,

COMP424 Final Project: A fast, fierce Reversi bot: meet GOAThello

Course Instructor: David Meger and Golnoosh Farnadi
Students: Rafael Reis (261037134) & Lucas Renaudie (261045005)

1. Executive Summary

The strongest algorithm we developed for playing Reversi combines an optimized minimax search with alpha-beta pruning with iterative deepening, using smart move ordering heuristics to look at the most promising moves first, and a complex evaluation function that uses 24 different strategically chosen weights which depend on the game phase (early, mid, late game) [1]. We also used game-state memoization by storing previously evaluated board positions, all while making sure to stay under the 500 MB RAM usage limit. Our agent achieves a high level of play quality, consistently outperforming the random agent (100% winrate across all boards) and greedy corners gpt agent (+99% winrate across all boards). Against an average human player, we expect our agent to win 99% of the time as we have played hundreds of games against it and are not able to beat it.

2. Detailed Explanation of Agent Design

2.1 Minimax Search with Alpha-Beta Pruning

At the core of the agent's decision-making process is the minimax algorithm enhanced with alpha-beta pruning [1]. Alpha-beta pruning is an optimization technique for the minimax algorithm that eliminates branches in the search tree that won't affect the final decision. It keeps track of two values: alpha (the best value the maximizing player can guarantee i.e. us (our student agent)) and beta (the best value the minimizing player (our opponent) can guarantee). By pruning branches where the current node's value cannot improve alpha or beta, we reduce computational effort while preserving the correctness of the result and thus gain a signicant amount of time. [3]

2.2 Iterative Deepening Search with Time Management

In order to stay within the time constraint of two seconds per move we capped the "thinking time" of our agent at 1.95 seconds, i.e. if we ever reach this threshold while inside the step() function, we immediately return the last best found move. This allows us to always search as deep as possible with the time we have.
In order to do this we use the Iterative Deepening Search algorithm (IDS) which dynamically increases the search depth, starting at 1, i.e. looking only one move ahead. This algorithm is very effective when in a situation where search time is limited because it uses results from previous iterations to guide the search. This way, we always have a "good" move ready to be made, even if time runs out during a deeper search.

2.3 Move Ordering Heuristics

To enhance the efficiency of alpha-beta pruning, we implemented several advanced move ordering heuristics in the `order_moves()` function. This essentially allows us to explore the most promising moves in the search tree, leading to using the 1.95 seconds we have in a very optimal way, because we spend more time looking at good moves.
For each possible move, we calculate a score by evaluating various factors. We prioritize moves that align with killer moves from previous searches. These moves are moves that previously led to a pruning cut-off [4]. We also put high priority on potential corner moves. The stability heuristic evaluates how secure the player's discs would be after playing the considered move. The history heuristic rewards moves that have previously improved evaluations. Positional weights, based on predefined values for different board regions, are also included to favor putting our discs in strategic areas of the board (we will expand on this further). Furthermore, the potential number of discs flipped by the move is also considered. After calculating all these scores, the moves end up with a final score each. We then sort them in descending order of calculated scores to facilitate better decision-making during search. All these steps combined lead to a significant time-save on the alpha-beta algorithm because we eliminate large portions of the search tree more effectively.

2.4 Evaluation Function: The core of our agent

At the core of our agent's performance is our evaluation function. It is absolutely crucial for assessing non-terminal game states since we cannot perform a full game tree search in the two seconds that we have. Therefore we must get a "feel" for how good a certain position is in order to make good moves in the allocated time. That is precisely the goal of our evaluation function. It combines several (9) heuristics ([2, 6]), with all but one being weighted differently depending on the game phase (e=early, m=mid, l=late):

1. **Disc Difference (e=5, m=10, l=20):** Measures the difference between the number of discs we have and our opponent has, favoring boards where we have more pieces. A common strategy in othello is to not capture too many pieces in the early stages of the game as the more discs we have, the more options we give our opponent. Towards the end of the game however, this weight becomes significantly larger as disc count is what determines the winner.

