A Minimax-Based Reversed Othello Agent

Zean He, *12011323*

Abstract—As gameplay is one of the traditional topics among artificial intelligence which has been widely adopted through computing techniques, Othello is also one of the famous game playing, which was widely researched on informed searches. This project implements the Reversed Othello, the reversed terminating evaluation version of conventional Othello, based on a heuristic search. Additional comparisons and analyses of the developed implementation have been done to prove its effectiveness and other methods such as MCTS have been tried for comparison and future analysis.

1 Introduction

Othello, an incarnation of the simple old Japanese board game of Reversi[1], is a deterministic zero-sum board game with perfect information. Its rule is quite simple: with a fixed initial setup of the board, two players (black and white) take turns to drop disks on the board, when two disks of the same color block the opponent's disks in a line, all the opponent's disks will be turned into the other color. The game terminates when both players have no valid place to drop their disks. While a typical Othello judges the player with more disks left on the board wins, our project, reversed Othello, considers the one with fewer disks to be the winner.

To win the game of reversed Othello in this project is not as simple as reversing the decision process of a traditional Othello, apart from designing different heuristic components, the interaction between different components also needs to be controlled by carefully weighted combinations.

As a zero-sum game, it offers inspiration for real-world problems such as the stock market and political issues, the research outcome also contributes to other games like Go and Gomoku. While the decision process of Othello can be facilitated by heuristic and meta-heuristic components and gameplay, it is also possible to model these problems using a similar but with a larger scale of heuristic evaluations. The rule of flipping disks in Othello games is also philosophical, the wisdom of "using retreat as advance" may guide our attitude towards our lives.

Due to the special rule of Othello, it has been widely researched by computer scientists [1]. Though many efficacious algorithms and heuristic pipelines have been proposed for Othello, the power lies behind the gameplay is still worth studying and inspires us for real-world problems. In the era of deep learning, we still want to explore some simple and interpretable algorithms for gaming problems. As a widely researched gaming problem, the game of Othello, or reversed Othello, is an ideal entry point for us to take a glance at the beauty of artificial intelligence.

2 Preliminary

In this section, we formulate the game of Othello, we will also introduce the terminologies and notations in this report.

1. https://en.wikipedia.org/wiki/Reversi

2.1 Formulation

The game of reversed Othello can be formulated as a Markov Decision Process (MDP) [2]. Such MDP can be specified by a tuple $(\mathcal{A}, \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$, where

- $\mathcal{S}$ is the state space, having $|\mathcal{S}| < \aleph_0$. For the board game of (reversed) Othello, we can use a matrix to represent a state (typically, $n = 8$):

$$\begin{pmatrix} c_{00} \in \mathbb{C} & \cdots & c_{0n} \in \mathbb{C} \\ \vdots & \ddots & \vdots \\ c_{n0} \in \mathbb{C} & \cdots & c_{nn} \in \mathbb{C} \end{pmatrix}_{n \times n} \tag{1}$$

- $\mathcal{A}$ is the action space, for a state $\mathcal{S}$, given the color of self is c_s, we again can use a matrix to represent the action space:

$$\left(\ c_{ij} = 0 \wedge \cup_{d \in \mathbb{D}} \textsc{Line-Up}((i,j), d, c_s)\ \right)_{n \times n} \tag{2}$$

  here we use a relation **line-up**: $position \times \mathbb{D} \times \mathbb{C} \rightarrow bool$ to represent that from the position, there's a disk of the specified color in the line toward the $direction$, separated by at least 1 disk of the opposite color. Note that in some state $\mathcal{S}_{end}$, the corresponding action space $\mathcal{A}$, with either $c \in \mathbb{C} = 1$ or $c = -1$, we have $\mathcal{A} = \mathbf{0}$, say, both side has no valid position to drop the disk, we call such states as the terminal states that the game ends. We can then compute the score of the state:

$$s = \sum_{i=0}^{\text{row}} \sum_{j=0}^{\text{col}} c_{ij} \tag{3}$$

  the result of the game is

$$\begin{cases} \text{Black win}, & s > 0 \\ \text{Game draw}, & s = 0 \\ \text{White win}, & s < 0 \end{cases} \tag{4}$$

- $\mathcal{P}$**,** $\mathcal{R}$**, and** γ are the probability matrix of state transference, reward function, and discount factor. In this project, we will not distinguish them, but treat them as the decision policy of the agent.

