Checkers AI with Minimax

Eric Clifford, Connor McAllister, Ramesh Poudel

Purpose:

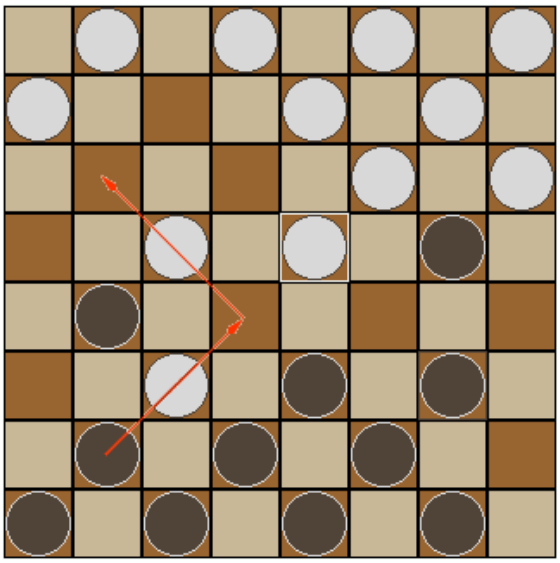
The goal of this project was to build an AI checkers solution, such that an automated bot could play against a human opponent and perform relatively well. The bot's moves should not be random, and the moves should have some sort of strategic purpose to them to make gameplay competitive.

Background - Checkers:

Checkers is a game played on a board with squares in an 8x8 pattern which alternates between two colors (for example, red and black). Each player begins with 12 pieces on opposing sides of the board, where every piece for both players must remain on a particular color of square (for example, all pieces stay on black squares) and only one piece can occupy a square. The color of a player's pieces are uniform for them, but different from their opponents. So one player may have all red pieces while the other has all white pieces.

Players alternate turns in which they can move a single piece toward their opponent's side in a diagonal direction (for example, square 3,4 to square 4,5). If an opponent's piece occupies a space to which the player's piece could move, and the square on the other side of the opponent's piece is empty, a player may move to the empty square (jump over the opponent's piece) and remove the opponent's piece from the board. These jumps can be chained such that, after a jump, if the jumping piece finds itself in a similar situation, it may jump the next piece until there are no more jumps left. However, the piece can only move toward the opponent's side of the board. Figure 1 displays this concept. Here, both white pieces which were jumper will be removed.

Figure 1: A 'double jump' in checkers.



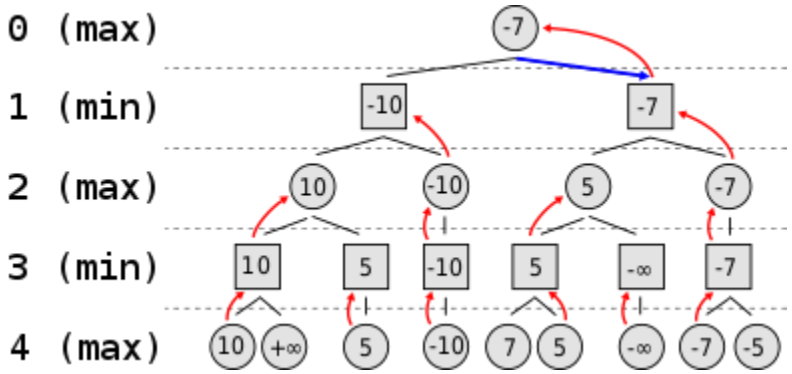
When a player's piece reaches the end of the opponent's side of the board (the row furthest from the opponent's side), that piece is made into a 'King'. A King piece no longer has the constraint that it must move toward the opponent's side of the board. It is allowed to move both toward and away from the opponent's side of the board. The game ends when all of a player's pieces are removed from the board, where the player with remaining pieces is the winner.