2. **Corner Occupancy (e=40, m=60, l=50):** Rewards occupying corner positions that cannot be flipped once captured. This is the underlying heuristic in the GPT greedy corners bot that we beat 99% of the time on all board sizes. Large weights are assigned, especially in the middle game, as the middle game is often where crucial, game-changing corner captures happen.

3. **Mobility (e=10, m=15, l=5):** Evaluates the player's ability to make moves compared to the opponent, favoring higher mobility to maintain control. This heuristic makes less sense to use in the late game, since at this point most of the board is full, and we want to focus on other key aspects such as capturing pieces and getting stable discs (discs that cannot be flipped).

4. **Stability (e=15, m=20, l=30):** Assesses the number of discs unlikely to be flipped, reflecting long-term board security. Stable discs are crucial in order to get winning positions. Through observing hundreds of games, we have noticed that our agent can win games where the score is 3 for us and 30 for the opponent, but our 3 discs are stable whereas our opponent's are not, leading to us flipping the entire game around and winning.

5. **Frontier Discs (e=-5, m=-10, l=-15):** Penalizes discs adjacent to empty spaces, as they are vulnerable to being flipped. Essentially, the more empty spaces we have, the more options our opponent has, which can be especially penalizing in the mid and end game where a single move can flip lots of discs.

6. **Parity (e=0, m=0, l=10):** Accounts for the parity of empty squares, favoring configurations that lead to an advantageous turn order in the endgame. This heuristic makes no sense to use in the early and mid game.

7. **Positional Score (e=15, m=10, l=5):** Rewards control of strategically valuable positions based on a weighted positional matrix that is based on board size. We retrieve these constant, carefully crafted matrices with the function `get_positional_weights()` [5]. We will expand in a later section on how these matrices are built.

8. **Potential Mobility (e=7, m=5, l=2):** Considers our agent's ability to limit the opponent's future moves by increasing our own move options. This heuristic is especially valuable in the early game where mobility is key: we want to maximize the amount of moves we can play while restricting our opponent's options. This heuristic has a similar effect to the disc difference one in the early stages of the game, altough still remaining different.

9. **Forcing Moves:** Adds a high bonus (+1000) if *this* move leads to our opponent having no legal moves, reflecting a dominant position where we get to make the decisions. We have found through testing that this heuristic significantly contributes to capturing corners and solving endgames because we make our opponent run out of moves.

By adjusting the weights of these factors based on the game phase, our agent adapts its strategy as the game progresses. We found that these weights carry extremely well across all board sizes and against multiple different bots.

2.5 Positional Weight Matrices

The positional score we attribute to a position is based on the constant positional matrices we crafted. These matrices attribute a set of different weights to specific important squares. The 8x8 positional matrix is based on a reversi guide we found while researching strategies [5]. Across all board sizes, we attribute:

- +120 to corners as they are the highest value position.

- -40 to X-squares (squares diagonally adjacent to corners) as they can give up corners and positional control.

- -20 to C-squares (squares adjacent to corners) because althought they can give up corners they can give stable edge positions in certain situations.
- +20 to edges next to C-squares as they are hard to flip in the future (stable) and can help to capture corners.
- 15 to squares on the "big diagonals" (corner to corner) diagonally adjacent to X-squares because they can help to capture corners and give center control.
- +5 to edges that don't satisfy the above conditions because edges in general are good, and likely to increase overall stability.
- -5 to the second outer ring (squares 1 square away from edges) because they can lead to our opponent getting access to edges.
- +3 to all the rest.

