give it an infinitely negative score, to avoid a move that could lead to a loss.

$$\text{if is\_end and scores[opponent]} > \text{scores[player]: return } -\infty$$

2.4.2 Piece Difference

Measures the difference in the number of tiles controlled by the player and their opponent, rewarding dominance.

$$p = \frac{\text{my\_tiles} - \text{opp\_tiles}}{\text{my\_tiles} + \text{opp\_tiles}}$$

2.4.3 Corner Occupancy

Rewards capturing stable corner positions, as they cannot be flipped and provide implicit stability.

$$c = \text{my\_corners} - \text{opp\_corners}$$

2.4.4 Corner Closeness

Penalizes tiles near unoccupied corners (C-squares and X-squares); avoid conceding corners to opponent.

$$l = -1 \times (\text{my\_close} - \text{opp\_close})$$

2.4.5 Mobility

Encourages maximizing the player's legal moves while minimizing the opponent's, ensuring flexibility.

$$m = \frac{\text{my\_moves} - \text{opp\_moves}}{\text{my\_moves} + \text{opp\_moves}}$$

2.4.6 Frontier Tiles

Penalizes tiles adjacent to empty squares, as they are vulnerable to flipping.

$$f = -1 \times (\text{my\_front\_tiles} - \text{opp\_front\_tiles})$$

2.4.7 Disk Squares

Rewards control of empirically advantageous board positions using a precomputed weight matrix for squares on boards of varying sizes. Scores for these weights were heavily inspired by research on Othello game theory (Kukreja, 2013).

$$d \mathrel{+}= \text{base\_weights}[i][j] \text{ if board}[i][j] == \text{player else } -\text{base\_weights}[i][j]$$

2.4.8 Stability

Rewards your surplus of discs that cannot be flipped, ensuring long-term control over the board.

$$s = \frac{\text{my\_stable\_discs} - \text{opp\_stable\_discs}}{\text{my\_stable\_discs} + \text{opp\_stable\_discs}}$$

2.4.9 Combined Heuristic Evaluation

All metrics are aggregated with predefined weights to compute the final state evaluation.

$$\begin{aligned}\text{score} = {} & \text{weights['piece\_difference']} \times p \\ & + \text{weights['corner\_occupancy']} \times c \\ & + \text{weights['corner\_closeness']} \times l \\ & + \text{weights['mobility']} \times m \\ & + \text{weights['frontier\_tiles']} \times f \\ & + \text{weights['disk\_squares']} \times d \\ & + \text{weights['stability']} \times s\end{aligned}$$

2.5 Key Optimizations

Several design features enhance ZMIDAB's efficiency along the process:

- **Zobrist Hashing**: Enables constant-time comparison of board states, significantly reducing overhead in transposition table lookups.
- **Move Caching**: Reduces the cost of legal move generation by caching pruning and results data for encountered board states.
- **Iterative Deepening**: Balances depth exploration with time constraints, ensuring the best move is always returned within the allocated time.
- **Move Ordering**: Maximizes pruning by exploring paths stemming from promising immediate moves first.

3 Quantitative Performance Metrics

3.1 Depth Analysis

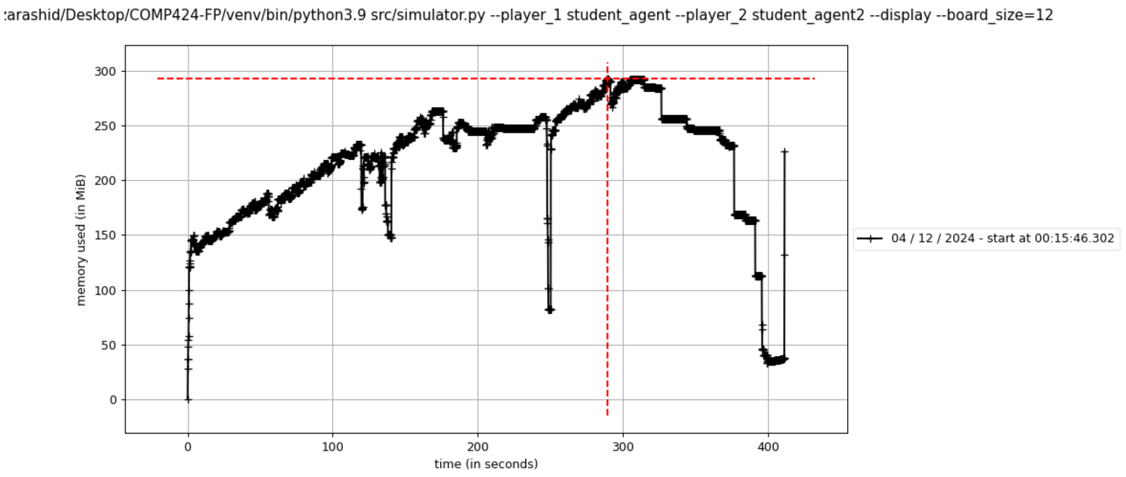
The agent achieves depths varying with board size, game phase, and opponent. The agent reaches the maximum IDS depth on the 6x6 board, regardless of game stage or opponent, due to its small size and branching factor. As for the larger board sizes, the agent is able to look ahead to the set limit consistently at the beginning of the game (when it makes its first move), and occasionally struggles to achieve the same look-ahead leading up to and during mid-game (25-66% occupied squares). We say occasionally because the memoization significantly improves its look-ahead overall, especially during mid-game, and it varies with the opponent's complexity; struggling more with sophisticated adversaries because good moves are more rare.

