Brent Kirkland
6365829
brentkirkland@cs.ucsb.edu

Machine Problem 2

Architecture

Initial setup: To quickly store all the data, I initialized a matrix of size -n (which is specified by args, or default 11). After that we initialize who is dark and who is light. Once the program knows this, computer moves are of value 1 and human moves are of value -1.

Choosing the Best Depth: The second part of the architecture is selecting where to play a piece. In order to do this, I first select a depth value based on how many plays are already on the board. Early game plays are less important as later game plays. The depth moves from 1 to 3 as the game fills with tiles.

Alpha-beta-minimax: After the program knows it's depth, it performs the minimax algorithm. It uses alpha-beta pruning to avoid overlooking nodes. When beta is <= alpha, the program knows it can prune leaves. The alpha-beta-minimax finds the highest weight which ensures a best move by the computer.

Calculating Weights: The minimax algorithm most importantly depends on how plays are weighted. To do this, the current matrix (at terminal nodes) is converted into a string. This string contains all rows, columns, and diagonals. Then a substring search is performed for each unique play. These plays have unique weights defined by me from games I have played with the given jar file. Unique substrings for each player are searched which gives us two weights: computer_weight and human_weight. The total weight is then calculated by computer_weight - human_weight.

Placing each Move: Once the best weight has been calculated, the computer plays the selected tile then switches turn to allow the human to play. The depth is then re-examined, the alpha-beta minimax algorithm runs, which then picks the best calculated weight. This process repeats until the game is over.

Search

To perform the search the alpha-beta minimax algorithm was used. To use this algorithm a selected depth is used (mentioned above). This depth tells the program how many plays to look ahead. By examining x plays ahead, the program can see potential moves from the human player. By limiting the depth, we can benefit from a faster search or a more informed search.

At each depth, the program exams every available move for the whole board. It then calculates the max or min depending on whose turn it is. The goal for the computer player is to maximize its score while minimizing the humans score hence the name minimax.

For the alpha-beta pruning part, the program tries to avoid over-searching by knowing which values it has already seen. If Beta is less than or equal to alpha, the minimax algorithms knows to prune and skip that part of the search.

After all nodes of the tree is examined and weights have been calculated, the program has an informed decision on where to play next.

The alpha-beta pruning made the minimax algorithm much faster. Not only, by changing the depth as more pieces are played, make the search better. By changing the depth slowly, this allowed a better pruning space which helps make quicker and more informed decisions.

Finally, using a string to search for potential moves makes the problem much easier. I had initially thought I could sum the rows, columns, and diagonals, but that made for special cases hard to catch. Looking for select strings allows the program to be more fined tuned.

Challenges

The hardest challenge for me was learning how the alpha-beta part of the minimax algorithm works. It took me sometime to learn how to exactly implement the algorithm. Also, as said above, defining weights was tricky. The program took about 5 days of work to understand all the special cases to the connect 5 game.

Weaknesses

My biggest weakness is python and memory usage. Python is much slower and I wish I had used a faster language. Creating all the many different string combinations is also very memory intensive.