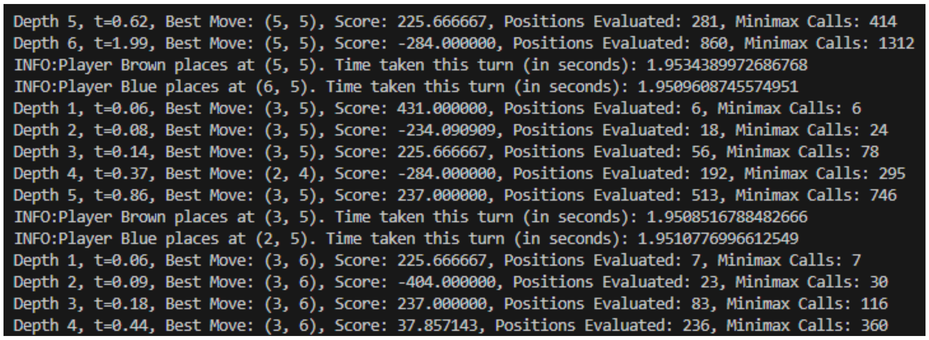
2.6 Game-State Memoization

On top of efficient alpha beta iterative deepening search with effective move ordering and an accurate evaluation function, we maintain a transposition table that stores evaluated board states along with their evaluations and the depths at which they were calculated. This allows us to save computation time if we already evaluated a specific board state. We check at the start of every `minimax()` call if we have a match in our transposition table. If we do, no need to evaluate that position as we already have a score. Initially, we stored evaluations found for specific moves. This didn't work well as there was no associated depth to those moves, leading to less accurate evaluations from shallow searches affecting deeper ones, which is counter productive. We modified the transposition table to store both the score and the depth, ensuring that only evaluations from searches of at least the same depth are reused. The transposition table has a fixed size limit (100,000 entries), and older entries are discarded when the limit is exceeded. This means we balance memory usage and retrieval efficiency.

3. Quantitative Analysis

3.1 Search Depth Achieved

Due to the optimizations in the minimax function and the effective use of alpha-beta pruning combined with advanced move ordering, our agent achieves a good search depth within the time constraints. How far down in the tree we go is heavily related to board size as the branching factor of the 12x12 board is significantly higher than the 6x6 board. Indeed on a 6x6 board there are $3^{36} \approx 7.6 \cdot 10^{17}$ board states (a tile can be empty, black or white so 3 options) and a 12x12 boad has $3^{144} \approx 5 \cdot 10^{68}$ possible board states, not all reachable due to the rules of the game of course. Using print statements we can see how our agent "thinks" i.e. we can print the achieved depth, the best move found so far and its associated evaluation score, the number of leaf nodes evaluated with the evaluation function and the number of calls made to the minimax function (this is essentially the number of nodes visited at every depth). This is what a typical log looks like:



```
Depth 5, t=0.62, Best Move: (5, 5), Score: 225.666667, Positions Evaluated: 281, Minimax Calls: 414
Depth 6, t=1.99, Best Move: (5, 5), Score: -284.000000, Positions Evaluated: 860, Minimax Calls: 1312
INFO:Player Brown places at (5, 5). Time taken this turn (in seconds): 1.9534389972686768
INFO:Player Blue places at (6, 5). Time taken this turn (in seconds): 1.9509608745574951
Depth 1, t=0.06, Best Move: (3, 5), Score: 431.000000, Positions Evaluated: 6, Minimax Calls: 6
Depth 2, t=0.08, Best Move: (3, 5), Score: -234.090909, Positions Evaluated: 18, Minimax Calls: 24
Depth 3, t=0.14, Best Move: (3, 5), Score: 225.666667, Positions Evaluated: 56, Minimax Calls: 78
Depth 4, t=0.37, Best Move: (2, 4), Score: -284.000000, Positions Evaluated: 192, Minimax Calls: 295
Depth 5, t=0.86, Best Move: (3, 5), Score: 237.000000, Positions Evaluated: 513, Minimax Calls: 746
INFO:Player Brown places at (3, 5). Time taken this turn (in seconds): 1.9508516788482666
INFO:Player Blue places at (2, 5). Time taken this turn (in seconds): 1.9510776996612549
Depth 1, t=0.06, Best Move: (3, 6), Score: 225.666667, Positions Evaluated: 7, Minimax Calls: 7
Depth 2, t=0.09, Best Move: (3, 6), Score: -404.000000, Positions Evaluated: 23, Minimax Calls: 30
Depth 3, t=0.18, Best Move: (3, 6), Score: 237.000000, Positions Evaluated: 83, Minimax Calls: 116
Depth 4, t=0.44, Best Move: (3, 6), Score: 37.857143, Positions Evaluated: 236, Minimax Calls: 360
```

