Shane Ferrante – Group and Individual writeup.

CS 4150

Alpha-Beta Pruning vs. Monte-Carlo Tree Search in Othello

The purpose of this project is to implement the game Othello and then create two bots to play against one another, one using Alpha-Beta pruning, and the other using Monte-Carlo Tree Search. Then, the goal is to compare the two to see which approach is more suitable for this game.

The Game

The game of Othello (similar to Reversi) is a two-player board game on a (usually) 8x8 board where players alternate placing black stones and white stones in order to get as many stones of their color onto the board. When a stone is placed, all opponent stones in any adjacent or diagonal direction that are "surrounded" by stones of the player's color are "captured" and flip to become stones for the player who made the move. This game has a very complex and unintuitive strategy that is hard to master. I implemented the rules of this game manually in the Othello.py file. The only key thing to note here is that black pieces are represented by a "1" and white pieces are represented by a "-1"

The Players

The first step of creating an agent for Othello, whether for Alpha-Beta pruning or MCTS, is to create a heuristic function to determine the evaluation of the position. The obvious choice for a function like this is to just count the number of stones for both sides. However, anyone who has played a reasonable amount of Othello will tell you that having more stones is not necessarily a good thing and it can even be better to have fewer stones in many cases. Because of this, a cleverer heuristic function is required. To create this heuristic function, I implemented a heuristic function created by Kartik Kukreja which is much more advanced and takes concepts like coin parity, mobility, captured corners, and stability into account. This heuristic function was implemented in Heurisitc.py.

I implemented several agents to play Othello, but the two main ones were an agent based on Alpha-Beta pruning, and an agent based on Monte-Carlo Tree Search. The Alpha-Beta agent

uses the minimax algorithm with alpha-beta pruning to search through all possible moves and counters, and counter-counters to a given depth, and evaluates all leaf nodes using the heuristic function mentioned above. This agent also uses iterative deepening to make it an “any-time” algorithm that can run for a given amount of time and if it reaches its time limit, it just outputs the move calculated from its most recent depth calculation. This agent was implemented in AlphaBetaAgent.py.

The MCTS agent implemented a standard Monte-Carlo Tree Search algorithm with the “Upper Confidence Bound” algorithm for child node selection. This algorithm simply chooses children based on a combination of win-rate of playouts and slightly favoring nodes that were visited less. Then, after the tree is sufficiently searched, to pick the best move, we switch to the lower confidence bound to slightly favor nodes that are visited more. This algorithm is used widely in reinforcement learning but is also applicable to MCTS as in a sense, each playout causes the agent to “learn” about its state and if the move it made is good or bad. This algorithm is “any-time” out of the box, so no modification was strictly needed. However, since playouts must play through the entire game, I scaled the early moves to have proportionally more time to make a move because the playouts are longer in these cases. This agent was implemented in MCTSAgent.py

I also created several simpler agents for comparison such as a MaxStones agent which always tries to take as many stones as possible, a MinStones agent which tries to take the least number of stones as possible, and a RandomAgent which just makes a random move. These were implemented in OtherAgents.py as agents which operate with their own heuristic functions at a depth of 1.

The Outcome

I ran several experiments with these agents in order to gauge their relative playing-strength. The first experiment was simply to use myself as a benchmark to play against the Alpha-Beta agent and the MCTS agent. I would not consider myself a particularly good Othello player, and I am definitely not an expert, but I do beat all of my friends consistently and I know some basic strategy. With my skill, I was able to beat the MCTS agent consistently, but I was not able to beat the Alpha-Beta agent even at relatively low depth of 3 or 4.

I also ran a multi-round-robin tournament with all of my agents and evaluated their performance using a relative ELO system. An ELO system is meant to characterize the relative strength of players and is often used in 2-player turn-based games like Chess and Othello. The higher the rating, the better the player. The agents for this tournament were 3 Alpha-Beta agents with 0.02s, 0.1s, and 0.5s to move, 3 MCTS engines with 0.02s, 0.1s, 0.5s to move, and one of each of the extra dummy agents for comparison. Each agent played 20 games against each other agents, 10 games as black and 10 as white for a total of 160 games per agent. These games were split into 10 rounds and ELO scores were updated after each round. The following table shows the results for this tournament and each agent's calculated ELO, their total score out of 160, and their winrate.

	AB (0.5)	AB (0.1)	AB (0.02)	MCTS (0.5)	Max Stones	MCTS (0.02)	MCTS (0.1)	Random	Min Stones
ELO	1705	1702	1597	930	833	745	698	453	296
Score	149	146	122	85	54.5	55.5	45.5	39.5	23
Winrate	93%	91%	76%	53%	34%	34%	28%	24%	14%

The Discussion

From the experiments performed, the Alpha-Beta agent performed excellently and exceeded my expectations, while the MCTS agent was very disappointing. There are a few reasons why I believe this happened. First, the heuristic function that was used does a very good job of evaluating the position, and the strength of any minimax agent is greatly impacted by the strength of the heuristic function. Second, given the performance constraints of my computer, my MCTS agent was not able to complete very many playouts, and especially at the beginning of the game when playouts are longer. Because of this, the agent takes a couple of seconds to even search as little as 10 playouts, so in a tournament where there is less than a second per move, the agent cannot collect enough data to play much better than random. The key here is that the strength of MCTS in agents such as AlphaZero is a very strong policy function, and the computational ability to run thousands of playouts per move. In this case, the benefits of MCTS were not seen, and Alpha-Beta pruning triumphed.

In addition, more time to compute did seem to help the agents on average as expected, though the trend was not 1-1. I expect this is because at such low time to move, the MCTS agent is only able to perform one or two playouts which is hardly better than random play.

The Product

To interact with my best agent, I have created a display in pygame which allows the user to play a game against the strongest agent (AlphaBetaAgent). The display shows the board and all of the possible moves. To play, click on one of the possible moves and the Alpha-Beta agent will play their response with a time-limit of one second. To reset the board, press the r-key. Note: the aim of this project was not to create an incredibly immersive gameplay experience, but to evaluate techniques for creating agents to play Othello.