REVERSI LAB REPORT

Chase Hiatt

1. INTRODUCTION

For this lab we were tasked with creating an agent capable of playing the popular game $Reversi$. In this game, players take turns placing down pieces until neither player has any valid moves. At that point, the player with the most pieces currently on the board wins. When either player takes their turn and places down a piece, any 8-connected adjacent enemy pieces which are sandwiched between pieces of the current player will be converted into the pieces of the current player. This "Capturing" mechanic can make heuristic evaluations very difficult, because it means that 'value' does not directly correlate with current score. Instead, a player must find a way to identify how good a board position is based on the relative position of the pieces. Furthermore, the value of all actions is determined not only by the current state, but by all potential future states which may be reached from that action. Due to this future facing value function, we need an efficient way to explore the play-space of potential actions.

2. MINIMAX AND ALPHA-BETA PRUNING

Minimax is an algorithm which finds the optimal play for a given situation. It presumes there are two players competing in a zero-sum game. One of these agents is the maximizer, while the other is the minimizer. The maximizer is seeking to maximize the value of some objective funciton, while the minimizer is trying to minimize that value. The Minimax algorithm was developed for just this kinds of games. While Minimax is guaranteed to find an optimal choice, it has one major draw back. In order to guarantee optimal choices, all future states need to be fully explored. According to Victor Allis' *Searching for Solutions in Games and Artificial Intelligence*, the size of a complete game-tree of *Reversi* is on the order of 10^{50} or roughly one Sexdecillion. This number is far too large for human comprehension and even with fast desktop hardware, this number isn't realistic for a typical computer. This forces us to find tricks to optimize exploration.

One way the searching process can be sped up is through the use of Alpha-beta pruning. In this process, we track alpha and beta values roughly representing upper and lower bounds as we do a DFS through the game space. If we assume the other player to be rational, then we can ignore large parts of the tree which an optimal player will not pick. This technique effectively halves the amount of computation needing to be done at each turn. According to Allis' paper, the branching factor of *Reversi* is 10. This means to search to a certain depth D, the expected number of nodes to explore is given by $O(5^D)$ instead of $O(10^D)$. After implementing Alpha-beta pruning, I ran my program against itself and tracked the number of states explored at each turn. Figure 1 is a graph showing the number of states searched with each technique. It is worth noting that disabling Alpha-beta pruning did not affect the moves that were made at all

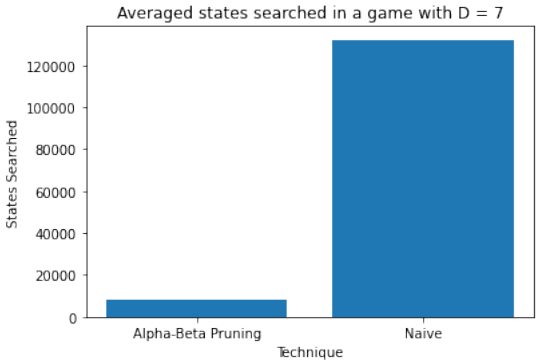

Fig. 1. The Effectiveness of Alpha-beta Pruning

While this speedup may not seem immense, imagine we are trying to search to a depth of 10. Rather than searching $\approx 10^{10} = 10,000,000,000$ states, we only have to search $\approx 5^{10} = 9,765,625$ states, a 99.9% reduction. While this optimization does a great job at reducing our search space, the number of turns in a *Reversi* game is still far too large to compute out. A typical game of *Reversi* lasts around 58 turns. Using our Alpha-beta pruning, this still leaves us with with around $3.5 * 10^{40}$ states to explore. This extremely large search space means we are not able to fully explore the game-tree of *Reversi*. Therefore, we are forced to prematurely stop our search and try to estimate the value of the current state using Heuristics.

3. HEURISTIC

My initial heuristic function was extremely rudimentary. The value of a given state would be given simply by the current

net score of player 1.

For example, if Black has 25 pieces on the board, and White had 15, the value of that state would be +10. If the piece counts were reversed, (Black has 15, white has 25) then the value of that state would be -10. I then made Player 1 (Black) the maximizer and Player 2 (White) the minimizer and ran a heuristic-fueled version of mini-max. As I played this version of the program, I quickly found it to be trivial to defeat by simply playing for corners and edges. This program did not understand that some positions in the game are inherently more valuable than others. As such, it would happily give up corners as long as in the immediate term it gained a piece or two over its opponent. Clearly this version of heuristic evaluation needed a sizeable upgrade before it was going to be effective.