Extensive testing through thousands of games across all board sizes yielded the following results:

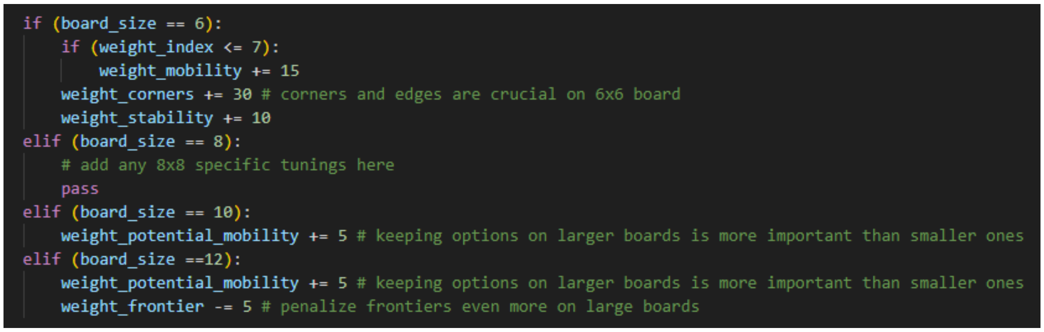
	6x6	**8x8**	**10x10**	**12x12**
Average Depth Acheived	7	5	4	3
Average Breadth (Leaf Node Evaluations)	2k-3k	1k-2k	500-1k	250-750

3.2 Search Breadth

As previously mentioned, at each level of the search tree, we consider all valid moves but prioritizes them using enhanced heuristics. This prioritization effectively reduces the branching factor, allowing the agent to focus on the most promising moves and prune less effective branches earlier. This leads to evaluating a higher number of leaf nodes for every depth of the iterative deepening search, leading to a higher achieved breadth.

3.3 Impact of Board Size and Time Constraints

Our design accounts for different board sizes through our constant positional weight matrices and board-specific tuning of some crucial weights. For example the weight matrix for the 6x6 board is slightly different than the ones for the 8x8, 10x10, and 12x12 boards. Also, we fine-tuned some specific, crucial weights based on board size:



```python
if (board_size == 6):
    if (weight_index <= 7):
        weight_mobility += 15
    weight_corners += 30 # corners and edges are crucial on 6x6 board
    weight_stability += 10
elif (board_size == 8):
    # add any 8x8 specific tunings here
    pass
elif (board_size == 10):
    weight_potential_mobility += 5 # keeping options on larger boards is more important than smaller ones
elif (board_size ==12):
    weight_potential_mobility += 5 # keeping options on larger boards is more important than smaller ones
    weight_frontier -= 5 # penalize frontiers even more on large boards
```

We found through testing that these small additions lead to a significantly stronger agent. We consider the same amount of moves for the max and min player, because we tried an implementation where we were considering less moves for the min player and got dissatisfying results.

3.4 Predicted Win Rates

Based on testing and performance analysis over thousands of games that we let run overnight using multiprocessing scripts, the predicted win rates are:

- **Against Random Agent**: Approximately 100% win rate across all board sizes, because we believe in our agent's strong strategic play and ability to punish mistakes. We also ran thousands of test games against this agent and consistently beat it 100% of the time.

- **Against Dave (Average Human Player)**: Expected win rate of around 100%, because our agent has a very deep look-ahead compared to an average human and uses advanced heuristics.

