CS 229 Final Project:
Using Reinforcement Learning to Play Othello

Kevin Fry ID: kfry
Frank Zheng ID: fzheng
Xianming Li ID: xmli

16 December 2016

Abstract—We built an AI that learned to play Othello. We used value iteration on a value approximation function in combination with minimax tree search with alpha- beta pruning. The performance of our four-feature linear value approximation function stands out above all other strategies in terms of win percentage.

1 Introduction

Othello is a two person game played on an 8 by 8 board with 64 pieces that are black on one side, and white on the other. Each player is assigned a color, black or white. Starting with four pieces in the middle of the board, two white and two black, players take turns placing pieces on the board. A legal move must be adjacent to other pieces on the board, and be placed such that it flips over at least one piece (Fig. 1). A piece is flipped over if it is between a piece just placed by a player and another piece of that player's color. The game ends when neither play has a valid move left, and the winner is the player with the most pieces of their color on the board. See the Wikipedia page for more information.

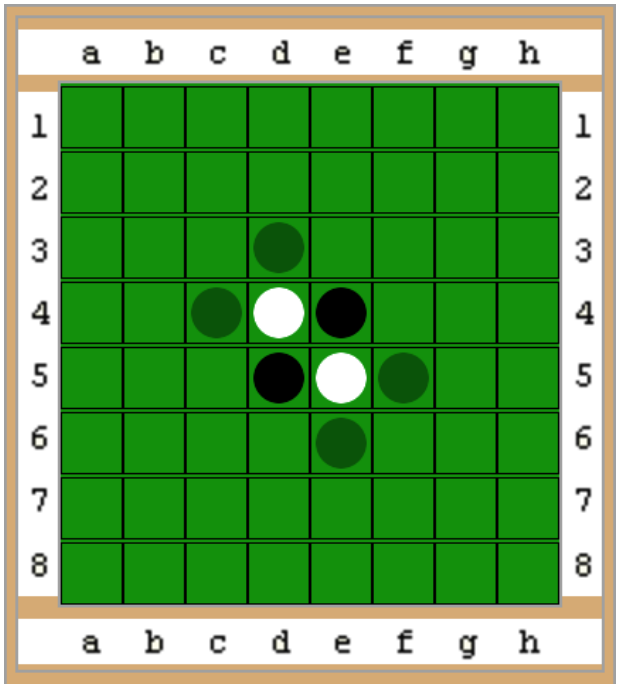

Figure 1: The valid moves for black

As in most reinforcement learning applications, our problem has a state space (the different configurations of the board), actions the player can take in a given state that cause a transition from one state to another (the valid moves for a player given a board configuration). Thus it is natural to use reinforcement learning algorithms to create our agents.

In our final agent, we use minimax with a value approximation function to play the game. Minimax is discussed in the next section. Our value function is a linear function of four board

heuristics.

2 Game playing and GUI

We built a GUI to display the game and a python program to allow us to play the game. The game is represented as a two dimensional 8×8 array. A 0 represents a blank square, a 1 represents a white piece, and -1 a black piece. The python program can take in player input for placing a piece, calculate all valid moves for a player, and properly update the board when a move is taken, including flipping over pieces. After each move, the array is interpreted and displayed in a GUI to make it easy to see the state of the game and how it progresses (Fig. 2).

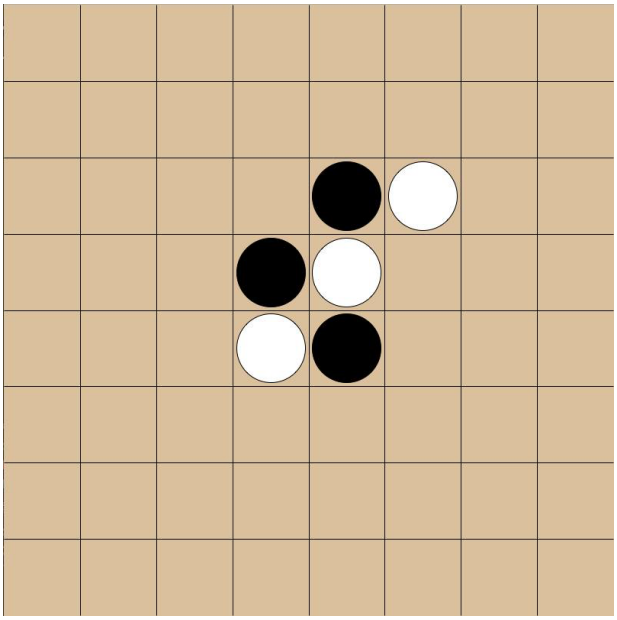

Figure 2: Example of our GUI

3 Minimax Heuristics

As a two-player deterministic zero-sum game with perfect information (both players know all previous moves at any given state of the game), Othello is suitable for the minimax algorithm, which provides a reliable way to converge upon the optimal solution, given enough allocated time and space.

