COMP 424 Final Project: Reversi Agent

Mira Kandlikar-Bloch, Jack Parry-Wingfield

November 2024

1 Introduction and Executive Summary

Our Reversi agent uses an implementation of the minimax algorithm, which we have enhanced with alpha-beta pruning and iterative deepening. We chose this combination of algorithmic techniques because it effectively balances decision-making with computational efficiency. Reversi is a zero-sum game, meaning one player's gain directly corresponds to the other's loss. Such a game is well suited for the minimax algorithm, which is designed to handle game strategy by choosing moves that maximize the player's advantage while minimizing the opponent's. Minimax enables our agent to consider the immediate impact of a move and its long-term consequences by simulating the opponent's best possible responses. This foresight is especially critical in Reversi, where early moves often dictate endgame results. We chose to implement alpha-beta pruning to improve computational efficiency. Alpha-beta pruning enhances the minimax algorithm by "cutting off" branches of the game tree that do not affect the outcome, reducing the number of nodes evaluated by our agent. This is particularly advantageous for our algorithm due to the 2-second time limit for each move. Through alpha-beta pruning, we ensured that our agent does not perform unnecessary computation. Its available time is used optimally to find the best move. This, combined with move ordering and a transposition table, was a large factor in our agent's success. To ensure that our agent always makes an informed move, we implemented iterative deepening. We incrementally searched the game tree, increasing the depth until we hit either the time limit, the endgame, or the required depth. We then returned the best move found so far. This mitigated situations where the agent would not have enough time to find a good move in a traditional minimax search. Using an iterative design process and research of relevant game heuristics, we built up our agent's algorithm and evaluation function. Based on the testing we did across boards and the random and greedy_gpt_corners agents, we achieved a strong play quality for our agent. We also found that it was relatively difficult to beat our agent when playing against it. The following sections detail the algorithm specifics and implementation, an analysis of our agent's performance, and possible further improvements.

2 Agent Design and Implementation

2.1 Minimax with Alpha-Beta Pruning

Our Reversi Agent uses a minimax algorithm with alpha-beta pruning and iterative deepening for gameplay. This section outlines the implementation and further theoretical justification for our agent design. In the *step* function of student_agent.py, we initialize the start time, a time limit of 1.9 seconds, the best move as decided by our *order_moves* function (detailed in section 2.5), and depth=1. While our agent is within the time limit, we repeatedly call minimax while incrementing the depth by 1.
Minimax is implemented as a class method for the StudentAgent class. It takes as input the following parameters:

- *Board*: the current board state.
- *Depth*: an integer specifying how deep the agent should explore, initialized as 1.
- Maximizing player: True if the current player is the maximizing player, False otherwise. Initialized as True.
- Player: An identifier (1 or 2) for the agent executing the algorithm

- Opponent: The identifier for the opposing player. Used to simulate the opponent's potential moves and responses.
- Alpha: a value representing the current lower bound of the search, initialized to +infinity.
- Beta: a value representing the current upper bound of the search, initialized to -infinity.
- Start Time: a timestamp indicating when the turn started
- Time limit: a value representing the amount of time the agent has to make a move, in this case 1.9 seconds.

The initial call to the *minimax* function is made in the *step* function. In *minimax*, the current board state is turned into a hash value to serve as a unique identifier for the transposition table. The function then checks the transposition table to determine if the current board state was previously evaluated. If a matching entry is found in the transposition table and is suitable for the current search, the stored evaluation is used directly for the current board state. Further details on the transposition table, its integration into the minimax algorithm, and what is deemed a "suitable evaluation" are outlined in section 2.3. If no suitable entry is found, the function proceeds with the minimax search. It checks whether the current depth is zero, if the game has reached an end state, or if the time limit has been exceeded. In such cases, the board is evaluated using the evaluation function and the result is stored in the transposition table with an "exact" flag before being returned. Otherwise, the function determines valid moves using *get_valid_moves* and prioritizes them using a heuristic-based *order_moves* function as outlined in section 2.5. For each valid move, the function makes a recursive call, swapping between maximizing and minimizing logic, with the depth decremented each time. During these recursive calls, alpha and beta values are continuously updated and branches are pruned when no further exploration is beneficial. After evaluating all moves (or pruning as needed), the evaluation score is stored in the transposition table. Then, the best move and evaluation score are returned to the previous minimax call.

