Keaton Spiller

Tue Doan

Long Chung

Nahom Ketema

CS441 Final Report

Description:

The game that will be discussed in this paper is called Othello. Othello is a game played by two players as follows. There is an 8x8 board initially where there are 2 players, one black and one white. When the game starts, one of the two players puts a tile in. However, the player can only put a tile where there are one or more of the opposition's tiles(horizontally, vertically, or diagonally) and make a "sandwich". When that happens, all of the opponent's pieces get flipped and become the player's pieces. For instance, the black piece can place a tile on D3 or F5 in the image below. The game ends whenever there is no possible move for both players or the board is full. The winner is determined by the player that has the most tiles.

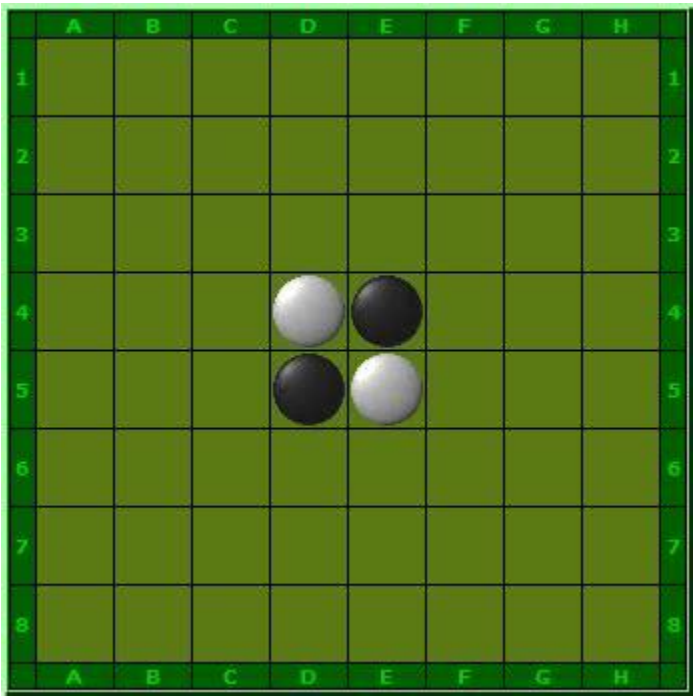


Figure 1: Initial configuration of Othello

Contribution:

For the final project, we all worked together on planning and building the framework for both the RL and MinMax algorithm. Then, we also worked together to test and compare both algorithms. Keaton Spiller did most of the coding for the MiniMax algorithm. While Tue Doan worked on the RL learning algorithm. Long Chung works on the heuristic functions for the algorithms and helps with the final report. And Nahom Ketema did most of the compiling and writing of this final report.

Setup:

In order to run the program, there are some libraries that have to be imported. The libraries used have been listed below:

- Random
- Numpy
- Pandas
- Matplotlib.pyplot
- Csv
- PySimpleGUI
- scipy.special

Program Description:

There are two different programs that were used for this project. The two programs are different implementations of an agent playing Othello. We will go in-depth on how these two programs function over the next few sections..

***Program 1: MinMax Algorithm**

This program applied a slightly modified version of the MinMax algorithm in order to play the agent, which we will go in-depth later on. The program has some simple helper functions that help carry out the basic tasks when playing. This includes initializing the board, finding the player's possible moves, and so on. The agent then has three different heuristics with which it can be tested. These are the following

1. Number of moves: This heuristic looks at the total number of moves a player has(the higher the better).
2. Number of tiles: This heuristic counts the number of tiles a player has(the higher the better).
3. Number of flipped tiles: This heuristic looks at the maximum number of tiles a player can flip during their turn(the higher the better).

Using one of these three heuristics, the agent attempts to do a reduced version of the MinMax algorithm. The reason the agent only does part of MinMax is because the state size is too large to do MinMax. Therefore, the agent can only simulate a small portion of the possible states.The MinMax consists of finding possible moves, and optimal tiles that will be flipped from the possible moves.

We play out games within games to update moves that have a higher score for player one when playing against an optimal opponent 'n' levels of depth deep into a game, and use heuristics to score how good one move is from another. In order to optimize the game states we used the 8 surrounding square tiles surrounding the current tile, and immediately threw out any surrounding tiles that didn't have possible moves, were outside the bounds, or did not have possible flanks from a particular move. This allowed us to avoid doing many unnecessary loops and not check

possible combinations of impossible states. This square consisted of a dial similar to a compass with (North, South, East, West and 4 diagonals).

