Fakharyar Khan
ECE-469: AI
Professor Sable
November 6th, 2023

Project #1: Checkers Bot

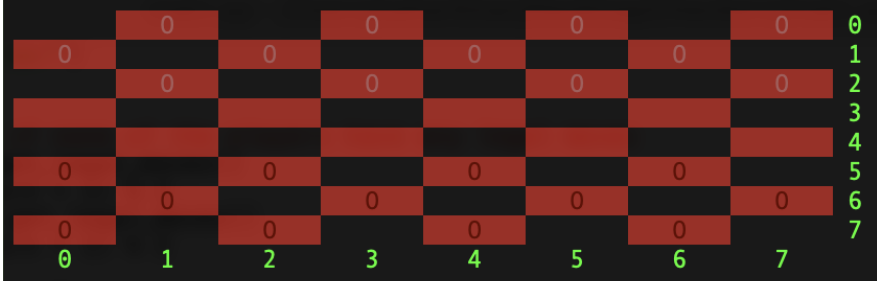
Program Instructions

To run this program, you should have Python 3.12.0 installed on your computer or any later version. You should have the three python files Computer.py, main.py, and Board.py in the same directory. You should then open up a terminal and cd into the directory where the files are located. Once there, you should enter in the following command: python3 main.py as can be seen below

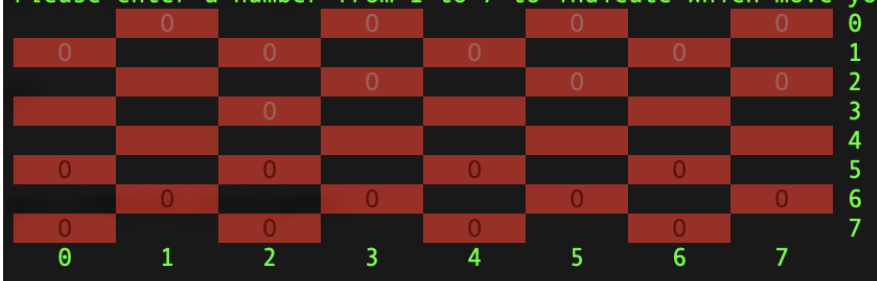
The program will prompt you for settings such as if you would like to load a file containing a game state, if you want the computer to play against itself, which color you want to be, and the amount of time you want to give the computer.

After this, the program will begin the game. When it's the player's turn, it will print out a numbered list of all of the possible legal moves that the player can make and will then prompt the user to select one by entering in the corresponding number for that move. When it's the computer's turn, it will notify the user of how many complete levels in the game tree it was able to finish and how many seconds it had taken to finish. After every move, the program will print out the updated version of the game. When the game is over, the program will let the user who won. A draw can occur if 20 consecutive moves have passed without a single capture or if both players have no legal moves.

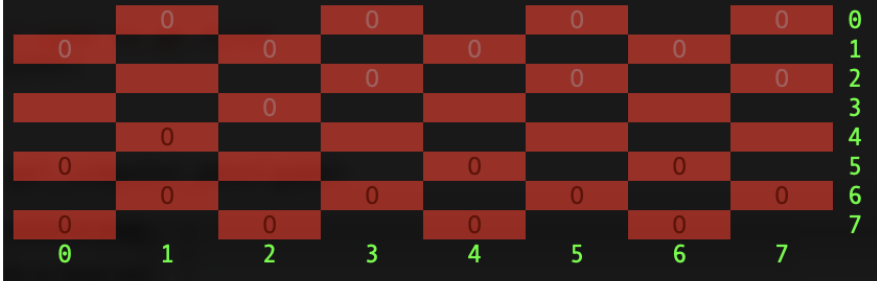
```
[saharkhan@Sahars-MBP Checkers % python3 main.py
Would you like to load in a game state from a file? (y/n) n
Would you like the computer to play against itself? (y/n) n
Would you like to play as black (0) or white (1)? 1
How much time would you like to give the computer? 3
```




```
1. ((2, 1), (3, 0))
2. ((2, 1), (3, 2))
3. ((2, 3), (3, 2))
4. ((2, 3), (3, 4))
5. ((2, 5), (3, 4))
6. ((2, 5), (3, 6))
7. ((2, 7), (3, 6))
Please enter a number from 1 to 7 to indicate which move you would like to make: 2
```




```
Searched to a depth of 8
Total time taken: 3.00448 seconds
```




```
1. ((1, 0), (2, 1))
2. ((1, 2), (2, 1))
3. ((2, 3), (3, 4))
4. ((2, 5), (3, 4))
5. ((2, 5), (3, 6))
6. ((2, 7), (3, 6))
7. ((3, 2), (4, 3))
Please enter a number from 1 to 7 to indicate which move you would like to make:
```