In the end-game, however, options become limited again, and the agent is able to look ahead to the minimum between: (its depth-limit for the corresponding board size) and (the number of moves along the longest path to an endgame state). We decrease the depth-limit as board size increases; this is effective both in terms of computational cost and game outcomes—the increased entropy on a larger board makes the breadth of search important as the agent is able to plan for more scenarios and thus generalize more effectively across a variety of opponents.

Concretely, the IDS explores depths from 1 to `self.max_depth` = { 6: 6, 8: 5, 10: 4, 12: 3 } (key: N in $N \times N$ board; value: max depth), time permitting.

3.2 Breadth Analysis

Breadth in our agent varies depending on board size, opponent strength, game stage, and the effects of pruning. As the game progresses, the number of legal moves at each level changes, typically increasing during the midgame as the board fills with more pieces and then stabilizing or shrinking in the endgame. The max and min players use identical strategies, leading to similar pruning behavior across turns. Memoization of states and caching mechanisms are critical to storing visited state data (especially across the breadth of the terminal level). Thus, the memory usage graph serves as a proxy to the explored breadth of the algorithm. Evidently, there is a consistent buildup of memory (indicating many states being stored) before stabilizing or shrinking, implying that our agent achieves significant breadth by exploring a large number of states and not repeating state-recomputations. The exponential rise in explored states during the midgame suggests that the breadth remains quite high in this phase.



```
:arashid/Desktop/COMP424-FP/venv/bin/python3.9 src/simulator.py --player_1 student_agent --player_2 student_agent2 --display --board_size=12
```

Figure 1: Total Memory Usage of ZMIDAB vs. a previous non-memoized Alpha-Beta Variant (12x12 board).

Regarding move ordering (which is crucial to achieving high effective breadth), we experimented with the evaluation function, penalization of C-squares, rewarding of corners, and the stability heuristic. Most were only effective in a limited number of settings, but the one that proved to be consistently effective across a variety of settings (board size, opponent, first or second player) is the stability heuristic. Furthermore, we implemented caching of move orderings, which helped the agent consider nearly double the (promising) branches of play, in the limited time given. Deducing from further tests on the code, the agent consistently prunes roughly 30% of the search space (obtained by dividing number of visited states by total potential states at all depths d), showing that the stability-heuristic move ordering was indeed effective.

3.3 Accounting for Board Size

The board size significantly impacts ZMIDAB's search behavior, particularly its look-ahead depth and search breadth. Larger boards inherently increase the branching factor due to the greater number of valid moves in mid-game states. This exponential growth in possibilities necessitates careful management of computational resources to ensure the agent performs optimally within the given time constraints. To address this, the `init_depth` variable sets the maximum depth for Iterative Deepening Search (IDS) based on the board size. For instance, on a 6x6 board, the maximum depth is 6, whereas it is reduced to 5 for an 8x8 board, 4 for a 10x10 board, and 3 for a 12x12 board. This adaptation balances the depth and breadth of the search, ensuring the agent completes its computations within the time limit and doesn't get cut off by the timer without having evaluated at least one full level.

Additionally, the agent's heuristic weights are customized for each board size, accounting for strategic differences in stability, mobility, and positional priorities. Larger boards place greater emphasis on mobility and stability, whereas smaller boards factor in the board weight matrix (to capture valuable positions sooner). Hence, the very nature of what a valuable move is, is conditioned upon the size of the board, in ZMIDAB's approach.