***Program 2: RL learning Algorithm**

This program used RL with a Q-learning algorithm. The board game is represented by a 8x8 2D array. Values 0, 1, 2 on the array represent blanks, black tiles, and white tiles consecutively. The state of the game is a flatten array of this 2D array, and the action is the index of this flatten array. There are two Q networks as in Figure 2. Each network is a multilayer neural network with networks' parameter θ, one hidden layer and one output layer. These networks have 64 inputs as the game's state and their output are arrays of 64 possible actions. After state s takes action a, it will receive a reward r. If current action leads to next won action, the reward will be 0.9. Otherwise, the reward will be 0.1 for loss state and 0.5 for draw state.

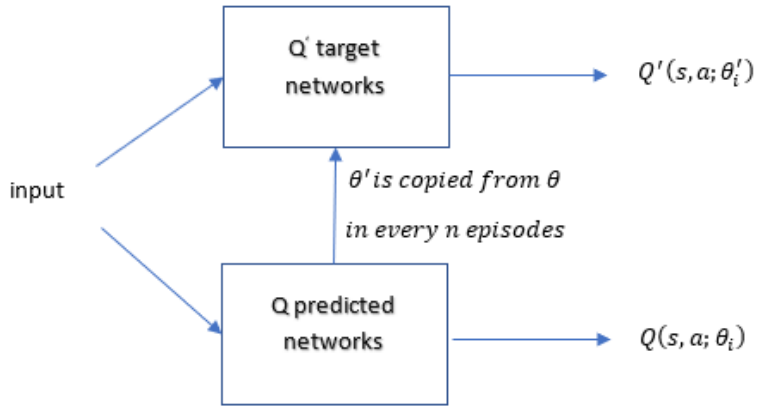


Figure 2: Q networks design

The network's parameter of the predicted network (or weight) is updated through a backpropagation algorithm.

Predicted value:

$$y^k = Q\left(s_t, a_t, \theta_t\right)$$

Target value:

$$t^k = Q'\left(s_t, a_t; \theta'_t\right) + \alpha\left[r_{t+1} + \gamma max Q'\left(s_{t+1}, a_t; \theta'_t\right) - Q'\left(s_t, a_t; \theta'_t\right)\right]$$

Error terms:

Output k: $\delta_k \leftarrow y^k(1 - y^k)(t^k - y^k)$

Hidden j: $\delta_j \leftarrow h_j(1 - h_j)\left(\sum_{k\ output} \theta_{kj}\delta_k\right)$

Weights update:

$\theta_{kj} \leftarrow \theta_{kj} + \Delta\theta_{kj}\ where\ \Delta\theta_{kj} = \eta\delta_k h_j$

$\theta_{ji} \leftarrow \theta_{ji} + \Delta\theta_{ji}\ where\ \Delta\theta_{ji} = \eta\delta_j x_i$

α, η: $learning\ rates$

γ: $discount\ term$

The weights of the Q networks are continuously updated by training, self-playing, and playing with humans.

Results:

Our project compared two different kinds of algorithms in order to see which one performed better against a randomized opponent. The first algorithm that was used is the MinMax algorithm. This algorithm was tested on three different heuristics(which were mentioned above). The second and final algorithm was an RL Learning algorithm. The program was trained in 10 iterations. Each iteration included training of 1,000 epochs with 1,000 episodes per epoch and self-playing with the previous version of the network's weights

Comparing Mini Max(Tree Depth = 20)and RL Learning Algorithm

Games played	Player One Score	Mini Max (Number of moves)	Mini Max (Number of tiles)	Mini Max (Number of flipped tiles)	RL (random player)	RL (self-playing)
5	Mean	36.6 4/5 wins	34.6 2/5 wins	32.8 2/5 wins	34.8 3/5 wins	42.68 4/5wins
5	Standard Deviation	3.14	7.71	8.66	10.70	8.73
10	Mean	32.7 5/10 Wins	39.5 8/10 wins	40.4 9/10 wins	34.19 6/10 wins	43.03 9/10 wins
10	Standard Deviation	9.12	8.00	10.45	10.42	9.26
20	Mean	37.75 14/20 wins	36.45 14/20 wins	37.6 15/20 wins	33.85 11/20 wins	42.37 17/20
20	Standard Deviation	10.57	8.48	6.91	10.39	8.98

Table 1: Mean and Standard Comparison between Mini Max and RL

The RL had the highest mean score over the first couple games, but games 10-20 the Number of flipped tiles was the best heuristic with 15/20 wins and an average of 37.6. It is interesting that the Number of moves had a high mean for 14/20 wins with a high standard deviation of 10 similar to the Number of tiles standard deviation.