Implementation Details

For my heuristic function, one of the things that I tried doing was training a classification model on a dataset of checkers games to see if it could predict which player would win given a board state. I'm not entirely sure if the output would be zero sum but I think it should be: the output could be trained to represent the probability that a player would win. So if player 1's chance of winning increases by 10%, player 2's chances would decrease by 10%. I found a dataset of around 20,000 recorded games that showed who won and who lost for each game. For each turn in the game, I added the current state of the board into the dataset and labeled it as 0, 1 (draw), or 2 depending on who won the game. I wasn't sure if there would be enough information in the first 3 turns in the game for the classifier to be able to predict anything so I decided to drop those board states from the dataset. After doing that, I had around a million data points in my dataset. For the classifier, I flattened the board and fed it into an MLP with about 4-5 hidden layers. And when I ran it, it was able to get a training accuracy of around 90% which I thought was insane. But then I looked at the validation accuracy and it was just terrible, I think I was getting around 45-50%. My model wasn't too big so I really wasn't expecting it to overfit like that especially on a dataset that large. I think the problem might have been that the dataset was too small. It might be that there are so many possible board configurations that even a dataset that large isn't representative enough. I tried getting more datasets like that one but I didn't really have much luck. I tried to see if I could find a way to download a small part of the endgame tablebase for checkers but I couldn't really figure out how to.

After that I looked into common checkers strategies to build an evaluation function for which I had a lot more success with. I broke up the game into two stages: an early/mid-game and the endgame and created evaluation functions for each. I realized that this was necessary because when I had a single evaluation function, my program would never be able to finish a game. Its evaluation function was optimized for the middle game where there are a lot more pieces on the board and therefore more opportunities to capture and be captured. But when there are so few pieces on the board, it looks like all of its moves seemed equally valid to it so it would either move around randomly or get stuck in corners.

For the mid game heuristic, at first, I only had a weighted sum of the kings and soldiers on the board. With this, it was ok, you could definitely tell that the program had a strategy in mind. But it would always do things like leave its back row exposed. So I added in a reward for not moving the pieces in the back row and for getting the opponent to move theirs. I also rewarded the computer for having more pieces in key regions like the middle rectangular box in the board and the two middle rows. For the box, the idea was that pieces in that region have the most freedom to move around. Pieces outside of that box are usually pushed to the side where they don't have as much mobility. For the two middle rows, I wanted the computer to prevent enemy pieces from being promoted by preemptively blocking them from passing.

I also found that my bot was having trouble promoting its pieces. It looks like it thought that it was too much of a risk to get a king. So it wouldn't try doing things like moving a piece in between two opposing pieces to get into the back row which made it easy to gain an advantage over it. To remedy this, I rewarded the program for being in positions where the opponent can't jump them because they have another one piece backing the other piece or if the piece is

hugging the sides. Finally, I checked to see which player had the most number of legal moves. While this was a pretty expensive thing to calculate in the heuristic, I think it was worth it because it incentivized the computer to restrict the player's moves.

For the endgame, in order to stop the computer from running away when it has more pieces, I used the total distance between the red and black pieces in my heuristic. If the maximizer had more pieces than the minimizer then the heuristic incentivized it to minimize the distance between its pieces and the minimizer's. On the other hand, if the maximizer had fewer pieces, the maximizer was encouraged to run away and maximize the total distance. This made the computer much more aggressive and helped it win more games instead of stalemating.

Additionally, my program had a really hard time dealing with the 2 king 1 king endgame where the one king would move back and forth in the top left and bottom right corners of the board. To fix this, I rewarded the program not for getting the opposing king out of the corner but for occupying the other corner square that the opposing king would move into. This helped the program figure out how exactly it should drive pieces out from the corner.

Finally, to ensure that my program would go for a quick victory, I added in a factor to the evaluation score for winning the game that decreased in magnitude as the depth of the search increased. This made it so that if the computer was winning, it would go for a quick victory and if it was losing, it would prolong the defeat for as long as possible.