2.2 Iterative Deepening

After each iteration of minimax, if a move is returned, we assign it as the best move. The assumption is that after each iteration with an increased depth, the algorithm has a better idea of the best move, as it can "see" further down the game tree - whatever it returns will be at least as good as the previous move. In this fashion, we use iterative deepening to store the best move so far, while still exploring the minimax tree. Then, if we run out of time during a call to minimax, we can be sure to return the best move found from the previous depth. While iterative deepening is a powerful tool for ensuring the best move within the given time limit, it does come with a trade-off: repeated computations. This redundancy occurs because the algorithm recalculates evaluations at shallower depths multiple times as it incrementally deepens its search. To solve this we implemented the transposition table, which effectively allows our agent to "skip" over computations done in previous iterations, allowing it to reach deeper game states more quickly [8]. The following section outlines the implementation of the transposition table.

2.3 The Transposition Table

We opted to implement our transposition table as a separate class, TranspositionTable, within student_agent.py. Each instance of a TranspositionTable object has a "table," or dictionary, that stores previously computed evaluations of board states. The dictionary uses a unique hash of a board state's string representation as its key, ensuring each specific configuration of the board state corresponds to a single, distinct entry in the table (9). The value associated with each key is a dictionary containing the following information about a board state:

1. **The Evaluation Score** is a numerical score that represents the utility of the given state of the board, based on our evaluation function. Storing this value avoids recalculating the score for the same board state multiple times during the minimax algorithm.
2. **The Search Depth** indicates the depth in the game tree where the board evaluation was performed. We store this value because deeper evaluations generally provide a more accurate assessment of a state's utility. This information allows our agent to decide whether the stored evaluation of the board state is still reliable. If the current search is performed at a depth less than or equal to the depth associated with a stored board state, the agent can safely reuse the evaluation of the board.

In this case, the agent has at least as much information as it would if it recomputed the evaluation score of the board at the current depth. If the stored evaluation score was performed at a shallower depth than the current search, the agent cannot be certain that the stored value of the board state is at least as good as a new evaluation of the state, given the current depth. In this case, the agent ignores the stored value and proceeds to perform a deeper search. Once the deeper evaluation is done, the agent overwrites the old evaluation score with a new, more accurate score for the board state.

3. **The Bound Flag** is the final and critical component of the transposition table, allowing integration with alpha-beta pruning. It indicates to the agent how the evaluation score of a board state should be interpreted. The bound flag takes on one of three possible values: *exact*, *lower bound* and *upper bound*. The *exact* flag means the evaluation score represents the exact utility of the board state according to the evaluation function. The *lower bound* flag indicates the stored evaluation is the minimum score the maximizing player can achieve, and the *upper bound* indicates that the stored value is the maximum score the minimizing player can achieve. The latter two flags are directly tied to alpha-beta pruning: the lower bound is set when the maximizing player finds a score that exceeds or meets the current beta value, causing a beta cut-off, while the upper bound is set when the minimizing player encounters a score less than or equal to the alpha threshold, resulting in an alpha cut-off. These flags enable the agent to efficiently prune branches and refine alpha-beta thresholds. If a lower bound exceeds the beta threshold or an upper bound is below the alpha threshold, the branch is pruned. Additionally, lower bounds can increase alpha, and upper bounds can decrease beta, tightening the search window and enabling earlier pruning. This integration with iterative deepening and alpha-beta pruning enhances the efficiency and accuracy of the minimax algorithm.

2.4 The Evaluation Function

A major and arguably the most important aspect of our agent is the evaluation function. The evaluation function is responsible for assigning a numerical score to each board state at the base case of minimax, representing its utility for the agent. The larger the evaluation score, the better the board state. This score allows the agent to make informed decisions about which moves to prioritize at each stage of the game. Our evaluation function incorporates the following heuristics tailored to Reversi.