10 games, Tree depth = 20

(Player One in Red, Player Two in green)

Figure 3. Mini Max (Number of flipped tiles)

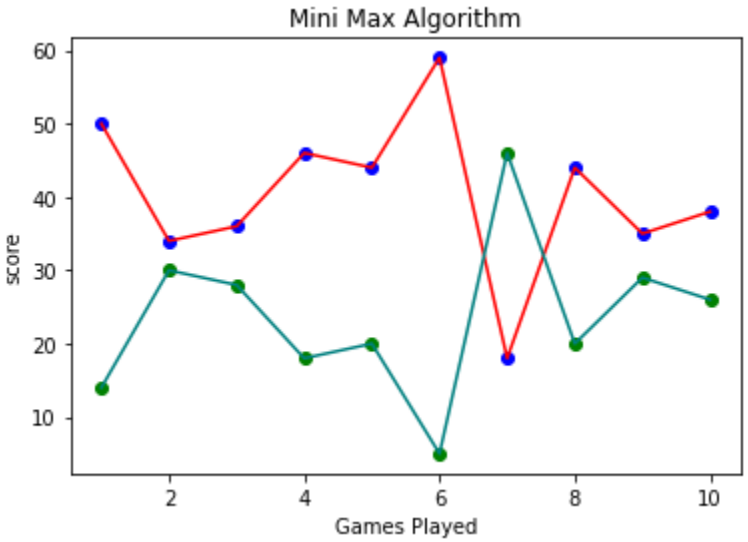
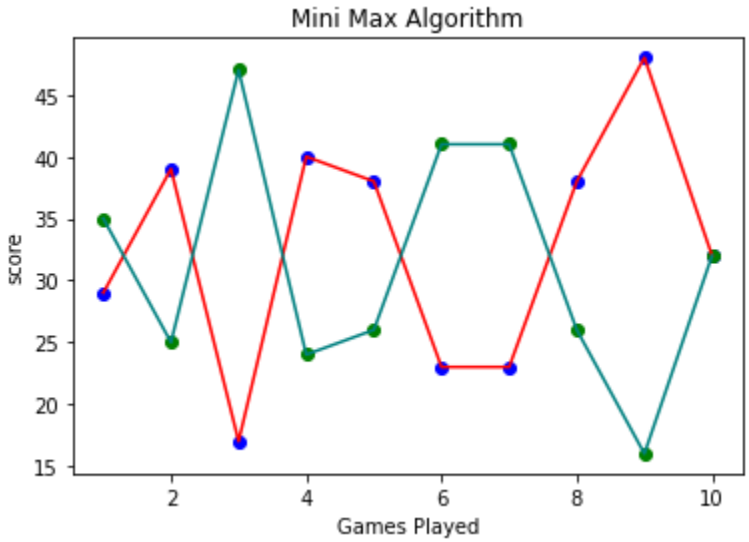


Figure 4. Mini Max (Number of Moves)



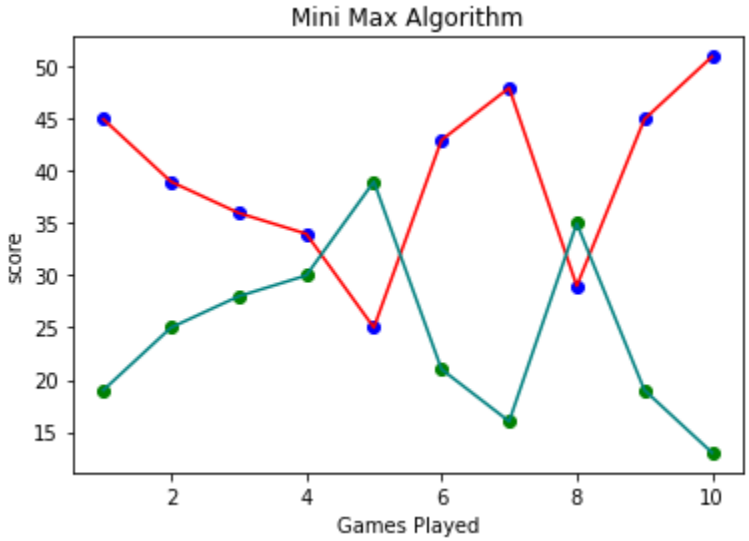


Figure 5. Mini Max (Number of tiles)

Number of tiles and Number of flipped tiles had the highest scores on average for the three heuristics, and Number of moves had the lowest average.