2.2 Terminologies and Notations

- **Coordinate** The chessboard of Othello is typically a square, we mark the vertical axis in numbers $(1, 2, \cdots)$, and the horizontal axis in alphabets $(a, b, \cdots)$. The most common chessboard size is 8, and we will use this size in the rest of this report, if not explicitly specified.

TABLE 1
Notations used in this report.

Notation	Description
$\mathbb{C} = \{0, -1, 1\}$	Possible colors for grids on the board ({none, black, white} corresponding)
$\mathbb{D} = \{(i, j) \mid i \in \{-1, 0, 1\}$ $, j \in \{-1, 0, 1\}\} \backslash (0, 0)$	Directions on the board
$c_{ij} \in \mathbb{C}$	The color of the disk on position (i, j)
b	Max branching factor of the search tree
m	Max depth of the state space
d_{self}	Set of self's disks in the board
d_{oppo}	Set of opponent's disks in the board

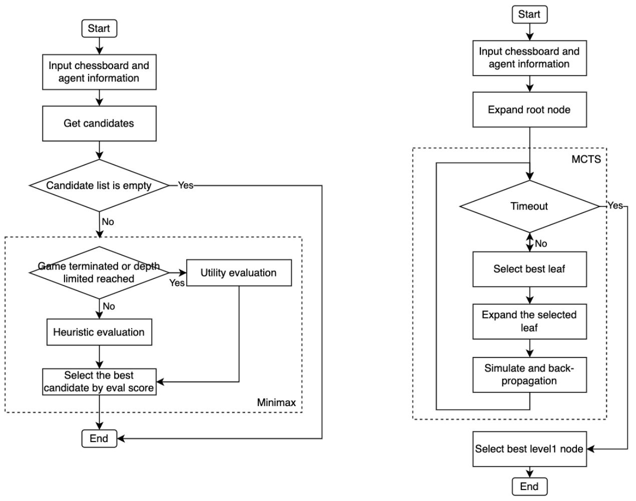

Fig. 1. General workflow for Minimax (left) and MCTS (right).

- **Stable disks** are disks that will not be flipped no matter how other disks drop in the rest of the game.
- **X-square** are positions that obliquely adjacent to the corner ($b2, b7, g2, g7$ for 8×8 chessboards).
- **Stoner Trap** is a move sequence that forces corner-eating controls the main diagonal by playing X-square first, then attacking the corner so that if your opponent tries to defend the corner, he will flip the X-square.

General notations are listed in Table. 1, while some notations appear in single equations may be listed just below their first appearance for a better understanding.

3 Methodology

This section contains two algorithms that the author has explored: the minimax algorithm with alpha-beta pruning, and the Monte Carlo tree search (MCTS). They share the same set of utility functions, while their policies for determining the best candidate differ.

3.1 General Workflow

Referring to Fig. 1, the general workflow of dropping a disk by the agent could be divided into three stages: *input*, *evaluating actions in $\mathcal{A}$*, and *select the best candidate as output*. In this project, the author applied minimax search and MCTS to the second stage.

To be more detailed, in the second stage: (1) the minimax way needs a manual decided depth limit to ensure the search can be finished in a reasonable time, note that one needs to make a tradeoff between the time cost and the precision of the search; (2) the MCTS way will first take most of the time (e.g. 95% of the time limit) to construct the game tree, then use the left time to select the best choice that has the highest win rate, and add the choice to the agent's candidate-list.

3.2 Algorithms and Model Design

3.2.1 Minimax with Alpha-Beta Pruning

We start with the well-known minimax algorithm. In this algorithm, the *Max* player (self) tends to maximize the score, while the *Min* player (opponent) minimizes it.

Algorithm 1 Minimax Search

```
Require: State $state$: the current chessboard
Ensure: Action $action$: the best candidate under the policy

  function MINIMAX($state, depth_{\max}$) returns an action
      return $\arg\max_{a \in \text{ACTIONS}(state)}$ \
              MAXIMIZE(RESULT($state, a$), $depth_{\max}$)

  function MAXIMIZE($state, depth$) returns a utility value
      if TERMINAL-TEST($state$) or $depth \leq 0$ then
          return UTILITY($state$)
      $v \leftarrow -\infty$
      for $a \in$ ACTIONS($state$) do
          $v \leftarrow \max(v$, MINIMIZE(RESULT($state, a$), $depth - 1$))
      return $v$
  function MINIMIZE($state$) returns a utility value
      if TERMINAL-TEST($state$) or $depth \leq 0$ then
          return UTILITY($state$)
      $v \leftarrow \infty$
      for $a \in$ ACTIONS($state$) do
          $v \leftarrow \min(v$, MAXIMIZE(RESULT($state, a$), $depth - 1$))
      return $v$
```

