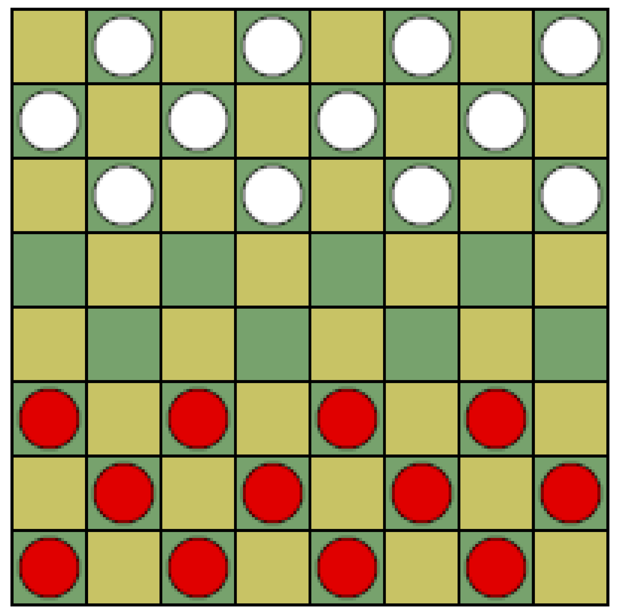
GAMES AND PROGRAMMING part II

Aaqel Shaik Abdul Mazeed, Abeer Fatima, Ahmed Shah, Jamie Hayes, Joshua Samuel, Moe Hishmeh, Nada Adzic Vukotic, Samuel Effendy, Syed Hussain, Umar Chaudhry, Vincent McNulty, Vira Kasprova
University of Illinois at Chicago

MSCS Undergraduate Research Laboratory

UIC

Checkers



We were able to represent any board position by having an array where each element represented one of the squares on the board. We then filled any empty spaces with 0 and the rest with an instance of a Piece class. Each piece has a color, position, and an is King value. Now to allow the pieces to move we created a function to find the legal moves any piece could make we did this by taking a piece and looking along both its diagonals to see if there was anywhere it could move to. And if it happened to jump over an opposing piece we would recursively call the function again to allow double and triple jumps. We added a few more simple functionalities such as king promotion, win detection, and a starting position.

We used the Minimax algorithm initially. However, it's important to note that this method is only as effective as the scoring function it uses, so after some trial and error this is my function:

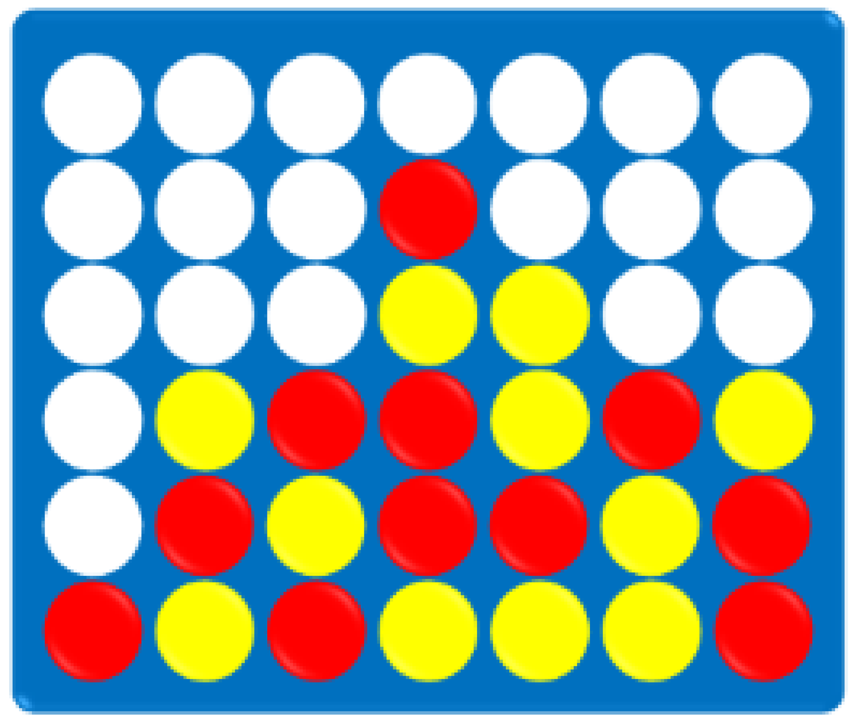
(White Pieces - Red Pieces) + 0.2*(Possible White Moves - Possible Red Moves) + 0.5*(Development Score)