20 games, Tree depth = 20

(Player One in Red, Player Two in green)

Figure 6. Mini Max (Number of flipped tiles)

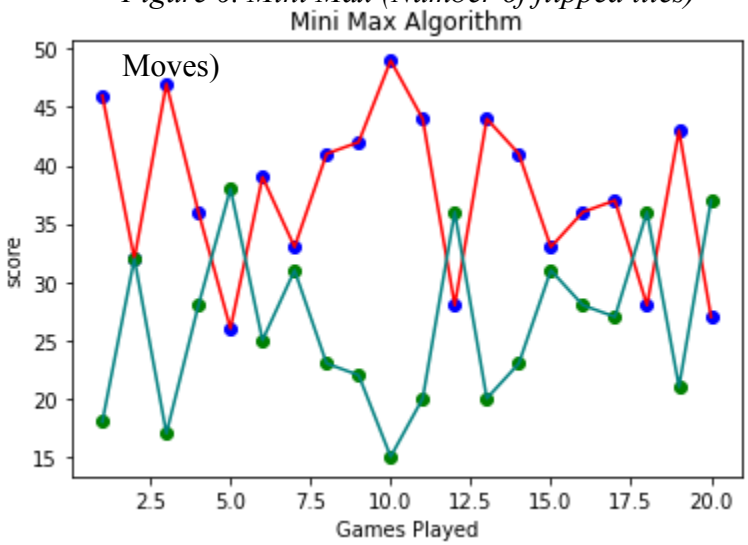
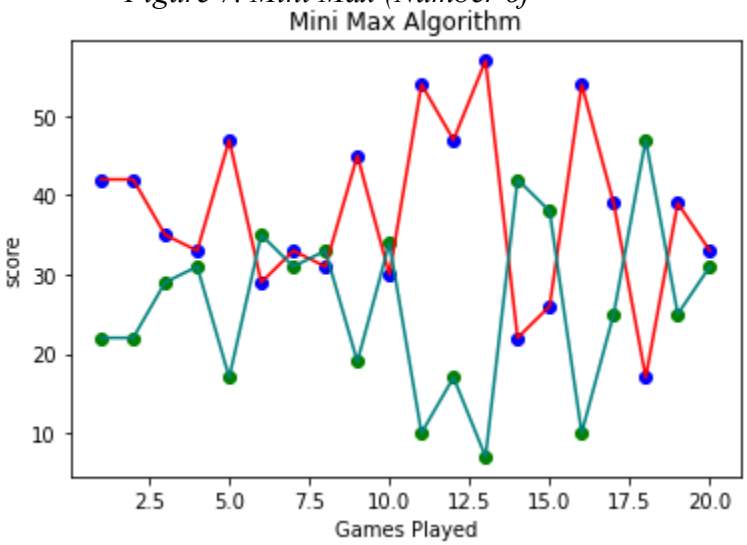


Figure 7. Mini Max (Number of Moves)