After my program's initial poor performance, I decided to learn more about *Reversi* to come up with a better heuristic. After briefly researching, it seems the consensus for *Reversi* is that the corner and edge spaces are significantly more important than interior spaces. This is the case because edge pieces are both: *a)* Harder to capture, and *b)* easier to use when capturing opponent's pieces. However, the spaces directly adjacent to the corners are undesirable because they can easily lead to your opponent claiming the corner location. After significant tinkering and experimentation, I eventually settled on a weight map that incorporates this domain knowledge when evaluating a state's value. Figure 2 shows a heat-map representing importance, with warmer colors meaning a higher value.

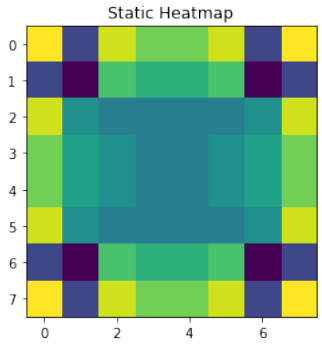

Fig. 2. Initial Heatmap for Evaluating State

Now, rather than use the Naive net-score evaluation, I calculate a net-value metric by adding up the values of the spaces owned by each player and subtracting them. I am still using the net values, but now my heuristic scoring is much more closely correlated with victory.

This approach has some major flaws, and I quickly identified that a temporally constant value map doesn't always do a good job of representing value. For example, imagine that you have already captured a corner space. According to Fig. 2, the adjacent squares still hold a negative value and should therefore be avoided. However, in reality, by holding a corner, you cause the adjacent squares to be completely immune to capture. Conversely, when an opponent holds the corner, the adjacent squares become almost impossible to hold on to, making them even worse. This led me to the idea of an adaptive weight map which can look at which spaces are held, then determine which spots are valuable. After another round of experimentation and gameplay, I made the places around the corner have a value contingent on who owns the corner. Fig. 3 is one potential generated importance map that corresponds to the current player holding the top corners, with the opposing player holding the bottom corners.

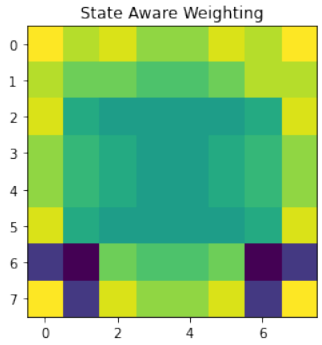

Fig. 3. Adaptive Heatmap Example

After playing against this program, it subjectively seemed to improve in performance compared to its non-adaptive counterpart. Unfortunately, I have no way of objectively measuring my program against other similar programs, so I can't say for sure whether this improved the agent's fitness.

4. EVALUATION

The first (and most scientific) way I evaluated my algorithm was through comparison to a random agent. I used the provided *RandomGuy* class provided to us and had them face off a few times. Overall, my agent won 8/10 games against the random agent. I did these runs at a max depth of only 7 so they would run quickly. With this depth, my agent would never take more than a second or two to respond. In an attempt to improve my algorithm's performance, I increased my max depth to 8, but that tanked my performance and sometimes left my agent thinking for minutes at a time. With a depth of 8, my algorithm would occasionally run into places where it would have to do heuristic evaluations on $64,000,000$ states before spitting out an answer. I tried removing my adaptive heatmap to speed up the heuristic evaluation function, but it didn't seem to make enough of a difference to get me to 8 layers deep. During these trials, my algorithm scored 6/10 games against the random agent. I'm not sure if the difference

is coming from losing my 'improvements' or if it's just from random variation between games.

After running my agent against random, I tried playing against 8 times. I found that I ended up winning 5/8 matches. Against the random agent, I was able to win 7/8 matches, so I do believe my agent to be better than random. I also had my significant other Beth play against the computer a few times and she lost to it 3 times in a row. She had never played *Reversi* before, so take this result with a grain of salt. These games were played with a max depth of 7 and I never noticed any significant delay between my move and the agent's response ($<$2 seconds).

5. CONCLUSION AND FUTURE WORKS

This project was very educational. I was glad to have an opportunity to experiment first hand with Minimax, Alpha-beta pruning, and heuristic evaluations. While my algorithm was able to beat the random agent, it wasn't particularly good against even a novice player. I think the weakest part of my algorithm is my heuristic. If I were going to try to perfect my algorithm, I would spend more time researching the intricacies of *Reversi* to better understand how to adaptively value different spots. I am confident there exist some heuristics which will vastly out perform mine, and I think that would be a fantastic place to improve my code. Another think I would do is create a script to do automate duels between two agents. For the purposes of this lab, I was just manually running the server and agents for each trial, but this was extremely tedious. If there was a script to start everything up with the correct arguments, I think my algorithms performance against random would measured much more accurately. This would also let me say with confidence whether or not my changes to the weight map made things better.