The Development Score is a function we made to encourage the AI to move its pieces forward. It is the sum of the row numbers of each piece and kings are worth 10, this incentivizes the AI to push its pieces to the end and get kings. After all this, the AI works quite well.

The next optimization is to add a Transposition Table. This is a table of all the board positions that the AI has seen in the past, it'll then save that position's best move and score at a certain depth of search and return it when it sees this board position again. However since each board was saved as a Board object we need to find a faster way to search and store these positions, so we created a hash function.

We then simply had the AI play a random player over and over again until it knows a large number of board positions. Currently, it knows about 321,403 board positions. Both of these optimizations have allowed Minimax to easily look up to 5 moves ahead while it used to struggle to look 3 moves ahead, on my machine. It always beats a random player and never makes unintelligent moves against trained players. The only problem is that it is slow, so if we can optimize it not only will it be faster but it could look more moves ahead.

Connect 4

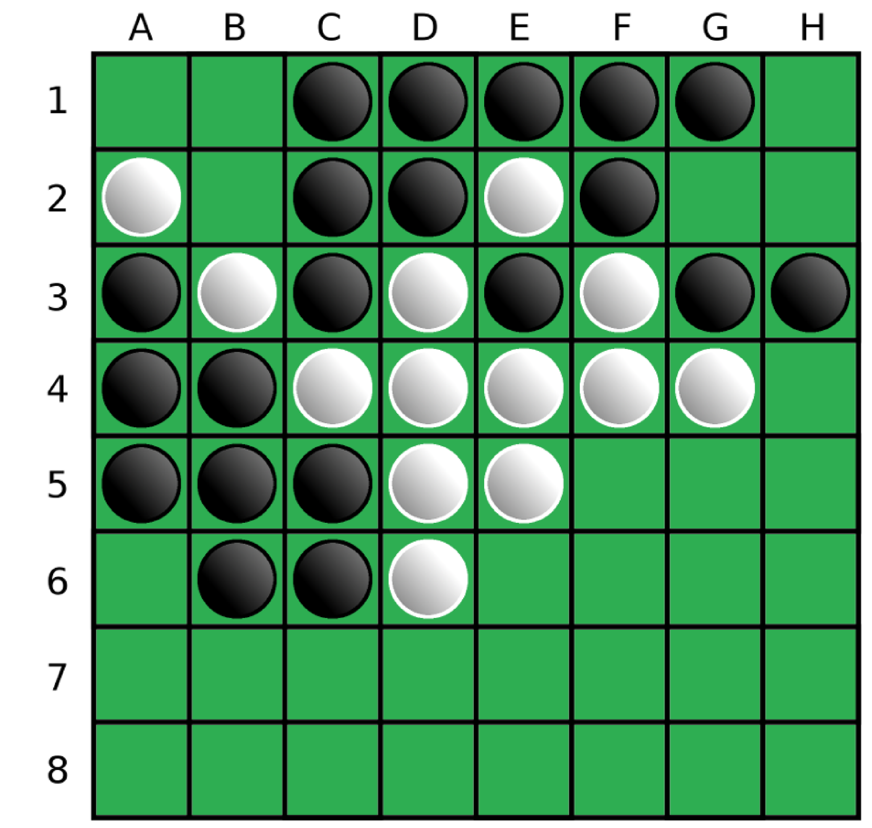


After trying to find a machine learning algorithm to simulate and train from games of Connect 4, we found the most success through Q-learning. Before creating a Python program that uses Q-learning, we wrote a starter program that would contain the functions needed to play a game for Connect 4 such as a game state evaluation function, and that program would allow for basic simulations between random players and display results of those simulations. Connect 4 is played through a game board, so we parametrized this board as one Python list that would contain 6 lists, and each of those lists would contain 7 elements. This creates a 6x7 2-D Python list similar to a 6x7 Connect 4 board. As humans, we can see pieces on the Connect 4 Boards as X's and O's or Red's and Blue's, and we can parametrize those pieces with numbers mapped to those human-readable values. In our program, we parametrized an empty space as 0, an X piece as 1, and an O piece as -1. Hence, when initializing the Python list serving as a game board, all of the elements in the lists will be 0s. The next step to simulate Connect 4 was to create functions that found all the available moves for a player and evaluate the game state. After all these functions were successfully programmed, we was able to simulate games between random players and compute some interesting results. After 10,000 games, player 1 won 55.41% of the time, player 2 won 44.33% of the time, and ties occurred 0.26% of the time. From these results, one can see how the first player to move has an advantage and ties are rare in the case of two players playing randomly.

After random simulations, we implemented Q-learning. Eventually, the program was running successfully, and we experimented by having two computers train with the Q-learning algorithm by playing against each other for 10,000 games and then, one of the trained computer players played against a random player for 10,000 games. We wanted to gauge the effectiveness of the Q-learning player by seeing how well it could defeat a random player, and the results were that the Q-learning player won 86.35% of the time, the random player won 13.65% of the time, and there were no ties. From these results, we can see that the Q-learning player is in fact learning.