- **Against Classmates' Agents**: We expect competitive performance due to the combination of optimized search techniques and a comprehensive evaluation function. We believe a win-rate of approximately 75% is a solid estimate.

4. Advantages and Disadvantages of the Approach

4.1 Advantages

- **Alpha-Beta vs MCTS**: We hesitated between making an Alpha-Beta based agent versus an MCTS based agent. We ended up going with Alpha-Beta because Reversi is a game with a relatively low branching actor (compared to more complex games like chess for instance), especially on the smaller sized boards (i.e 6x6 and 8x8). (A smaller branching factor allows for the Alpha-Beta search to go deeper, therefore seeing more moves ahead).
  At each move on an 8x8 board, there are rarely ever more than 15 legal options, compared to chess where there can often be more than 50 (in fact, Reversi's branching factor on an 8x8 board is estimated at 10, whereas chess' branching factor is estimated at 35).
  While the branching factor does go up significantly as the board size increases to 10 or 12, we considered that overall the branching was small enough for us to be able to implement a strong alpha-beta agent, provided that we were able to come up with a solid evaluation function.

- **Adaptability and Strategic Depth**: Dynamic adjustment of heuristic weights and positional weights enables the agent to adapt its strategy throughout the game.

- **Optimized Transposition Table Usage**: Using transposition tables significantly improves efficiency and performance. The table stores previously computed evaluation values for specific board configurations, allowing the algorithm to avoid wasting time on redundant computations. (If the algorithm encounters the same configuration again, it can reuse the stored value instead of recalculating it.) By reducing redundant calculations and pruning unnecessary branches, transposition tables allow the algorithm to allocate resources more effectively, searching deeper in the game tree within the same computational time.

4.2 Disadvantages

- **Suboptimal Heuristic Function Weights**: The heuristic functions used in our alpha-beta algorithm's evaluation function, as well as the heuristics used to order the search space from best to worse moves, are both certainly weaknesses in the optimization of our agent.
  This is mainly because the weights used in these heuristics were tuned manually, which limits the accuracy of their optimization. And while we fine tuned them by testing them over thousands of games against the gpt greedy and random agents, as well as against ChatGPT-generated alpha-beta and MCTS algorithms, their values probably remain a very weak estimate of what the true set of globally optimal weights actually is.
  Another weakness is our restricted number of heuristics. We use 9 different heuristics for the evaluation function, and 6 for the move-ordering-based pruning, which is not bad, but we are well aware that there are many specific scenarios that could benefit from additional heuristics (see the "Future Improvements" section for examples).

- **Computational Complexity**: Advanced features like depth-aware transposition tables and detailed evaluation functions introduce additional computational overhead.

- **Time Constraints Limitations**: Despite optimizations, the strict per-move time limit may still prevent the agent from exploring as deeply as desired in complex situations.

5. Future Improvements

5.1 Implementing MCTS

Given the weaknesses of our agent listed above, it is entirely possible that a strong MCTS agent could be better overall. We would have liked to try to implement one, and compare it with our current agent.

It is possible that the MCTS would have resulted in higher win rates on bigger board sizes, and Alpha-Beta would have resulted in higher win rates on lower board sizes. This is because Alpha-Beta gets much weaker as the board size increases, as it is not able to go as deep in its search due to the increased branching factor. A possible alternative would be to use Alpha-Beta on the smaller boards and switch to MCTS for the bigger boards.

With more time, we would ideally have created a strong MCTS agent, and tested which configuration resulted in the highest win rates. It is certainly possible that a strong MCTS would have resulted in higher win rates on all board sizes, and we would really have liked to confirm or deny this.

5.2 Implementation of a Genetic Algorithm to Optimize Weights

Currently, the agent uses manually assigned weights for the heuristics functions. These weights were thoughtfully designed, as we optimized them by fine tuning over thousands of games against the gpt greedy and random agents, as well as against ChatGPT-generated alpha-beta and MCTS algorithms.