The bare minimax search is as complex as a depth-first search, with a time complexity of $\mathcal{O}(b^m)$. However, considering the time restriction, such algorithm is not practical, since it discovers some nodes that are not possible to contribute a better score. To be more specific, we add a parameter $depth_{\max}$ to limit the search depth (m in the term of time complexity) of minimax, such that the search could be ended in a reasonable time and return the partial-best decision before timeout. The $depth_{\max}$ we can set is quite small in practice (we will describe the detail later).

Note that when doing the terminal test, we may find the state is a terminal state that both sides have no place to drop. In this term, we can do special judges that treat the result as strong information, and the utility function could return some special numbers, like $\pm\infty$.

To solve this problem, we introduce two bounds, α and β, prunings can be performed when at least one possibility has been found that proves the move to be worse than a previously examined move [3], say, $\alpha \leq \beta$.

The time complexity of minimax search with Alpha-Beta pruning varies from an ideal $\mathcal{O}(b^{m/2})$ to the worst $\mathcal{O}(b^m)$, which is affected by the move ordering [3], which will be discussed in Section 4. Compared with the bare minimax, the $depth_{\max}$ could be set for alpha-beta pruning is about 1 time larger (this parameter was omitted in Alg. 2).

Algorithm 2 Minimax Search with Alpha-Beta Pruning

Require: State *state*: the current chessboard
Ensure: Action *action*: the best candidate under the policy

```
function ALPHA-BETA-SEARCH($state$) returns an action
    $v \leftarrow$ MAXIMIZE($state, -\infty, \infty$)
    return the action $a$ in ACTIONS($state$) with value $v$

function MAXIMIZE($state, \alpha, \beta$) returns a utility value
    if TERMINAL-TEST($state$) then return UTILITY($state$)
    $v \leftarrow -\infty$
    for $a \in$ ACTIONS($state$) do
        $v \leftarrow$ max($v$, MINIMIZE(RESULT($state, a$), $\alpha, \beta$))
        if $v \geq \beta$ then return $v$
        $\alpha = \max(\alpha, v)$
    return $v$
function MINIMIZE($state, \alpha, \beta$) returns a utility value
    if TERMINAL-TEST($state$) then return UTILITY($state$)
    $v \leftarrow \infty$
    for $a \in$ ACTIONS($state$) do
        $v \leftarrow$ min($v$, MAXIMIZE(RESULT($state, a$), $\alpha, \beta$))
        if $v \leq \alpha$ then return $v$
        $\beta = \min(\beta, v)$
    return $v$
```

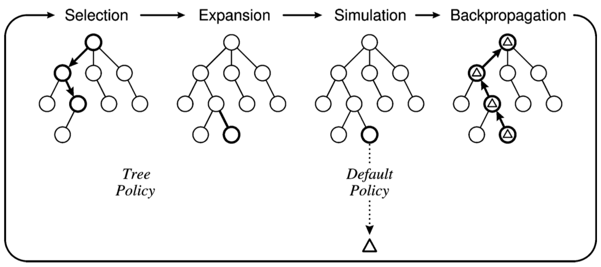

Fig. 2. One iteration of the general MCTS approach [4].

Finally, as an informed search technique, the core of minimax search is a *heuristic function* that takes a state (chessboard) as input, and outputs a score for the state. The agent designer is responsible to provide a reasonable one, which requires domain knowledge. For the heuristic function designed in this project, we refer to Section 3.3.

3.2.2 *Monte Carlo Tree Search*

MCTS is a family of heuristic algorithms that require only the basic rules of the game – the agent designer could be a terrible player. A typical MCTS is performed by firstly repeating **selecting**, **expanding**, **simulating**, and **back-propagating** (as shown in Fig. 2), then, after a period of time, we consider the result is convergent and can thus access the optimal result [4].

The MCTS algorithm tested in this project was the upper confidence bound for trees (UCT), which is an optimized search algorithm that applies upper confidence bounds (UCB) to replace the in-tree search in the stage of selection. In each iteration, a child node j is selected to maximize

$$\text{UCT} = \bar{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}} \tag{5}$$

where $\bar{X}_j$ is the average reward (say, win rate) from child j, n is the number of times the current (parent) node has been visited, n_j is the number of times child j has been visited, and $C_p > 0$ is a constant that decides the MCTS prefers more on exploration or exploitation. By applying UCB, the MCTS can stop at any time and return the local best choice.