Background – Minimax Algorithm:

The minimax strategy is a form of adversarial search in which one player is attempting to maximize a utility value of a move while the other attempts to minimize it. This works best in a two-player situation where players alternate turns, and in which an advantage for one player is a disadvantage for the other. Checkers and chess are perfect examples of this type of game.

The minimax search uses a tree to consider all possible moves over the next *n* turns. The leaves of the tree represent utility scores of each possible state after *n* turns. Each level of the tree alternates between the 'max' player and the 'min' player, such that the goal of the max player is to choose the highest-scoring node and the goal of the min player is to choose the lowest-scoring node. These decisions propagate up to the root node, which informs the decision of the player (usually max) to which that level corresponds. Figure 2 displays this concept.

Figure 2: Visual representation of Minimax Search algorithm.



Implementation, automate function:

Our project attempts to build on top of a pre-existing checkers game written in Python. The source code for this checkers game can be found here:
https://github.com/techwithtim/Python-Checkers
This checkers game is meant to be played between two human players, so each move for both sides is human-driven. The first step that we had to take was to automate the move of only one of the players (in this case, the white pieces). To do this, the change_turn function in the game.py file was modified such that, if the current player is Red, the player is switched to White

and a new function named ‘automate’ is called. The automate function first sets a Boolean tracker variable named ‘automating’ to true. This prevents the game from changing turns while the minimax search is carried out. Next, the automate function calls the minimax algorithm, which returns the best move for White. The piece is then moved and automating is set to false. The current player is then switched back to Red and the game continues.

Minimax function:

The minimax search being called in the automate function occurs through a function called ‘minimax’. The minimax function takes in a boolean value ‘final’. This represents whether either player is down to their final piece. If final is true, a minimax move/score tree of depth 2 (one move) is obtained. Otherwise, a minimax move/score tree of depth 4 (three moves) is obtained. This is done through a function called ‘get_move_tree’, which will be covered soon. After obtaining the move tree, a breadth-first search is performed on the tree such that at each level, either the maximum or minimum value (alternating) is taken and passed into the prior level. When the best move is obtained, it is then passed up to the calling ‘automating’ function.

Get_move_tree function:

The get_move_tree function being called by the minimax function is passed the ‘final’ variable from the minimax function. It is also passed a Board object, which represents the board on which the game is being played, or a theoretical board for which a move might occur (when testing possible moves). Finally, a ‘level’ variable is passed into this function where 3 – level indicates the number of moves ahead to consider. If the ‘final’ variable is set to true, this number will initially be 2. Otherwise, it will initially be 0. This function is also a recursive function, allowing it to look ahead up to 3 moves and return a tree. It will return a move/score tree which is stored in an array called ‘all_moves’

When the get_move_tree function is called, it first obtains all of the white pieces from the provided board. For each piece, every valid move for that piece is analyzed. For each valid move, the board and it’s necessary attributes are copied to a temporary Board object, which is then passed to a temporary Game object (so that any testing does not affect the actual game being played). The valid move is then performed in the temporary game.

After the move is performed, if the ‘level’ value is 0, an array containing the piece, the move, and the result for get_move_tree at level 1 (with the new temporary board passed in) is stored in the ‘all_moves’ array. If the level is 1, the process is instead carried out on red pieces. After each move, an array containing the piece, the move and the result for get_move_tree (with the new temporary board passed in) at level 2 are stored in the new all_moves array. If the level is 2, the initial process is again carried out on white pieces. If ‘final’ is false, after each move, a score obtained from the get_score method (explained later) is stored in the new all_moves array. If ‘final’ is true, an array containing the piece, the move, and the score from get_score is stored in the new all_moves array.

Put in simpler terms, the get_move_tree returns an array containing arrays. If the initial level is 0 (final is false), the array is structured as follows:
[[whitepiece, move, [redpiece, move, [whitepiece, move, score], …], …], …]
Every white piece/move combo has a number of red piece/move combos, which all have a number of white piece/move/score combos. If the initial level is 2 (final is true), the array is structured as follows:
[[whitepiece, move, score], [whitepiece, move, score], …].