2.4.1 Disc Count

In Reversi, the Disc Count can be an important indicator of a player's dominance on the board. It represents the difference between the number of discs controlled by the agent and those controlled by the opponent. A higher disc count may signify an advantageous position, as the player with more discs is closer to winning. However, this heuristic must be used with caution. In early and mid-game scenarios, maximizing the disc count can lead to an unstable position, and make it easier for the opponent to flip many discs. Later in the game, the Disc Count becomes more significant as it directly determines the winner. Because of this, disc count is generally weighted very little in the beginning and mid-game stages, and later at the end.
We calculated the Disc Count as follows:

$$100 \times \frac{\text{player\_discs} - \text{opponent\_discs}}{\text{player\_discs} + \text{opponent\_discs}}$$

This formula normalizes the Disc Count relative to the total number of discs on the board. A positive Disc Count reflects a larger number of discs for the player, while a negative Disc Count reflects a larger number of discs for the opponent. The motivation for this heuristic and the formula are adapted from source (1).

2.4.2 Mobility

Mobility is the relative number of valid moves a player can make given the current board state. A higher number of moves ensures flexibility, allowing our agent to avoid bad plays that could lead to a loss. Mobility is particularly important in the early and mid-game phases, where controlling the board and restricting the opponent's options lead to an advantage and avoid plays that could allow the opponent to take control (4).

We calculate Mobility using the formula:

$$100 \times \frac{\text{player\_moves} - \text{opponent\_moves}}{\text{player\_moves} + \text{opponent\_moves}}$$

A positive mobility score indicates a larger number of moves for the player, while a negative score indicates a larger number of moves for the opponent. The motivation for this heuristic and the formula are directly adapted from source (1).

2.4.3 Corner Control

Corner Control, or the relative number of corners occupied by a player, is one of the most important heuristics in Reversi. Corners are stable, meaning they cannot be flipped once occupied. They also serve as anchors, providing stability to adjacent discs and reducing the opponent's ability to control the board. We calculate Corner Control by summing the number of corners occupied by a player on the current board. Then we use the formula:

$$100 \times \frac{\text{player\_corners} - \text{opponent\_corners}}{\text{player\_corners} + \text{opponent\_corners}}$$

The evaluation function assigns a very high weight to this heuristic, particularly in the mid-to-late game when corners become critical for securing victory. Earlier in the game corners are given less weight as they are often still inaccessible by both players due to the smaller number of tiles on the board.

2.4.4 Corner-Adjacent Penalization

A heuristic closely related to Corner Control, and thus highly significant in the game of Reversi, is the penalization of corner-adjacent tiles. These tiles are disadvantageous because occupying them allows the opponent to gain control of the adjacent corners. Therefore, our evaluation function assigns a high penalty to boards where our agent occupies these squares. The penalty is calculated based on the control status of the adjacent corner. If the adjacent corner is controlled by the player no penalty is assigned, since these tiles no longer pose a risk. If the corner is controlled by the opponent or unoccupied, the penalty is high. We calculate the Corner Adjacent Penalty as:

$$\text{Corner Penalty} = -50 \times \text{Number of player's discs adjacent to non-player occupied corners.}$$

This heuristic is particularly important in the early and mid-game phases. Unlike other heuristics, we chose to penalize the raw count of the player's discs in these positions due to their severe drawbacks. While it can be advantageous if the opponent occupies these tiles, the risk of placing our own discs there is so significant that the penalty is designed to strongly deter the agent from making such moves.

2.4.5 Stability

Stability is defined as the number of a player's discs that cannot be flipped, regardless of subsequent moves by the opponent. Stable discs are advantageous because they provide a secure foundation for future plays and ensure a consistent presence on the board. Stability is particularly critical in the late game, where securing stable discs often determines a game's outcome. Calculating stability is more complex than calculations for other heuristics. To tackle this, we defined the function *count_stable_discs* which takes the current board state and the player's identifier as input.
In the function we initialize a 2D boolean array with the same shape as the board to track the stability of each disc. All entries in this array are initially set to *false*. We then mark the corners occupied by the player as stable. Next, we propagate stability outward from the corners, along rows and columns. For a disc to be stable, it must be connected to a stable disc and form an unbroken chain of discs belonging to the same player anchored to the edge of the board. The *count_stable_discs* function iteratively updates the stability array until no additional discs can be marked as stable. The function then returns the sum of the number of stable discs for the given player. We then calculate Stability as:

$$100 \times \frac{\text{player\_stable\_discs} - \text{opponent\_stable\_discs}}{\text{player\_stable\_discs} + \text{opponent\_stable\_discs}}$$