The idea of minimax is built on top of the concept of decision game trees, which makes use of the concepts of state space and move space of the game board. The state, s_t, at any given time t is the configuration of pieces for both players on the board at that time, and the move space, $m(a, p)$, is the set of all possible next moves/actions, a, for any given player p. The game tree starts at the root s_0, which is the starting configuration of the game, and branches down to all possible state for each possible move choice for the current player, alternating moves between players. This procedure develops down the tree, until the entire state space for the board has been filled. At this point, the set of leaves at the bottom of the tree would constitute all possible winning/losing positions for the players. Each path from the root to a leaf would signify an entire game.

In order to make a move at each branch for the minimax, a criterion called the evaluating function is employed. The evaluating function takes the form, $f : S \to [m, M]$, where S is the set of all possible configurations of the board, m is the minimum possible score for a player, and M is the maximum possible score for a player. Without loss of generality, naming one player the maximizing (MAX) and the other player the minimizing (MIN), the goal of minimax would then be for the MAX to achieve M and for the MIN to achieve m.

3.a Minimax Considerations

Due to the nature of the game tree, if we assume the branching factor (number of possible moves per turn) is 6, and that there are about 55 turns for both players combined, propagating minimax entirely down to the leaves would require $6^{55} > 10^{42}$ static evaluations, which takes considerable time and space.

To tackle this problem, we create an insurance policy in which, given time constraints, at some level d down the tree that may not be the leaf-level, we assume that level to be an

approximation state of the leaf level. We then run the evaluation function on the nodes at this level and backtrack minimax up the tree. In addition, we plan to also take advantage of the $\alpha\beta$-pruning technique which optimizes minimax search space by avoiding completely traversing down certain branches where we know fairly quickly that the optimal solution would not be found there. In doing so, we may be able to look ahead several more levels in a given amount of time and generate more accurate evaluations.

4 Discussing Heuristics

All heuristics discussed below are done so from the perspective of playing as the black player.

4.a Frontier Disks

Frontier disks are the pieces that are adjacent to open spaces. These disks are often volatile because they are easily flipped back and forth between the two players. Therefore, it is intuitive that we try to minimize these pieces for ourselves and maximizing them for the other player so that we have a solid base of pieces. To calculate this function, denote the number of frontier disks of black B_f, and the number of frontier discs of white, W_f. If $B_f > W_f$, then the frontier discs score is

$$f = -100\frac{B_f}{B_f + W_f}$$

If $B_f < W_f$,

$$f = 100\frac{W_f}{B_f + W_f}$$

And 0 if $B_f = W_f$.

4.b Mobility

This measures the number of moves available at a time. Running out of moves is bad because it gives the other player an extra turn. Thus, we will try to maximize our own mobility and minimize the other players mobility. Denote B_m the number of available black moves, and W_m the number of available white moves. For $B_m > W_m$ the mobility is

$$m = 100\frac{B_m}{B_m + W_m}$$

If $B_m < W_m$

$$m = -100\frac{W_m}{B_m + W_m}$$

And 0 if $B_m = W_m$.

4.c Piece Difference

This measures the difference between our own pieces and those of the other players. We try to maximize this number as one of our heuristics, because ultimately, the objective of the game is to have more pieces than the opponent at the end of the match. Let B_p be the number of black pieces, and W_p the number of white pieces. If $B_p > W_p$, then p is given by

$$p = 100\frac{B_p}{B_p + W_p}$$

If $B_p < W_p$

$$p = -100\frac{W_p}{B_p + W_p}$$

And 0 if $B_p = W_p$.

4.d Value Matrix

The value matrix is a matrix that we generated with reinforcement learning to determine which spaces in the board are more or less valuable to have. Intuitively, we thought of this heuristic because we knew that corner pieces are extremely valuable in Othello because once occupied, they can never be flipped to another color. In this sense, we also can conclude that spaces adjacent to corner pieces are bad, because they allow a way for the other player to occupy corner pieces. With this in mind, we designed an agent that learned a value matrix which gave values to each space on the board. Since the board quadrants are symmetric, below we only give the value matrix V for the upper left quadrant:

$$V = \begin{bmatrix} 3.2125 & 1.775 & 1.875 & 1.975 \\ 0.15 & 2.3 & 0.6625 & 1.8375 \\ 3.525 & 0.85 & 2.675 & 0.175 \\ 1.125 & 1.95 & 0.15 & 0 \end{bmatrix}$$

A score is calculated from this matrix by adding the matrix value for each space of the board occupied by black, and subtracting the matrix value if it is occupied by white. Formally, the matrix score s is

$$s = \sum_{i=1}^{8} \sum_{j=1}^{8} f(i,j)V(i,j)$$

Where f is defined as

$$f(i,j) = \begin{cases} 1 \text{ if black} \\ -1 \text{ if white} \\ 0 \text{ otherwise} \end{cases}$$

The value matrix was computed through the method listed below:

1. **repeat for 10000 iterations:**
2. Initialize every value of board to 0.
3. Based on the decreasing exploration rate ϵ, either make the optimal move using the current value matrix, or make a random move
4. For each move, record it in a moves array.
5. At the end of the game, add a 1 to the value grid matrix for each move in the moves array if black won. Subtract a 1 to the value grid matrix for each move in the moves array if black lost.