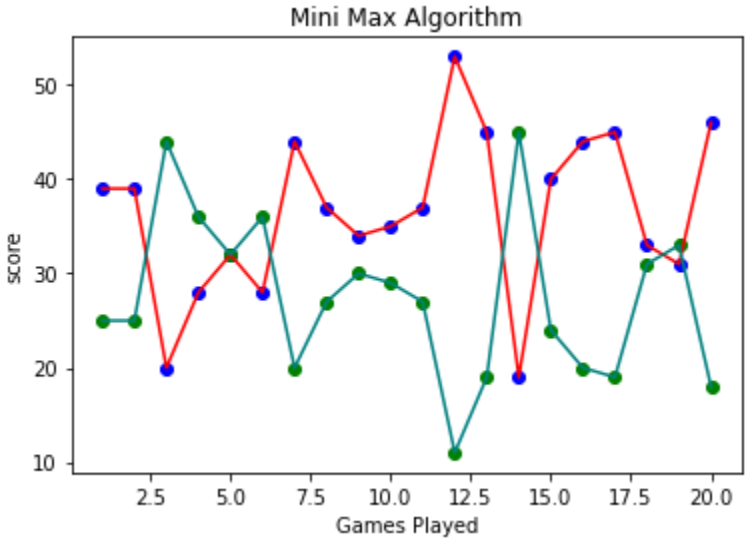
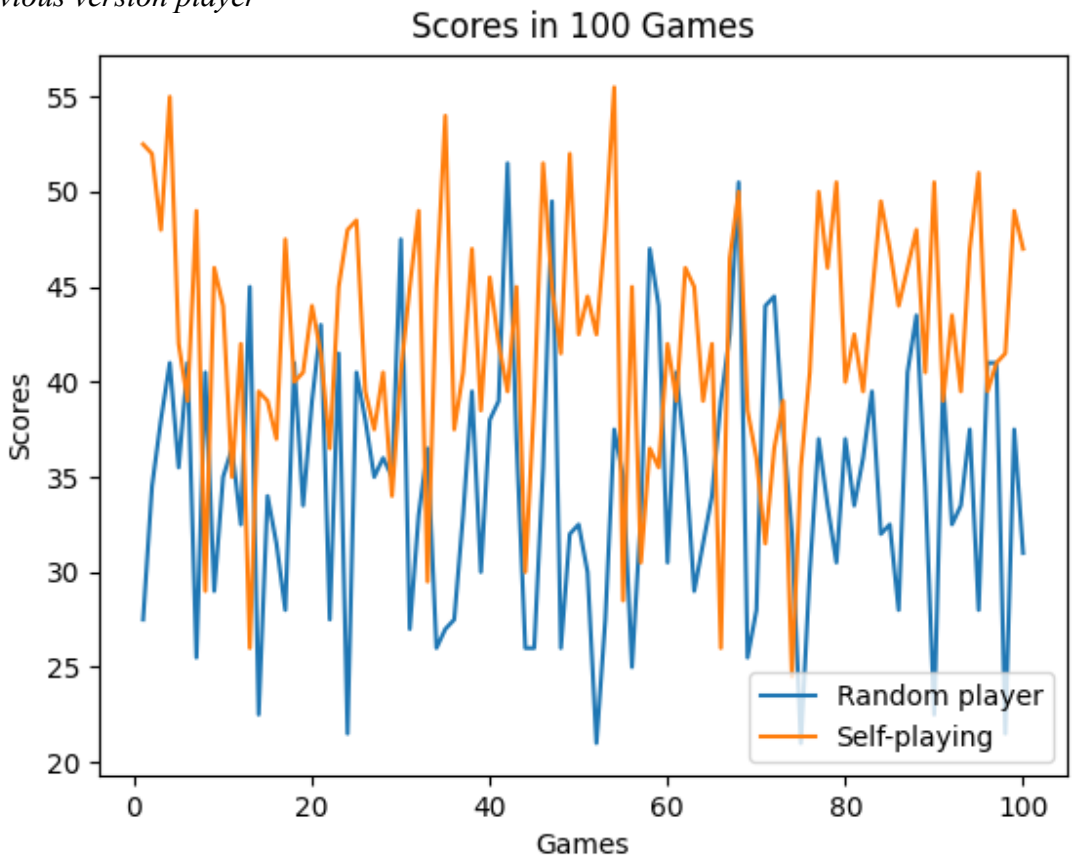


Figure 8. Mini Max (Number of tiles)

Number of tiles and Number of flipped tiles had the highest scores on average for the three heuristics, and the Number of moves heuristic had the lowest average.

Figure 9. RL - scores in 100 games when the program played against a random player and a previous version player



The RL algorithm shows better performance when it plays with its previous version player than a random player. This can be explained by the number of training epochs (10,000) being too small to explore all the states of the game, which is about 10^{28} states. However, it has a big improvement when playing with its previous version.

Conclusion:

Playing 20 games on a trained RL neural network took seconds to run, yet the minimax took over 10 minutes for each heuristic. Training the neural network initially took over an hour for 1000 epoch's. The best algorithm for 20 games on a random player was the MiniMax with 15 wins out of 20 games using the Number of flipped tiles heuristic. It is important to note that the RL learning algorithm could be trained more extensively, and produce significantly better results with larger epoch's, iterations, and hiddens nodes.

The Minimax algorithm could be improved by replacing base python methods with numpy packages. Each python While loop, and for loop incrementing in python is 10x slower than numpy code iterations written in C, which could significantly improve runtime. There could also be further optimization in the subloop game within the game for the minimax trees generated, such as alpha-beta pruning, or other optimization approaches to trim branches of trees that are unnecessary.

Reference:

H. v. Hasselt, A. Guez and D. Silver, "Deep reinforcement learning with double q-learning", Proc. 38th AAAI Conf. Artificial. Intell., pp. 2094-2100.

Russell, Stuart J., and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson, 2022.