Our evaluation function generally assigns increasing weight to this heuristic as the game progresses. In the early and mid-game, stability is less critical, as corners have yet to be occupied. However, in the late game, the weight of stability increases significantly, reflecting its importance in determining the outcome. The motivation for using this heuristic as well as the methodology for *count_stable_discs* are adapted from source (1).

2.4.6 Potential Mobility

Potential Mobility is the number of empty tiles adjacent to the opponent's discs. This heuristic evaluates the agent's ability to increase its valid moves in future turns, which expands future mobility and restricts the opponent's options. We calculate Potential Mobility by iterating over the board to identify all empty squares adjacent to the opponent's discs. The formula for this heuristic is:

$$\text{Potential Mobility} = \text{Number of empty squares adjacent to opponent discs.}$$

This heuristic is particularly valuable in the early and mid-game phases, where maximizing future mobility can help the agent maintain flexibility and control over the board. The motivation for this heuristic comes from source (1).

2.4.7 Frontier Discs

Frontier discs are discs adjacent to at least one empty square. These discs are vulnerable for the player because they are more likely to be flipped by the opponent in subsequent moves. We calculate the number of Frontier discs for a player by iterating through all their discs on the board and checking if they are adjacent to any empty squares, using the *count_frontier_tiles* function. To incorporate this heuristic into the evaluation function, we use the following formula:

$$\text{Frontier Score} = -100 \times \frac{\text{player\_frontiers} - \text{opponent\_frontiers}}{\text{player\_frontiers} + \text{opponent\_frontiers}}$$

Note that this formula uses -100 instead of 100 used for previous heuristics because a higher number of Frontier Discs is disadvantageous for the agent. Motivation for this heuristic comes from source (1).

2.4.8 Game Phase and Board Size Adjustments

An important aspect of our evaluation function is weighting the previously outlined heuristics based on the game phase. We implemented a function called *get_game_phase*, which takes the current board state and determines the game phase. If less than one-third of the tiles on the board are filled, the game is classified as the **early game**; if between one-third and two-thirds are filled, it is considered the **mid-game**; and if two-thirds or more are filled, the game is in the **late game** phase.
This classification is critical because the importance of various heuristics changes as the game progresses, according to source (4). Along with heuristic weight changing with the game phase, it also varies depending on board size. We customized the weights assigned to each heuristic based on both the current game phase and the board dimensions. During the early game, mobility and potential mobility were generally weighted more heavily to ensure that the agent prioritized flexibility and restricts the opponent's options. As the game transitions to mid-game, corner control becomes increasingly important. In the late game, the focus shifts toward stability and maximizing the disc count. The optimal weights we found for each board size and game phase are as follows:

- **6x6 Board**:
  - Early weights:

    ```
    weights = {'disc': 2, 'mobility': 5, 'corner': 30, 'stability': 4,
               'potential_mobility': 3, 'x_square_penalty': 30, 'frontier': 0}
    ```

  - Mid-game weights

    ```
    weights = {'disc': 5, 'mobility': 4, 'corner': 50, 'stability': 8,
    'potential_mobility': 3, 'x_square_penalty': 30, 'frontier':0}
    ```

  - Late weights:

```
weights = {'disc': 15, 'mobility': 2, 'corner': 20, 'stability': 15,
'potential_mobility': 0, 'x_square_penalty': 10, 'frontier':0}
```

- **8x8 Board**:
  - Early weights:

    ```
    weights = { 'disc': 5, 'mobility': 15, 'corner':50, 'stability':8,
    'potential_mobility': 15,'x_square_penalty': 200,'frontier': 0}
    ```

  - Mid-game weights

    ```
    weights = {'disc': 10, 'mobility': 10, 'corner': 2000, 'stability': 20,
    'potential_mobility': 10, 'x_square_penalty': 500, 'frontier': 10}
    ```