4.e Training Our Weights

Algorithm:

1. **repeat until convergence:**
2. Randomly initialize weights θ.
3. Every ten iterations update training opponent $\theta_{opp} = \theta$ (First ten iterations, opponent is value matrix agent)
4. Based on exploration rate ϵ, either make the optimal move using current value function, or make a random move
5. For each move, record the board value v using current weights and feature vector f, and the corresponding minimax score v'
6. $\theta := \alpha(v' - v)f$

5 Benchmark Agents

We implemented several other agents as benchmarks to test our final agent against. First we built an agent that makes a random choice as a baseline to test all other agents against. We also implemented two piece difference agents: one that greedily tried to maximize the number of pieces of the current turn, and another that used a minimax of depth 3 to determine which move to make. Finally, we implemented an agent that used a value matrix that we found online, developed by Korman, a researcher who worked on Othello(1). His matrix was determined experimentally, and we used this matrix in two ways: one as a greedy value matrix that only accounts for the current state of the board, and another which uses a minimax of depth 3.

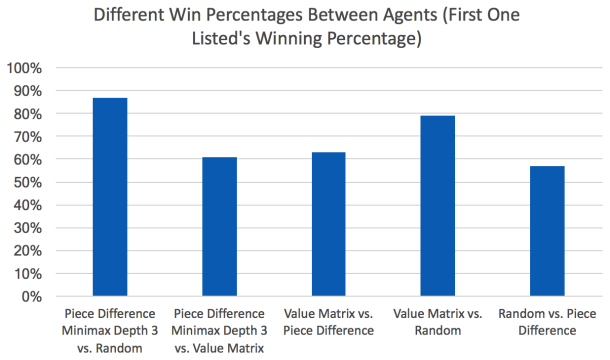

Figure 3: Win Rates vs. Value Matrix With Different Training Opponents

Fig. 3 shows how these basic agents stacked up against each other. As shown in the figure, the value matrix and piece difference with minimax agents were the best of our benchmark agents.

6 Results

Below are the results of our final agent. Figure 4 shows the results of different training opponents and Figure 5 shows the win rates against our benchmark agents. Finally, Figure 6 shows the learning rate of our final agent until convergence.