3.4 Heuristics, Pruning Methods, Move Ordering

ZMIDAB's current heuristic builds upon previous simpler metrics: coin parity, mobility, and corner control. These evaluations are now integrating into a more thorough, weighted sum tailored to board size, but independent of game phase. The prior Alpha-Beta Pruning model's heuristics were strongly correlated (e.g., prioritizing corners $\Rightarrow$ prioritizing stability). This means that various facets of board states that may make them valuable are missed by this combination of heuristics, resulting in a highvariance strategy that sometimes misses preferable moves. The scoring for these heuristics used two weighting schemes: early game and late game.

The refined heuristic addresses prior limitations by combining stability, mobility, and corner control with new dimensions such as frontier tiles and disk square weights. Frontier tiles penalize risky positions adjacent to empty squares, while disk square weights leverage predetermined values to assess the strategic significance of specific locations. The phase-based weighting scheme was found ineffective and removed. This evolution diversified its predecessor's risk and variability, and does not overlook subtly valuable states. Its calculations are also more time efficient.

The addition of a Zobrist hashed transposition table further expedites the move comparisons throughout the iterative deepening Alpha-Beta search. Move ordering is handled by the lightweight stability heuristic which is the best overall proxy for a "good state" (Kukreja, 2013). These various features yield a time-efficient, adaptive, and calculating agent that plays with the priorities of a professional.

4 Performance Predictions

4.1 Against a Random-Move Agent

Even though the nature of the game is stochastic—and there exists the theoretical risk of Random playing the "best game ever" of Othello Reversi—by looking ahead and having some intuition of "good" gameplay and loss prevention, ZMIDAB will nearly always outperform Random, with an empirically approximate win rate of 98%.

4.2 Versus Prof. David Meger

Against Prof. David Meger, who is an average human Othello player, the expected win rate is 95%. ZMIDAB has a much stronger ability to consider states far into the future, evaluate and compare them (using research-based heuristics), in order to choose a move in the present—compared to most humans who have not devoted significant time and memory space to Othello strategy. That being said, the Professor's level of gameplay is certainly superior to random, uninformed moves.

4.3 In a Tournament of COMP 424 Agents

In a tournament of agents created by the students of COMP 424, Fall 2024, the anticipated percentile by class rank is 65%. The research and calibration involved in developing ZMIDAB, its heuristics and their weighting schemes, along with the attention to optimization for depth and different board sizes, likely ranks this agent among the better ones in the class. Nonetheless, other groups may have found stronger heuristics, more adaptive weighting schemes, better proxies to winning gameplay (or avoiding loss), and so with near-certainty, there will be better agents than ZMIDAB.

5 Approach Summary

5.1 Advantages

- Despite our model's use of memoized states, cached legal moves, and move orderings, it only uses 300MB of RAM on the 12x12 board (and less on the others), as its maximum memory usage is limited by the time it is given to compute states.
- Alpha-Beta pruning has the search time complexity of BFS without the large memory requirements thanks to the IDS approach, which does not suffer from repeated depth search due to memoization of state evaluations.
- The aggregated heuristic evaluation is highly modularized, making it easy to fine-tune individual components of the model.
- Testing ZMIDAB against a variety of opponents and in a variety of settings is crucial to ensure that its weights are not overfitting to a particular opponent, board size, or player number (1 or 2).

5.2 Disadvantages

- The model utilizes many individual heuristics in its evaluation function, in an effort to reduce bias in identifying strategic moves. This implicitly makes it difficult to pinpoint which heuristic or weight is to blame for a poor move choice or overall performance.
- The heuristic is especially sensitive on the 6x6 board, where even a slight (in the order of 110) changes to heuristic weights can cause drastic changes in performance.
- Without extensive testing, there is a risk of overfitting a particular opponent, board size, or player position.

5.3 Expected Failure Conditions and Weaknesses

- The model's sensitivity to heuristic weights on the 6x6 board can lead to drastic and unexpected changes in performance.
- The complexity of the evaluation function makes it hard to identify specific causes of failure when the model performs poorly.
- If not tested sufficiently, and properly weight-calibrated, the model might fail to generalize and perform well against different opponents, board sizes, or player positions.