Algorithm 3 The UCT Algorithm

Require: State *state*: the current chessboard
Ensure: Action *action*: the best candidate under the policy

```
struct MCTS-NODE
    $parent$: MCTS-NODE
    $children$: LIST⟨MCTS-NODE⟩
    $state$: CHESSBOARD AND COLOR
    $w$: NUMBER                    ▷ ♯ times this sub-tree has win
    $n$: NUMBER            ▷ ♯ times this sub-tree has been visited

function UCT-BUILD($state$) returns root node
    $root \leftarrow$ CREATE-NODE($state$)                ▷ $parent$ is nil
    for $a \in$ ACTIONS($state$) do
        $node \leftarrow$ CREATE-NODE($state, root$)     ▷ $parent$ is $root$
        appends $node$ to $root.children$
        BACK-PROPAGATION($node$, SIMULATE($node$))
    while not timeout do
        $best \leftarrow$ SELECT($root$)
        EXPAND($best$)
        for $child \in best.children$ do
            BACK-PROPAGATION($child$, SIMULATE($child$))
    return $root$

function SELECT($node$) returns leaf node with largest UCB
    while $node.children$ is not empty do
        $node \leftarrow$ max($node.children$, sort by UCB)
    return $node$
function EXPAND($node$)
    if ACTIONS($state$) then
        $chance_{\text{oppo}} \leftarrow$ create node with the same chessboard,
but the opponent's color
        appends $chance_{\text{oppo}}$ to $node.children$
        return
    for $a \in$ ACTIONS($node.state$) do
        $cs \leftarrow$ CREATE-NODE(RESULT($node.state, a$), $node$)
        reverses the color of $cs$
        appends $cs$ to $node.children$
```

After the UCT has been constructed in a limited time, we can then access the first level of UCT and select the node j with maximum $\bar{X}_j$ as the action.

3.3 Evaluation Terms

As previously mentioned, minimax search requires a heuristic function to evaluate the state for each result of the action, which generally is a weighted sum of several evaluation terms. We will introduce several terms proposed by Barber *et al.* [5] and Sannidhanam & Annamalai [6] that are applied to this project.

3.3.1 *Positional Preference*

The rule of flipping disks is both an enemy and a friend of (reversed) Othello players. In reversed Othello, we tend to let the opponent has as many stable disks as possible, an intuitive way is to set the weight of positions to let the agent has "preference" of a position – it will try to drop disks on the position with high weigh once it is possible. We can therefore perform Stoner Trap or some other actions.

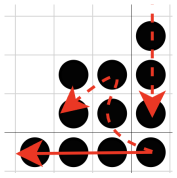

Fig. 3. Right lower corner of an example chessboard, we only count the stable disks in the edge, and continued stable disks from the corners

Provided a weight matrix $\mathbf{W} = [w_{ij}]_{n\times n}$ which is of the same size as the chessboard, we have

$$score_{pos} = \sum_{i=0}^{\text{row}}\sum_{j=0}^{\text{col}}(w_{ij}c_{ij}c_{\text{self}}) \tag{6}$$

where $c_{ij}, c_{\text{self}} \in \mathbb{C}$, the multiplied c_{self} ensures that self disks contribute to the evaluation term, while the opponent takes negative effects on the term.

3.3.2 *Drops Parity*

The parity of the self's disks is also an intuitive way to evaluate the state. Yet we need to consider the possibility of flipping all disks in an enclosed area at one drop.

$$score_{parity} = |d_{\text{self}}| - |d_{\text{oppo}}| \tag{7}$$

where d_{self} is the set of disks in self player's color, and d_{oppo} is the set of disks of the opponent's.

3.3.3 *Stable Disks (Partial)*

One drop that will cause an area of self's disks to become stable disks is certainly a bad choice, vice versa. We may thus static the stable disks of the result of an action and consider such effect in the heuristic function.

Note that it is a bit time-costing to find out all stable disks, we only count part of the stable disks, which is a tradeoff decision of precision and computation cost, take Fig. 3 as an example.