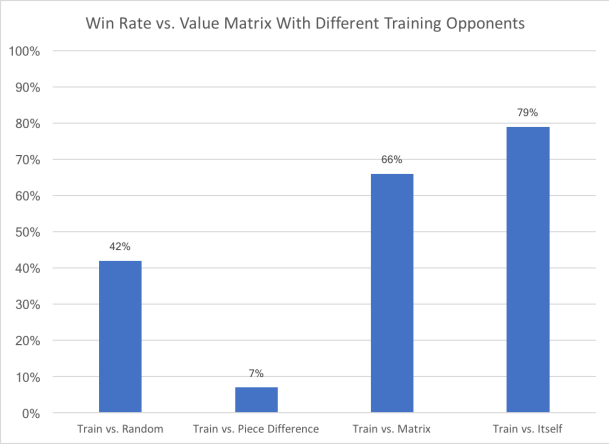

Figure 4: Win Rates vs. Value Matrix With Different Training Opponents

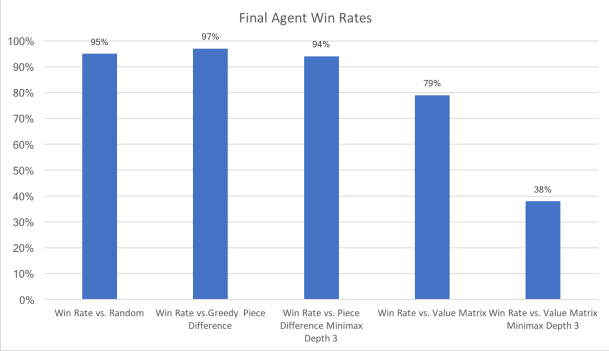

Figure 5: Final Agent Win Rates

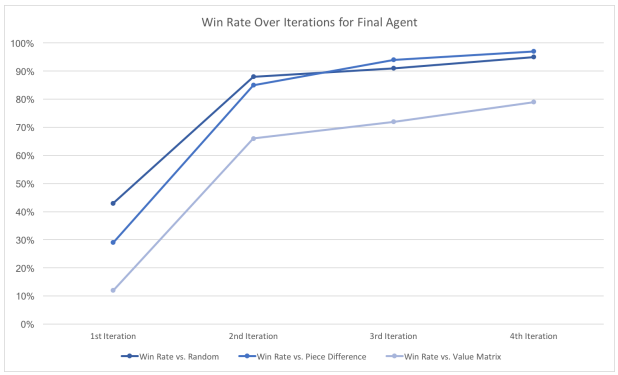

Figure 6: Final Agent Learning Rate Over Iterations

7 Discussion

Our final agent performed well against the agents that we tested it against. It performed extremely well against the random agent and the two piece difference agents (greedy and minimax). It also performed well against the greedy Kormans value matrix, although it was unable to beat the minimax Korman matrix.

It dominated against the random agent, because as in most board games, playing randomly is not a great strategy. However, it is noted that the random agent wins about 5% of the time, because playing randomly is actually not a terrible strategy (random agent actually wins against piece difference agent 57% of the time).

Our agent also won handidly against both piece difference agents, because the piece difference agents fail to account for the fact that many of its current pieces can be flipped in the future.

Finally, it worked well against Kormans value matrix when the agent was only using a greedy current-state look, but it failed to win most of its games when the Korman matrix used minimax of depth 3. We think that this can be attributed to the fact that Korman's value matrix was determined experimentally. Because of this, he was able to put his own biases and knowledge about the game into his value matrix. On the other hand, our matrix,

generated as described in 4d, simply added or subtracted 1 from a position based on if the winning player played that position. Since this method values every move equally, it struggles differentiate between genuinely valuable moves and those that just happened to be played by the winning player. Also, though we failed to beat the Korman matrix with minimax of depth 3 handidly, we only lost around 60% of the time, which is not terrible by any means.

8 Future Extensions

8.a Neural Network for Value Approximation

Currently, we use a linear value approximation function. The agent might work even better with a neural network to approximate the value function. We had planned to implement this ourselves, but because of time constraints (exams) we were unable to make good on these plans. The following details how one would implement this in future work.

We would use an artificial neural network, with one hidden layer and each node being a perceptron or sigmoid function. The input would be the board, which would feed into 64 input nodes, these input nodes would be fully connected to all hidden layer nodes (two-thirds of the input number should be appropriate), and these hidden nodes would all be connected to an output node.

At the start of training, we would randomly initialize weights and thresholds for the neural network. For training, we would feed it many examples (thousands of examples even) of board positions obtained from simulated games using the linear value approximation agent. For each input board position, the signals would be propagated through the neural network to produce an output. Then by providing the neural network with the scoring assigned by the linear approximation agent, back propagation can be used to tune the weights and thresholds of the network.

Once a network had learned to play at the same level as the linear approximation agent, we could then train neural networks against each other. Both neural nets would be initialized with the weight and threshold values found for the first neural network. In this case, the networks would play games until the end, with scores given based on how quickly they won/lost; highest scores for winning quickly, and lowest scores for losing quickly. Then these scores could be used for back propagation to update both agents. Hopefully this would result in agents that learned to play even better than the original neural network.

Another approach, possibly less appealing because of its potentially long training time, is a genetic algorithm to select the weights and thresholds for a neural network. The process would involve 100 neural networks, most initialized with random values, and some initialized with values from the first neural network. These agents would then play against each other in a round-robin tournament. At the end of each generation, we would keep the top performers, discarding the rest. We would then create copies of these top performers, and mutate these copies to create 100 neural networks for the next generation. Then the process would repeat. The process would terminate when mutations were no longer providing improvement. This would be determined by periodically playing the best performer of the latest generation against some benchmark agent (e.g. random or greedy piece difference), to ensure real improvement is actually occurring.

8.b Feature Selection

Another way our project can be improved is through our selection of features/heuristics. We experimentally chose the four heuristics used here, but future work should come up with more heuristics and use forward or backward search feature selection to determine the optimal number and combination of heuristics to use as features in the linear value approximation function.

8.c Deep Learning

Finally, another way to improve on our project is using Deep Learning instead of our current model. Deep Learning would mean that it would essentially come up with its own features instead of the heuristics that we came up with ourselves for this project. This way, the model would be free of our own biases as to what moves and what heuristics are good or bad.

References

[1] M. J. Korman, "Playing othello with artificial intelligence," 2003.

[2] J. van Eck and M. van Wezel, "Application of reinforcement learning to the game of othello," 2008.

[3] V. Sannidhanam and M. Annamalai, "An analysis of heuristics in othello," 2015.