After this data was found, the program had the two trained Q-learning players play against each other for 10,000 games. The results were the first Q-learning player won 52.4% of the time, the second player won 47.1% of the time, and there were 5 ties.

Othello



Othello is a perfect information game because every player is perfectly informed of every event in the game, including the initialization of the board. It is also zero-sum, since every gained advantage of a player is equal to the gained disadvantage of their opponent, e.g. every tile gained by a player is lost to another player. Next, it is deterministic, because there are no random elements in its rules. This game is a perfect testbed for reinforcement algorithms like Q-learning, since there is a clear distinction between reward and consequences as well as a numeric value for each action to train the agent.

The board is initially empty but for the four center squares, which feature discs on the two squares on the major diagonal ([4][4] and [5][5]) and the two squares on the opposing diagonal, respectively [4][5] and [5][4].

The player who moves first is referred to as "Black" and the player who moves second is referred to as "White". A disc must be placed on an empty square in order for there to be a sequence of one or more discs belonging to the opponent, followed by one's own disc, in at least one direction (horizontally, vertically, or diagonally) from the square played on. In such a series, the opponent's discs are flipped and changed to one's own color.

Othello's state space has a maximum of 60 movements and a dimension of around 1028. Two players use 64 two-sided discs that are black on one side and white on the other to play Othello on an 8 by 8 board. The discs are placed on the board with the white side up by one player and the black side up by the other player.

We chose to write our parameterization in Python due to its accessibility and library support for machine learning. To represent the board in our version, we will utilize an 8 by 8 character array. Instead of using black and white pieces, we will use Xs and Os. The lower left corner is at coordinate [0][0], while the upper right corner is at coordinate [7][7]. Since the rules for movement in Othello are complicated, we broke the movements down into three functions. The validMove() checks if the move is valid or not. The move function takes as input the board, the XY coordinate to place the piece, and the piece we are placing (X or O). If a piece can be flipped in any of the eight directions and the space on the board is unoccupied, the move is valid. We check all adjacent cells to our current location by calling checkFlip() function for left, right, down, up, down-left, down-right, up-left, and up-right.

Chess

Chess is the ultimate game that can appear as a challenge in the context of game programming. Almost thirty years ago it was unfathomable to have machines that beat the leading players. Nowadays it is almost unfathomable to have any leading player that can come close to what a machine can do. Not only machines can easily beat all humans - they have also changed the landscape of the game completely. They have revealed strategies that were very little contemplated before. They have answered questions that remained open for a while. And the journey goes on as people try to perfect the use of computers in learning, understanding and appreciating chess.

Chess will be our exclusive goal of a similar project for next semester. We will try to approach the subject of chess programming from different angles and to produce interesting code from scratch in many different directions. So the journey goes on...