  - Late weights:

    ```
    weights = {'disc': 30, 'mobility': 5, 'corner': 3000, 'stability': 50,
    'potential_mobility': 0, 'x_square_penalty': 2000, 'frontier': 0}
    ```

- **10x10 Board**:
  - Early weights:

    ```
    weights = { 'disc': 5, 'mobility': 15, 'corner':50, 'stability':8,
    'potential_mobility': 15,'x_square_penalty': 200,'frontier': 0}
    ```

  - Mid-game weights:

    ```
    weights = {'disc': 15, 'mobility': 10, 'corner': 2000, 'stability': 25,
    'potential_mobility': 15, 'x_square_penalty': 2000, 'frontier': 30}
    ```

  - Late weights:

    ```
    weights = {'disc': 100, 'mobility': 5, 'corner': 2000, 'stability': 40,
    potential_mobility': 0, 'x_square_penalty': 3000, 'frontier': 0}
    ```

- **12x12 Board**:
  - Early weights:

    ```
    weights = { 'disc': 5, 'mobility': 15, 'corner':50, 'stability':8,
    'potential_mobility': 15,'x_square_penalty': 200,'frontier': 0}
    ```

  - Mid-game weights

    ```
    weights = {'disc': 10, 'mobility': 10, 'corner': 2000, 'stability': 15,
    'potential_mobility': 10, 'x_square_penalty': 300, 'frontier': 0}
    ```

  - Late weights:

    ```
        weights = {'disc': 50, 'mobility': 2, 'corner': 2000, 'stability': 25,
    'potential_mobility': 0, 'x_square_penalty': 300, 'frontier': 0}
    ```

By tailoring heuristic weights based on both board size and game phase, the evaluation function ensures the agent adapts its strategy to the changing dynamics of the game. This phase-aware approach enables the agent to make contextually relevant decisions at each stage, optimizing performance across different scenarios and improving overall gameplay in Reversi.

2.5 Move Ordering

Alpha-beta pruning is a powerful optimization tool in our agent's algorithm. However, it's utility heavily depends on the order in which moves are evaluated [7]. This is where move ordering plays a critical role in our agent. For each board explored during minimax, we use the *order_moves* function to rank valid moves from best to worst based on a heuristic evaluation. By exploring the most promising moves first, we maximize the pruning potential of the alpha-beta process, ensuring the best moves are never pruned. We defined move order as follows:

- If a move is a **corner move**, it gets a 100000 bonus, as corner moves should always come first.
- If a move is **adjacent to a corner** and the corner is not controlled by the player, it gets a -3000 penalty. Corner adjacent moves should always be considered last.
- If a move is in the **edges** of the board, it gets a 200 bonus. Edge moves are valuable because they can connect with captured corners and quickly become stable.
- We calculate a move's **mobility** by executing the move on a new board and computing the mobility for that board. A 10 * mobility bonus is given to the move.
- The last item we consider is the **number of tiles that are flipped** by a given move. We use the *count_capture* helper function to determine this value. Then 7 * the count capture is added to the move value. However, if the game is in the late stage, 15 * the count capture is added, to encourage moves that maximize discs.

The sum of each of these values gives the value of the move. Moves are then ordered from highest to lowest value for exploration.

3 Analysis of Agent Performance

3.1 Depth and Breadth achieved by the Agent

The depth level achieved by our agent across board sizes displays a clear trend: as the board size increases, the median depth decreases. For a 6x6 board, we observe a median depth of 6–7 across most games. In contrast, an 8x8 board yields a median depth of 5–6, a 10x10 board has a median depth of 4–5, and a 12x12 board has a median depth of 3–4. This suggests an inverse relationship between median depth and board size. This trend is explained by the relationship between board size and the branching factor, or breadth of the game tree. As board size increases, so does the branching factor. For a 6x6 board, the maximum branching factor is ≈ 12, for an 8x8 board it is ≈ 15, for a 10x10 board it is ≈ 23, and for a 12x12 board it is ≈ 30. This increase is expected because the branching factor is equivalent to the number of valid moves, which typically grows with board size. A larger branching factor forces the agent to spend more time exploring the breadth of the game tree, leaving less time to explore deeper levels within the same time constraints, thus decreasing the median depth achieved. This characteristic is inherent to our iterative deepening approach, as an increased breadth of exploration negatively impacts the achievable depth. However, when alpha-beta prunes unpromising branches with the help of our transposition table, we observe increased depths. Pruning allows the agent to allocate more time to deeper exploration by skipping unnecessary computations. During the endgame, the agent consistently achieves a slightly greater depth across all board sizes. This occurs because the number of valid moves decreases significantly at the end, reducing the branching factor. With fewer moves to evaluate, the agent can allocate more computational resources to exploring deeper levels of the game tree.