However, our manual tuning limits the accuracy of the optimization of the weights. We considered applying Genetic Algorithms to achieve a much more precise optimization, however, given our limited time and resources, we ran into the counter fitting issue. Because we only had less than a dozen agents to put against our own agent in the genetic algorithm, the algorithm would optimize the weights to maximize the win rate against these specific agents, and this potentially results in globally less optimal weights (i.e. weights that result in lower win rates when tested against a large number of agents).

With more time, we would generate hundreds of strong agents, using AI for instance, or by using the student agents from each team in comp 424, (or both!), and apply the genetic algorithm against all of them. This would ensure a more accurate estimate of the globally optimal weights for our heuristics. In this manner, we would avoid the counter fitting issue as the games would be played by a large number of different agents.

5.3 Pruning the Search Space

We also tried reducing the search space by pruning the set of legal moves at each move. Our main approach was to consider only the best 10 moves (*), estimated using our move-ordering heuristics. This would allow the Alpha-Beta search to go deeper (i.e. see more moves ahead), as it would spend less time on each individual move (because at each move, it would have less options to consider thanks to the pruning).

The problem with this method is that the pruning was based off our heuristics' estimate of the move-ordering, and could therefore potentially miss better moves. After some testing, we found that the version of our agent with the pruning was often worse than the version without it. The reason for this is probably that our heuristics for move-ordering were just too weak, and needed more strengthening. We would have loved to have the time to optimize our move-ordering heuristics (potentially using a genetic algorithm, mentioned above), to be able to fully implement pruning.

(*) We also tried connsidering only 5 moves, and we also tried considering a number of moves proportional to the size of the board, but both of these methods resulted in significantly lower win rates. Again, this is most likely due to our weak estimations for move-ordering, and optimizing the move-ordering heuristics would have allowed us to fully explore the potential of pruning.

5.4 Expanding the Heuristics

With more time, we would also have liked to test out more heuristics. Our current heuristics form a solid general base for the evaluation function, but certain specific scenarios required more advanced heuristics. Examples of this include an opponent mobility limitation heuristic, which measures the potential to limit the opponent's future moves, rather than the maximizing of your own mobility. Another example is a threat detection heuristic, which would identify moves that allow the opponent to capture strategic positions (e.g., corners or critical edges) in the next turn or two.

5.5 Machine Learning Integration

Integrating machine learning models to adjust heuristic weights dynamically based on the opponent's play style or to predict more effective moves could improve the agent's adaptability and performance. This requires neural networks / reinforcement learning, which we could've tried to implement, but this is very time consuming, and we would need datasets to train the models on.

6. Conclusion

In conclusion, our agent demonstrates very strong strategic and tactical gameplay in Reversi, effectively utilizing a combination of optimized alpha-beta pruning, iterative deepening search, advanced move ordering heuristics, and a carefully crafted evaluation function. By using game-state memoization and game-phase-specific heuristic weights, our agent adapts to different game phases and board sizes, achieving nearly perfect win rates against various opponents like greedy corners gpt and random over hundreds of games. Extensive testing confirmed its superiority against baseline agents like random and greedy corners, with nearly perfect win rates across all board sizes.

However, we recognize that there is room for improvement. Expanding the heuristic set and applying machine learning techniques fine-tuning evaluation weights are all promising avenues for future development. Additionally, the exploration of Monte Carlo Tree Search (MCTS) as an alternative or hybrid approach could enhance the agent's performance, particularly on larger board sizes with higher branching factors.

Overall, this project shows how well-thought-out algorithms and heuristics can lead to a highly effective and competitive game-playing agent. With more adjustments and more time, our agent could serve as a benchmark for evaluating other strategies and even compete effectively in broader competitions.

References

[1] Concepts covered in the COMP 424 class curriculum

[2] Samsoft: A guide on how to play othello

[3] Using minimax and alpha-beta to play othello

[4] Killer Move Heuristic

[5] Weighted matrices for othello

[6] Othello Academy: Strategy Guide for Othello