**Amin Babar**

**CS375**

**03/06/20**

**Project 2**

For this project I implemented several Othello players including minimax algorithm, Alpha beta pruning with a simple heuristic, Iterative Deepening Search player with minimax and alpha beta pruning, a player that relies on time remaining and uses IDS, a player using IDS, alpha beta pruning, and a refined heuristic, a player that accounts, in its heuristic, for c-squares in Othello.

**Minimax Algorithm:**

I started by first implementing the minimax algorithm. This minimax algorithm was implemented with a simple heuristic where the count of white tiles was subtracted from the count of black tiles. Black was the Max player and white was Min player. To test the efficacy of my algorithm, I ran it against a random player at different depth levels. Any depth level over 5 took too long and was not feasibly. Running the algorithm at different depth levels led to following results:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sr No** | **Game Type** | **Depth** | **Black Wins** | **White Wins** | **Draws** | **Mean Time** |
| 1 | Black (Minimax) vs  White(RandomPlayer) | 1 | 28 | 19 | 3 | 0.298s |
| 2 | Black(RandomPlayer) vs  White(Minimax) | 1 | 17 | 31 | 2 | 0.197s |
| 3 | Black (Minimax) vs  White(RandomPlayer) | 4 | 34 | 16 | 0 | 48.5s |
| 5 | Black (Minimax) vs White (minimizer) | 4 | 0 | 1 | 0 | - |
| 6 | White (minimax) vs Black (minimizer) | 4 | 0 | 1 | 0 | - |
| 7 | Black (Minimax) vs White (Maximizer) | 4 | 0 | 1 | 0 | - |
| 8 | White (minimax) vs Black (Maximizer) | 4 | 0 | 1 | 0 | - |

The algorithm seems to perform better at a deeper depth which makes sense because the further it can look down in the tree the better moves it can make as it has more insight. Since we were limited by 150s per player, depth of 4 seemed to do the best job while not running out of time. A better depth would give a better insight, but the player would lose because the time would run out. The player took shorter to reach a depth of 4 during the first few moves and the last few moves, but during mid game, the player took much longer because of the branching factor. For this project, I took most of my stats for different algorithms by using that algorithm on black. Since black is at a disadvantage because it does not get to flip last in most cases. I did not have the time to run 50 games against random for each color because 50 games take around an hour to complete.

For games vs short term minimizer and short-term maximizer, I only ran one iteration because I should expect the same result for multiple iterations. It was surprising that minimax managed to lose while it was black. I was expecting at depth 4, minimax, which is a smarter algorithm to do better against “not smart” minimizer and maximizer algorithms. This could be a result of my heuristic which I improve on for an algorithm that I discuss later.

My first implementation for minimax for 50 games had a mean of 69.8s which was higher than that of professor Helmuth’s in class algorithm. Therefore, I spent some time trying to make my algorithm more efficient. I was using \_\_repr\_\_ to access the move pair. Also, I was using lists to keep track of the max or min values at each recursive depth. After removing these 2 inefficiencies and a few more redundant calculations, I was able to bring my time down. Since this algorithm was going to be the basis for most of the rest of my algorithms, I was concerned with making it efficient.

**Alpha Beta Pruning Algorithm:**

Alpha Beta Pruning should essentially give us the same results as the minimax algorithm because it improves on minimax and cuts off search at redundant branches. Alpha beta pruning drastically improved the time for me. My average for 50 games against a random player went down from 48.5s to 16.8 s. I used the same heuristic I used for minimax algorithm emphasizing on the getting more pieces strategy. I found the implementation of alpha and beta in this algorithm really interesting. Keeping track of maximum value in alpha and minimum value in beta at each stage, and then cutting off the search if alpha >= beta seemed like a really simple but effective trick. For the rest of the algorithms that I implemented, I used Alpha Beta Pruning because it let me search deeper as the cost of time was less. One of the goals throughout the project was to be able to search deeper within the given time constraints so that we could “see ahead” and make a better move. Alpha beta pruning helped a lot with that.

**Iterative Deepening Search:**

I implemented IDS with both minimax and alpha beta pruning algorithms. Alpha beta pruning obviously performed much better. For this algorithm, I basically searched through the tree for a given amount of time and saw how much deep I could go for each iteration of depth in specified amount of time for each move. After that time, I would report the stats from the last depth completely searched. 4.9s seemed to do a good job for IDS. More time per move would usually result in timeout. IDS usually searches from depth 0 to infinity, but since the maximum possible depth in an Othello game cannot be greater than 64, the total number of spaces on the board, I chose 64 as my depth. Keeping track of depth at each level I saw that IDS helped me in the beginning of the game as it searched deeper (around 7 nodes deep). It also played a crucial part in the end game as it was able to search the complete remaining game tree and make a much more informed decision. I ran IDS on 50 games as black with 4.9 s per move against a random player, and it won 32 games while it lost 18 games. The mean time for the games was 129.4 s. These stats were similar to minimax algorithm, but refinements to IDS in the next player that I implemented gave me a greater control over the game.

I implemented the algorithm for IDS in a way I thought was weird. I implemented it using try and catch, and I was not sure if that was weird in a good way or a bad way.

if time.time() - start\_time >= time\_to\_spend:

return "timeout"

if is\_Max:

value = -1000

for move in available:

try:

value = max(value, self.AlphaBeta\_algorithm(new\_state, currDepth + 1, depth, move, False, alpha, beta, start\_time, time\_to\_spend))

alpha = max(value, alpha)

if alpha >= beta:

return value

except:

return "timeout"

return value

I kept track of the epoch time from the start of the turn in each of the recursive calls. As soon as I would reach the time limit, I would return a string “timeout” in the recursive calls. This would throw an error in the max and min functions for Max player and Min player. I would catch this error by returning string “timeout” again. This helped me keep track of the last best depth:

if moves\_eval\_list[-1] == "timeout":

break

else:

previous\_depth\_list = moves\_eval\_list

moves\_eval\_list = []

If any time outs were found, the IDS would stop and return the results at the previous depth.