3.2 Impact of heuristics and algorithmic design

Our most significant design choice was our weighting of heuristics across game phases and board sizes in the evaluation function. Extensive fine-tuning for each board size revealed that prioritizing corners and penalizing squares adjacent to corners were critical factors. Increasing the weight of corners progressively from early to late game proved highly effective, while penalizing corner adjacent squares most heavily in the mid-game helped maintain board control. Mobility and potential mobility were essential in the early game as they provided flexibility for our agent to avoid bad moves. Their importance diminished in the late game as the number of valid moves decreased. Disc count also became more important in

later stages —starting with a low weight minimized opponent flips early, but increasing its weight in the endgame helped secure victories. Similarly, stability was insignificant in the early game, as corners were rarely secured, but became crucial in the mid and late game. Frontier discs, by contrast, had little impact and were excluded from most boards except the 10x10. This is likely because we did not find the correct balance of frontier disk weighting versus other heuristics.
Initially, we implemented alpha-beta pruning alone. Implementing move ordering slightly improved our agent although the impact was not significant. However, once we combined these with the transposition table —one of the final additions to our agent— it provided the computational efficiency needed to search deeper into the game tree. The heuristics and their weights were already effective, however, the addition of the transposition table gave our agent the final push needed to consistently outperform random and greedy GPT agents. This process is intuitive: the ability to "see" further ahead in the game tree in the same amount of time gave our agent a better understanding of the later game, ultimately leading to its overall success.

3.3 Prediction of Win Rates

3.3.1 Against the Random Agent

Throughout this project, we conducted thorough testing of our agent across all board sizes. Specifically, against the random agent, we consistently observed a 100% win rate, playing 100 games per test. Thus, we have sufficient evidence to predict a 100% win rate against the random agent across all boards, barring the rare occurrence of an exceptionally lucky move by the random agent.

3.3.2 Against Dave

To estimate the win rate of our agent against an average player, we considered our experiences playing against it. Across all board sizes, we each played 10 games and found that, on average, our agent achieved a win rate of approximately 90%. Performance did not vary significantly across different board sizes. Therefore, we estimate a win rate of roughly 90% against the average player.

3.3.3 Against Classmates

This comparison is less straightforward. As our agent performed well against the random agent and GPT's greedy agent, we assume that most of our classmates' agents can perform at a similar level. The fact that both of us could occasionally beat our agent suggests it has shortcomings that other classmates' agents may exploit. Nevertheless, based on our testing we are confident that our agent will perform competently, achieving a win rate of approximately 65% to 70%.

3.4 Strengths & Weaknesses

Our Reversi agent demonstrates several strengths. One of its primary advantages is its consistent performance, achieving win rates almost always 100% against random agent opponents on board sizes such as 6x6, 10x10, and 12x12. The agent's evaluation function incorporates critical metrics, including mobility, corner control, stability, and potential mobility, which collectively guide the agent to make informed decisions. Additionally, the implementation of the minimax algorithm with alpha-beta pruning, move ordering, and the transposition table ensures computational efficiency. Another key strength is the agent's ability to adapt dynamically across game phases by applying distinct weightings for early, mid, and late-game strategies, ensuring its decisions remain relevant throughout the match.
Despite these strengths, the agent has notable weaknesses that present opportunities for improvement. As we will explore below, the use of static, manually assigned weights for evaluation metrics limits the agent's adaptability to varying opponents or unforeseen gameplay scenarios. The evaluation function also lacks comprehensive positional awareness, focusing primarily on corners and corner adjacent squares without accounting for the strategic value of edges or central control. Additionally, against the greedy agent, there are some occasional losses. We are slightly puzzled by this - since that agent is deterministic, ours should be too. If we beat the greedy agent on autoplay 20 times for each board, it should remain so for all games, since the greedy agent plays the same each time. This brings to light a possible source of non-determinism in our algorithm, which would require further analysis to pinpoint.

