Aerilynn Nguyen
CS5100
Project Write-up
Group mates: None

For this project I programmed a game of Othello in Processing, a software sketchbook (please note Processing is required to run the program) with four different types computer opponents. This game can be played computer vs computer or computer vs human. Three of the computer agents were 'intelligent' while one was 'unintelligent' in that it simply picked random moves to play. The algorithms used for the intelligent agents were: greedy best first search, minimax, and q-learning.

One of the most crucial parts of writing an algorithm for an intelligent agent is coming up with the right evaluation function that can accurately capture the value of a given state. For Othello, what comes to mind immediately is to choose the number of flipped pieces as that is an easy measure of what is gained per turn. However, as the game progresses more pieces tend to get flipped per turn because there are simply more pieces on the board – which of course can't be the single indicator as to how well a player is doing. For this reason, I chose to still keep the number of pieces flipped as a evaluation measure but gave it lower weight than the factors that will be described next. Being that this is a strategy game, we should also keep into account the position of pieces on the board as well. Corner pieces in particular give a player an extra advantage as those pieces are stable in that they can never be flipped. Another way to evaluate a board is to count how many additional moves to newly placed piece might open up for the player. In the end, I decided to use number of pieces gained, number of new corner pieces, and number of new possible moves as an evaluation measure for a move. Number of corner pieces were given the highest weight in the total sum while number of tiles gained was given the least weight.

With an evaluation function now defined, a greedy best first search algorithm could be effectively implemented by always choosing the move/board with the highest evaluation value. The same evaluation function was also used for the minimax algorithm. For the q-learning algorithm, however, I built a reward matrix instead showing the value of each piece placed in each square on an 8x8 grid. This reward matrix was constructed based on the idea of stability, briefly mentioned earlier when considering the evaluation function. Corner spots have the greatest stability in that they cannot be flipped at all once they are placed so pieces placed there get the highest reward. The positions adjacent to the corners get penalized because once you place a tile there, the chances of you also being able to place a tile in the same corner are greatly reduced. Edge positions, while not completely stable as corner positions are given about half the value as corner positions because they can only be flipped horizontally instead of vertically and diagonally and therefore semi-stable. The positions next to the edge positions (towards the board) are penalized as well for the same reason the positions adjacent to the corner positions are – although they are only penalized half as much.

To measure the performance of each algorithm, I pitted each of the intelligent agents against myself and the unintelligent agent for 10 games. For reference, when I played against the unintelligent agent, I won almost every time putting in minimal effort. The q-learning agent played about 1000 games against the unintelligent agent before its q-matrix was fully populated and stable. All intelligent agents beat the unintelligent agent about 9 out 10 times. Although each intelligent agent had the same score against the unintelligent agent (about a 9:1 win to loss ratio), when playing agent the agents myself, I noticed the minimax based agent making decisions much slower than the other two algorithms even at a depth of 2 or 3. The q-learning agent performed faster as expected given that it could get q-values in constant time by array indexing.

While this game and the agents programmed for it certainly work, the runtimes for the algorithms themselves are slow and no human player would have the patience to wait that long for a computer to make a move. In the future, if I were to do this again, I would like to focus more on the efficiency of these algorithms like making them run in constant or at the least, linear time.