3.3.4 *Aggregation*

By competing with the top agents in the OJ platform, we can observe that they tend to drop disks in a gathered area, instead of splitting the disks everywhere on the board. By considering the average distance of the self's disks, we can let the agent drop into a cluster that is expected to be flipped by the opponent at once.

$$\bar{d}_{\text{self}} = \frac{1}{|D_{\text{self}}|}\sum_{d\in D_{\text{self}}} d \tag{8}$$

$$score_{agg} = \overline{\text{L2-distance}} = \frac{1}{|D_{\text{self}}|}\sum_{d\in D_{\text{self}}}(d-\bar{d}_{\text{self}})^2 \tag{9}$$

3.3.5 *Mobility*

Mobility is the cardinality of action space (valid positions for a player c under a state s). We strive to minimize the opponent's mobility, while making our own mobility large. The difference in mobility can reflect the agent's control of the game.

$$\text{mobility score} = |\mathcal{A}_{\mathcal{S},c_{\text{self}}}| - |\mathcal{A}_{\mathcal{S},c_{\text{oppo}}}| \tag{10}$$

3.3.6 *Weighted Sum*

The heuristic function evaluates the state by considering all the aspects above, which requires the designer to give a proper set of weights to make the best overall decision. The final score $\mathcal{H}$ can be calculated as

$$\mathcal{H} = \mathcal{W}^T \cdot \textbf{scores} \tag{11}$$

where **scores** is the vector of scores listed above, and $\mathcal{W}$ is the vector of weights assigned to each aspect. Specially, we can let $\mathcal{W}_0 = 1$ to reduce one dimension.

3.3.7 *Different Weights for Different Phases*

A game of reversed Othello can be divided into three phases depending on the number of steps (the percentage of disks on the board). Actually, a good agent should focus more on different aspects in different phases. We therefore consider adjusting the weights to make them different between turns of the game. This would require us to provide three different $\mathcal{W}$ for one game.

4 Experiments

4.1 Setup

To test our agent locally, a game platform is needed. A Python script that can import two versions (may be the same, or may one player to be human) of agents was used to play reversed Othello locally. It can print and store the chessboard step by step, it can also manage the parameters (weights) and static their results.

With such platform, I generated most part of the data by playing with my different versions of agents, as well as playing with others on the OJ platform. Moreover, I downloaded some data from BotZone[2] and analyze them to deliver better parameters.

The configurations of the test environment are listed in Table. 2.

TABLE 2
Configurations of the test environment.

System	Hardware	Environment
macOS 12.5	Apple M1 Pro (aarch64) 16GB RAM	Python 3.9.13 NumPy 1.21.5 Numba 0.56.2

4.2 Results

4.2.1 *Running Time*

The running time of MCTS is mainly affected by the speed of simulations. Using cProfile to analyze the function stack, we can find the most time-costing functions are finding the candidate list and applying changes. By applying numba to these two functions, the speed of simulating was boosted by about 260% (as Table. 3 shows).

With numba enabled, we then benchmark the minimax strategy. The running time of minimax search with alpha-beta pruning is mainly affected by both the efficiency of the heuristic function and the rate of pruning.

2. https://www.botzone.org.cn/globalmatchlist?game=53e1db360003e29c2ba227b8

TABLE 3
Benchmark results. With board size $= 8 \times 8$, timeout $= 5s$ (assigned $4.9s$ to build UCT), simulations start from several randomly picked chessboards from the downloaded log.

MCTS	♯ Avg.Simulation
Base	425
njit(cache=True, fastmath=True, inline='always')	1551

We did benchmarks on the performance of alpha-beta pruning with different orderings. Take the positional preference as an example, which was applied to the final version of code:

TABLE 4
Benchmark results. With board size $= 8 \times 8$, $depth_max = 4$, simulations start from several randomly picked chessboards from the downloaded log. The result supports our theory in Section. 3.2.1.

Ordering	♯ Avg.Runtime
None (by the order of candidate list)	77.5 ms ± 1.22 ms
Positional Preference	33.6 ms ± 1.83 ms
Reversed Positional Preference	90.9 ms ± 4.08 ms

As for the selection of evaluation terms, it was performed by checking the log downloaded from the OJ platform. It is another tradeoff between a more comprehensive evaluation and the computation cost. The final version with all evaluation terms mentioned in Section 3.3, with $depth_{max} = 4$, generally does not timeout the $5s$ time limit, or may have $1 - 3$ steps timeout out of the total 30 steps.