Get_score function:

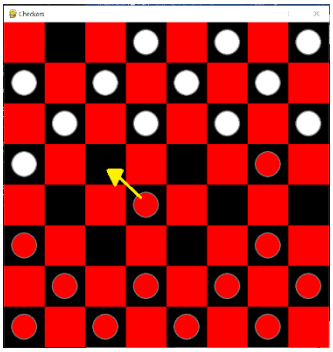
The get_score function is pretty simple. It takes a look at the board being tested. It counts the remaining white pieces, red pieces, white kings, and red kings. The following formula is used to determine and return a score value:
white pieces – red pieces + (white kings * 1.5 – red kings * 1.5)
This score is then passed into the calling get_move_tree function, to then be passed to the minimax function alongside pieces and moves within the array being passed.

Result:

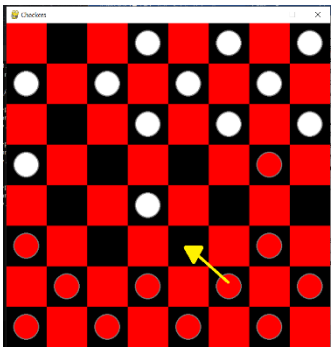
What results from the mentioned changes to the initial python code is a working checkers game in which a bot will play the moves for all the white pieces. The bot's moves are strategic and effective, and it has been a challenge to try and win the game against it. In the process of creating these changes, a couple of problems were encountered. The scoring didn't seem correct, and this was due to a problem where the get_score_tree function wasn't considering all valid moves. This was solved by updating the attributes for the temporary board when testing future moves. Another issue was that the game would crash right before a final move was made. This was due to an 'index out of bounds' error when trying to traverse the score tree. This was solved by introducing the 'final' variable and beginning at different levels depending on whether or not a player was down to their final piece.

These updates helped to inadvertently solved another error, in which the bot would play relatively well for most of the game, but would not make the final move to defeat the human player (it would avoid capturing the player's final piece). To solve this issue, we also needed to fix a bug in the initial program in which the stored tracker for the number of kings would increase every time a piece reached the king position, even if that piece was already a king. Before this change, the bot's pieces would just bounce back and forth between king position and the closest square so as to boost that value for the total number of kings.

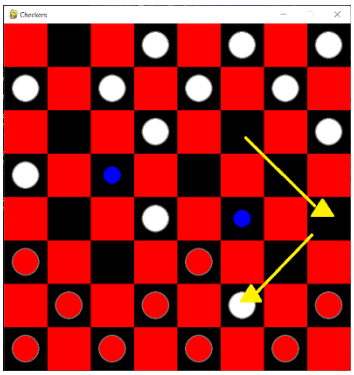
Attached are pictures from the first few moves of a game being played. To discuss, we will use a (row, column) notation to indicate piece locations, where row 1 indicates the top row of the board and column 1 indicates the left most column. The user is controlling the red pieces, and the bot is controlling the white pieces. In the first picture, the user moves the piece at (5,4) to (4,3).



After the user makes the move, the bot responds by capturing the piece moved to (4,3) with the white piece that was at (3,2). The bot incentivizes capturing this piece over other moves that could've been made. Notice that the red piece at (4,7) is available for capture as well. The user then moves the red piece at (7,6) to (6,5).



The bot then responds by capturing 2 red pieces at (4,7) and (6,7) with the white piece at (3,6). Here, the bot incentivized the only move that would allow it to capture 2 red pieces, choosing this move over all other moves. To see a complete game being played, refer to the included file "CheckersGame.mp4".



Possible updates:

A couple of possible improvements come to mind when looking at this program. A difficulty level could be introduced, such that the number of future moves which the bot considers could be scaled by difficulty. Testing on the scoring metric could also occur, as to determine the optimal scoring metric for the bot to use. Finally, we could allow the player to choose whether to play as white or red, as the only choice now is for them to play as red.