**Player relying on remaining time:**

I thought keeping track of remaining time while using IDS was a really helpful tool. It helped me vary the time for IDS based on the remaining time. This solved the problem of time out towards the end of the game. I tried giving my algorithm 4 seconds in the start because I did not feel like that beginning had much of an effect on the game. Towards the middle of the game which I thought would be over the 70s region, I gave my player more time so that it can search deeper and handle the huge branching factor. As the time remaining lessened, I reduced the time available for the player more. End game was usually fast, so the remaining time did not have a huge effect on it. I optimized the reliance on remaining time for IDS further in a player discussed later on in the paper.

**Player relying on better heuristic:**

At this point in the project, I felt like I was being held back by my heuristic rather than the algorithm I was implementing, so I dove into the strategies for Othello game, and gained a much better understanding of the game itself.

I realized that the strategy to maximize one’s own pieces was flawed because it gave the other player more options to play in the later turn usually. With the maximizing strategy, the player would start the game really well, but towards the end, the player would usually lose even though it would have had higher pieces for most part of the game. There was a need to reduce the other player’s mobility in the game. I did that by changing the heuristic function and giving a value of 1 to available moves. This ensured that the player maximized its own number of moves, while it would try minimizing the opponent’s moves and making less options available for them. Othello strategies for mid game and start game usually revolve around this concept. Maximizing my own moves also let me have the game be in the center of the board because moves on the edges are limiting in the start of the game. Othello strategy websites recommended that I keep the game centralized, so this worked out well for that.

Th player was performing better, but it since I had the pieces still ranked at 1 each, it was still focusing on maximizing its own pieces, so I played around with different values and found 0.01 as the utility that worked best. I even considered having a negative utility for more pieces based on strategies, but that did not do so well towards the end of the game.

Another important strategy in Othello is getting the corners so I assigned a high value to the corners (15) hoping that I would be able to get the corners often that way. I did see my AI try going for the corners, but there were still a lot of cases where the random player would end up in the corner. My corners were worth 1500 discs and each move option was worth a 100 discs so playing around with these values did not do much good. Upon running a bunch of games, I realized my player was going for squares knows as the “x squares”. These squared are the squares directly diagonal to the corner. Any move over here would often lead to the capturing the corner. I assigned negative value of 5 to these squares, and that ensured that my AI only went for these squares when it had to capture the corner. This did lead to another problem. In the end game, my player was still avoiding the corners, and that led to the other player controlling these squares giving them a “false corner”. Keeping this in mind, I decided to only use this strategy during start game and end game. Playing around with a bunch of games, I tuned my algorithm by deciding to only use this strategy when I had more than 15 moves.

Running my new heuristic with alpha beta pruning and remaining time-based IDS, my player as black won 48 out of 50 games. This was a great improvement compared to the previous heuristics, but I decided to examine the 2 lost games in attempts to go for a perfect 50. In the 2 games I had lost, I noticed that my player was losing the corner because it would make a move on “C squares”. These were the squares that were adjacent to the corner squares. I decided to handle this in another player class called the **c\_squares\_heuristic**. I used the same strategy I did with the X squares assigning each of these a value of -5. My implementation negatively affected my AI player, as it seemed to slow it down, and it was not going as deep anymore. I decided to examine this in more detail later if I had the time.

Black was still always at a disadvantage to white, so I researched more into that. There’s a concept of parity in Othello where black is at a disadvantage because white goes last, and the last move is counted as a “stable piece”. The parity can be reversed if at some point black can make white skip a move so that it has odd number of moves remaining. I implemented this by giving an advantage of 5 points to whichever player had the advantageous parity.

One weird thing that I noticed was that running my better\_heuristic (white) player versus alphabetapruning player (black), alphabetapruning player won 1 game out of 10. All the other games were the exact same game except for the one that black one. I was expecting the same result throughout. I concluded that it might be because of the processor speed varying because I was running multiple algorithms at once which might have caused my time remaining based IDS algorithm to perform differently.

**Ordering moves player:**

The next logical step was to increase the speed of IDS, but my IDS algorithm kept running into problems when I tried reordering the moves. I ran out of time before I could debug it, but the intention was to order the moves, so that AlphaBetaPruning would happen faster since my heuristic function was taking a toll on the speed of search and was being limited to a lesser depth.

**Superior Player:**

Small optimizations were implemented for this player. The first move does not really affect the game, so it picks the first move as the first available move. This saves time on the first move.

Also, the branching factor in low on the first few turns so it provides the first few moves less time, trying to save time for end game.

Tried optimizing end game. The intention was to run the end game for the entire remaining tree as the branching factor is less and I have saved time for the end game. I run moves in the end game for a longer time, unless I am running out of time. This ensures that I have enough time to go through the entire search tree during end game.

For end game since we can search the entire tree, I call on a different heuristic which ranks based on the moves that will maximize the tokens for the current player.

This player needed more optimization, so it was not chosen as my tournament player.

**Conclusion:**

My algorithms were definitely smarter compared to HW 1. They performed better against random and in most cases non-random players. Further improvements can be made by debugging the ordering of IDS to make the algorithm search deeper. The heuristic could be further made better by including more strategies. More time could be spent on tried and tested beginning moves which gives the player an advantage.