4.2.2 Optimality

Comparing MCTS with the minimax agent, it is disappointing to conclude that MCTS's performance in the game of Othello was *awful*. While the UCT simulates over 1500 times every step, it may be still not enough to find out the tricks, such as Stoner Trap, and will be easily tempted to drop on the corners.

As for the minimax agent, it's hard to assert one agent is taking the optimal decision, but what can say is, the agent is not un-optimized. From one inspection, in the supervised $20+$ turns played on the OJ platform, the agent seldom steps to corners, and for the most won turns, the agent drop disks to the central area, as expected, and had them flipped at the last step. Another inspection is that, compared with the milestones I saved before (their weights were all finely adjusted, and tested on the OJ), the latest version could always win over the old one. As Table. 5 shows.

TABLE 5
Comparing the latest agent with previous milestones, numbers are marked in the format {black:white}.

Opponent (white)	Final Count	Flip Count
Position	13:51	154:103
Position + Mobility	10:54	163:113
Position + Mobility + Stability + Drops Parity	23:41	145:127

We may also notice an interesting fact that the winner always has a larger amount of flipping operations.

4.3 Analysis

The experiments support that the agent is getting better when developing and adding evaluation terms. The agent was ranked 15, with a score of 210, before the platform was shut down. The overall performance of the agent is satisfying.

From the above experiments, they reveal that the involvement of the heuristic components such as mobility, stability, and drop parity do decrease their performance lag with the final versions, with a decreasing tile count as well as less disk flip. Though the milestones with only position and mobility reveal worse results, this may due to the negative effect of it during the early stage of the game, and this is solved with the weighted heuristic components and optimized parameters in the later versions.

Furthermore, from the average runtime of alpha-beta pruning in the context of random ordering or ordered by positional preference matrix, we can see a considerable reduction in runtime corresponding to the correctly ordered version. This verifies the effectiveness of the weighted matrix and the feasibility of pre-ordering before the minimax search and boosts the overall performance.

5 Conclusion

This project targets at the implementation of traditional gameplay together with some heuristic and meta-heuristic search, utilizing the power of game playing and some domain knowledge of the game of Othello with the expression of heuristic components. Apart from trying to achieve a better performance, a detailed discussion of the effect of each heuristic component and also the interaction between the linear combination of them are followed and inspire the optimization of the hyperparameters.

Apart from the theoretical analysis of the heuristic components, a more practical and effective way of fine-tuning the parameters is building a self-playing platform to examine the effect of different sets of parameters, or say reinforced learning (RL). Also, crucial steps from online gaming logs are used to evaluate the rationality of each decision.

Future works could be mainly divided into two parts, namely the optimization of parameter tuning of minimax search, and the pipeline design of MCTS. By utilizing the Genetic Algorithm (GA) along with the existing gameplay platform, theoretically better generations could be generated with crossover and mutations and selected out through self-playing [7]. Also, combinations of MCTS along with neural networks or other techniques have shown considerable performance in the game, developing it into a pipeline may boost the win rate over the current algorithms.

References

[1] L. Galway, D. Charles, and M. M. Black, "Machine learning in digital games: a survey," *Artif. Intell. Rev.*, vol. 29, no. 2, pp. 123–161, 2008.
[2] N. J. van Eck and M. C. van Wezel, "Application of reinforcement learning to the game of othello," *Comput. Oper. Res.*, vol. 35, no. 6, pp. 1999–2017, 2008.
[3] J. Pearl, "The solution for the branching factor of the alpha-beta pruning algorithm and its optimality," *Commun. ACM*, vol. 25, no. 8, pp. 559–564, 1982.

[4] C. Browne, E. J. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. P. Liebana, S. Samothrakis, and S. Colton, “A survey of monte carlo tree search methods,” *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, 2012.

[5] A. Barber, W. Pretorius, U. Qureshi, and D. Kumar, “An analysis of othello ai strategies,” pp. 3–6, 2019. [Online]. Available: https://barberalec.github.io/pdf/An_Analysis_of_Othello_AI_Strategies.pdf

[6] V. Sannidhanam and M. Annamalai, “An analysis of heuristics in othello,” pp. 3–10. [Online]. Available: https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/Final_Paper.pdf

[7] O. E. David, H. J. van den Herik, M. Koppel, and N. S. Netanyahu, “Genetic algorithms for evolving computer chess programs,” *IEEE Trans. Evol. Comput.*, vol. 18, no. 5, pp. 779–789, 2014. [Online]. Available: https://doi.org/10.1109/TEVC.2013.2285111