4 Future Improvements

The `evaluate_board` function has significant influence over the performance of a minimax game playing agent such as ours. There are several ideas we had developed for performance enhancements. However, we never had the time to implement these ideas, which we now consider as possible methods for future improvements. Namely, implementing positional weights (i.e. each position on the board has its own weight), tuning the weights through machine learning, prioritizing certain corner adjacent squares over others, and edge control. We visit each of these ideas in detail below.

4.1 Positional Weights

A natural extension of our weight implementations would be to assign **positional weights** to each square, reflecting their importance in different phases of the game.

For example:

- Corners might retain the highest positive weight, as they secure stability.
- Squares adjacent to corners could have high negative weights earlier in the game but diminish in influence later.
- Edges could have higher weights than central positions to encourage the development of stability.

To do this, we could design a **positional weight matrix** corresponding to the board size. For an 8x8 board, this would be an 8x8 grid of weights. Then we could integrate it into the evaluation function by summing over the values of positions occupied by the player and adding it to the score returned by the evaluation function.

4.2 Tuning Weights via Machine Learning

Our current weights are manually assigned based on intuition and testing. Machine Learning could optimize the weights for each heuristic in our evaluation function by training a model on a dataset of game states, with the features being the heuristic values and the target being whether the game was won or lost. The supervised model would learn to adjust the weights of the heuristics to maximize the probability of a win. Once trained, the model's learned weights would replace the manually assigned ones in the evaluation function, allowing the agent to make more effective decisions that are better aligned with winning strategies. This idea is directly adapted from [6].

4.3 Corner Adjacent Square Priority

Squares located adjacent to corners often serve as stepping stones for controlling corners. When no other options are available, taking a square orthogonally adjacent to a corner is usually safer than taking one diagonally adjacent. This is because edge moves can provide some stability while diagonal squares do not – however, this is only true if the relative corner is later occupied by the player[5]. Then, we could implement an ordering heuristic such that if no other moves are available, squares located orthogonally adjacent to a corner are occupied before those diagonally adjacent.

4.4 Edge Control

Edges provide a balance of stability and mobility, making them highly strategic. An agent with strong edge control can constrain its opponent's options while securing stable discs[5]. We considered this heavily and implemented it in our move ordering, but never in our evaluation function. If we were to directly implement edge control into our evaluation function, it could improve our agent's strategy.

5 References

(1) Vaishu, Muthu. *Mini-Project 1: Implementation of the Minimax Algorithm*. University of Washington, 2004, `https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/miniproject1_vaishu_muthu/Paper/Final_Paper.pdf`.

(2) Codecademy. ”AI — Minimax Algorithm.” *Codecademy*, `https://www.codecademy.com/resources/docs/ai/minimax-algorithm`. Accessed 3 Dec. 2024.

(3) Shehu, Amarda. ”Lecture 5: Game Playing (Adversarial Search).” *CS 580 (001): Spring 2018*, George Mason University, 2018, `http://cs.gmu.edu/~ashehu/CS580_Fall18/Lectures/lecture5_gameplaying.pdf`. Accessed 3 Dec. 2024.

(4) Lucas, Melz. *Connect-4 Minimax Algorithm Implementation*. GitHub, `https://github.com/lucasmelz/connect-4-minimax`. Accessed 3 Dec. 2024.

(5) https://othelloacademy.weebly.com/basic.html

(6) https://edstem.org/us/courses/63632/discussion/5803948

(7) COMP424 Lecture 7

(8) https://hackernoon.com/the-methods-we-used-to-solve-othello

(9) https://stackoverflow.com/questions/29990116/alpha-beta-prunning-with-transposition-table-iterative-deepening