6 Future Development Improvement

Given additional time and resources, several strategies could be implemented to further enhance the agent's performance:

6.1 Machine Learning

A more modern approach to fine-tune heuristic weights would be by training the agent against a strategically-diverse assortment of strong-performing AIs. By using an optimization technique like gradient descent, the agent's evaluation weights could be fine-tuned based on performance across numerous games to maximally reduce the error with respect to every top agent. This approach would ensure that the agent is not only generalizable (as the weights are trained on many adversaries) but also capable of adapting to high-level playstyles. Fine-tuned weights are especially important given the heuristic's apparent sensitivity to small tweaks. Consequently, hand-tuning would be inferior in positively influencing the agent's strategic priorities.

A full machine-learning evaluation could also be implemented using an efficiently updated neural network (NNUE) which outputs a best move, given a current board state and calibrated through training (Stockfish, 2024).

6.2 Progressive Heuristic Adjustments

Introducing dynamic weighting for heuristics (shifting priorities) based on the game phase could make the agent more adaptive. For example, emphasizing mobility in the early game would preserve flexibility, while prioritizing stability and corner control in the late game would secure the final positions. Alternatively, the square weights of the board could alter as the game progresses. This approach would use the percentage of filled board squares to determine what phase's heuristics / weighting scheme to apply. Although initial experimenting with dynamic weights showed worse performance, this is dependent upon the heuristic, phase definition, and other adjustable factors.

6.3 Improved Move Ordering

The current move-ordering mechanism could be further modified (perhaps a faster, more insightful heuristic exists) to gain even more depth / breadth through pruning. By analyzing more games, we could gain a better understanding of which types of moves (e.g., stability-heavy or high-mobility moves) typically lead to better outcomes and use this information to prioritize moves during search. This could also be done with machine learning classification techniques. Improved move ordering would allow the agent to prune more effectively, closer to b^d time complexity enabling deeper searches within the constraints.

6.4 Leveraging Opponent Behavior

Adapting the agent to recognize and counter specific (popular) opposing strategies could also be considered. For instance, if the opponent favors corner-seeking strategies, the agent could dynamically allocate greater weight to corner defense—and prevent corner captures. Alternatively, against more aggressive opponents (that seek to maximize piece difference and mobility difference), the agent should increase priority to keeping itself mobile (particularly in the early game) (Kukreja, 2013). A defensive approach, prioritizing stability and frontier control. Such opponent-specific adaptations would make the agent more versatile against strong, general-case opponents.

7 Input Credit

We used a combination of ChatGPT and external references. We started by consulting with ChatGPT to brainstorm features and to get us started on a custom agent. The inquiry began as follows: "We are building an Othello AI agent, here is a sample player agent that uses 1-step lookahead and prefers to capture corners, provided to us: `[gpt_greedy_corners_agent code]`. We want to build and implement a minimax player, with alpha-beta pruning and more sophisticated heuristics..." The result of the model developed with input from ChatGPT consistently lost to the random agent, but beat `gpt_greedy_corners_agent`. We then conducted extensive research via online forums and published papers, looking to Kukreja's 2013 C++ heuristics and evaluation function. This method had thorough documentation on game strategy, which seemed promising, along with its wide array of evaluation metrics.

This is the approach that inspired us to implement the stability heuristic and base weights matrices (dependent on board size). We then prompted ChatGPT to convert the heuristic calculations of the C++ implementation into Python, and subsequently incorporated this into our existing Alpha-Beta pruning agent. This is the agent that, after combining other research-backed heuristics, ultimately evolved into ZMIDAB (Zobrist-Memoized, Iterative Deepening, Alpha-Beta Pruning algorithm). Upon meticulous testing, it outperforms prior Alpha-Beta variants (including one with Kukreja's Python-converted heuristics and evaluation function), `gpt_greedy_corners_agent`, on all board sizes. To build ZMIDAB, we also consulted with ChatGPT on how to efficiently store and memoize the visited states, which is what led to the use of Zobrist hashing. The idea to incorporate memoization was inspired by the best performing model from last year, as mentioned in the project spec.

8 References

References

[1] Kukreja, K. *Heuristic Function for Reversi (Othello).* Retrieved from `https://kartikkukreja.wordpress.com/`

[2] OpenAI. (2023). *ChatGPT (December 2023 version)* [Large language model]. Retrieved from `https://chat.openai.com/`

[3] University of Washington. (2004). *O-Thell-Us: An Implementation of Othello with AI.* Retrieved from `https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/O-Thell-Us/Othellus.pdf`

[4] Official Stockfish. (n.d.). *NNUE.* GitHub. Retrieved December 4, 2024, from `https://github.com/official-stockfish/nnue-pytorch/blob/master/docs/nnue.md`