Run the program checkers with Python 3.5 by typing into the command line:

python checkers.py

This program has a dependency on the numPy 1.14 module.

The program will prompt for a sample filename, where the format is the first 8 lines represent the initial board position (first line being the second player’s home row), and the 9th and 10th lines are user player number and search time limit for the AI.

Note:

Player 1: X, Player 2: O (Kings are XX and OO, respectively)

There are two base classes for this program: Player and Checkers.

The Checkers class implements functions to handle all the routine/annoying stuff, e.g. reading in and parsing commands, printing out the board, keeping track of the history of the board, etc. Most of the stuff in this was used to debugging (e.g. keeping track of the history of the board was useful when trying to recall how the AI got into a particular position, and printing out the board the heuristic calculations at each ply was useful as well). It also implements functions to check for winner, end states. It is initialized with two Player objects and an initial board (a numPy array). It’s main function “run” is basically a game loop which maintains the current board state and lets the Player objects interact with it one at a time.

The Player class was more important. It implements functions for both AI and User classes, which are derived Player classes, such as getting locations of pieces, moving pieces and changing the board to “tell” the Checkers class which move to make. The important function in the base class is \_get\_moves, which takes a board and determines all the moves it can take. It first looks for all possible jump moves (as jump moves are forced) and calculates start and end positions through a recursive process. It then looks for all possible moves that aren’t jumps. There is a dumb implementation of select\_move, which is a function which selects a move to make randomly. This is overridden in the derived classes.

The AI derived Player class mainly overrides the select\_move method with the aid of iterative alpha beta pruned h-minimax search. To make things efficient, a search tree is initialized with the root as the initial board position. During the alpha beta search, the successive levels of board states are calculated *and stored* into the search tree. For example, at the first iteration, the alpha beta search calculates to the first level, so the search tree *calculates* and stores all the board states of the first level. At the second iteration, the alpha beta search no longer needs to calculate the first level board states as they are all in memory. This could be time efficient as the search tree becomes deeper.

The heuristic evaluation function used to estimate the value of a board position in the minimax search algorithm is what differentiates the derived AI classes. There are couple implemented in the code as you’ll see, but the best(?) one *should* be **BoardPositionEndGameAI**. The main idea for the heuristic was to prioritize piece count advantage, king promotion, end game positioning, center control, and some small random component (in this order). Piece count was just adding up all the player pieces and subtracting enemy pieces, with kings worth twice as much as pawns. King promotion was determined by average row distance from enemy’s home row, excluding kings. End game positioning kicks in after there are about 10 pieces left on the board, in which the player wants to decrease “distance” from the enemy when the player has piece count advantage and increase “distance” from the enemy when the player has piece count disadvantage. This “distance” is determined solely by the maximum distance between any of the player’s pieces and the enemy’s pieces (this may not be great and may be computationally not worth, admittedly). Center control was determined simply by the average distance of all the player’s pieces from the center of the board.

As I was coming up with ideas for heuristics, I tested them by basically having two AI Player objects fight each other many times and check the winning percentages. While this admittedly isn’t great, it’s a pretty good way to estimate progress when I’m too lazy to keep playing checkers over and over again…

Getting the weights for the factors in the heuristic function right was the tricky part, and ideally I think we could do something cool like a genetic algorithm which mutates the weights on the factors in the evaluation function in the minimax search algorithm, which I think could be really interesting. I also think you can do something like an Monte Carlo search to estimate the evaluation.

One thing that was tricky was *forced* captures, which I hadn’t considered initially when I noticed the AI taking stupid moves that jumped itself into double jumps.

I’d like to apologize also for the game display: I know X’s and O’s were not suggested but I didn’t find any better ascii characters, to be honest. I’ll update with a version with better characters if I come across any. Also, I am working on Windows and I had some trouble getting different colors into command prompt. Sorry…