A Python Implementation

Jennifer You

Julien Yang

Emy Xiao

Checkers

May 3rd 2024

PART A) MINI-MANUAL

Game Rules:

1. Checkers may only move diagonally forward on dark squares.
2. Players can capture opponent pieces by jumping over the captured piece, skipping a square on the diagonal. The same player can continue capturing pieces by moving the SAME checker until they run out of pieces to capture.
3. Checkers become kings when they reach the other side of the board. A king may move diagonally forward AND backwards on dark squares. The same rules are applied to capture opponent pieces.

4. A player wins by capturing all of their opponents pieces or once the opponent is unable to move on their turn.

User Manual

To play the game once the program is running, the player may select the piece they would like to move by clicking on the piece. The game will display the selected piece and its possible moves in green. The player can then choose to select another piece to view and choose amongst its possible moves or move the selected piece by clicking on one of the possible moves displayed in green.

Once the player has moved a piece, the program will generate and execute a move of its own. The game will automatically update and display the game board following the computer's move.

The player can continue playing moving another piece or quit the game by simply closing the window. To start a new game, the user may run the program again.

PART B) PROGRAMMING IMPLEMENTATION

Board Implementation

Summary

The general idea of the game implementation comes from A3 (from programming 101). As with Othello, Checkers is a game with an 8x8 board that can be represented as a matrix. The board functions are then imported into the game implementation which is constituted of all of the necessary functions to make the game run and in this case, connect it to the UI interface.

1. Implementation - checker_board.py

The board is represented as an 8x8 matrix created with a list with 8 sublists each containing 8 elements. When we initialize the board using b_create(), we used a for loop comprehension to construct the matrix. The default value of each position is 0, which is a placeholder meaning that the position is unoccupied. Further functions are in the board implementation to facilitate the usage of the board. A couple important functions are b_eval(board,row,col) which is used to evaluate the value at a position of the board, b_count(board,value) which is used to see if one of the game over conditions is fulfilled, and b_copy(board) which returns a copy of the board.

2. Player Values

The arbitrary value given to the HUMAN player is 1, and the value given to the COMPUTER player is 2. Their kings are differentiated with the player value +2, so the HUMAN king is of value 3 and the COMPUTER king is of value 4.

Game Implementation

Summary

The skeleton of the game implementation is also similar to the Othello implementation from A3. A basic turn of the game would require an input from the player (in the form of clicking a pawn of choice and a position to move it to), followed by running a turn of the game using game_turn(board, h_pawn, h_move) to execute the human move and run a move of the computer afterwards. This would conclude one whole turn of checkers. Functions are used to find all of the legal moves of each pawn and some optimization is used to offer move-choosing functions of differing performance. The three different implemented functions to find the optimal move are the following:

_*choose*_move_random (random move choice)

_choose_move_greed (values capture over other moves. If there are no captures, a random move is played)

_choose_move_recursion (checks through a recursive implementation of a graph using BFS to a chosen depth to predict opponent moves)

1. Starting a new game - game_start

To start a new game, game_start is called to create a board with the starting position for both players. The board is created using b_create from the board implementation, and

for loops runs through all of the starting positions for each player and changes the positions to their respective values using b_put.

2. Choosing a move - Pawn & Move format

For every turn, the human is required to input a move in the form of a tuple with two tuples of coordinates. The first one represents the position of the pawn and the second one represents the final position of the pawn. It should look like this ((row,col),(row,col)). Kings are represented with an extra element 0: (row,col,0). In a similar fashion, a capture move is represented with an extra tuple with the coordinates of the captured piece: (row,col, (capt_row,capt_col)). The reason for this implementation will be discussed in the next section. The pawn and move chosen by the HUMAN is then inputted into the game_turn function which will execute the move (and any further capture moves). Before the end of the turn, the computer will automatically choose a move of a similar format using one of the three functions mentioned previously. That move will be executed in the same manner as the HUMAN move.

1. Finding legal moves - legal_moves dictionary

Finding all of the legal moves of either player is crucial, whether it is to check if the move a human chose is legal (and to show legal moves on the UI) or for the computer to be able to choose a move. We represent all of the moves of a player with a dictionary: the keys represent the pawn that they wish to move, and the values are the possible moves associated with said key. To do that, it is important to differentiate kings from other

pieces as they can also move diagonally backwards, giving them more move options. To do that, we append a 0 to each of the cases that have the value of the player + 2. A player king at (2,3) would thus be represented as (2,3,0). For pieces with 3 elements (kings), we evaluate all 4 diagonals, which are all of the possible moves. From said moves, we remove the moves that would lead to an occupied square or off the board. If the move leads to an opponent piece, we can check if it is capturable (if the further diagonal is free). If so, we append that further diagonal as a possible move, with an extra element that represents the diagonal in between. If the king at (2,3) can capture a pawn at (1,2) and end up at (0,1), we represent that move as (0,1,(1,2)). Legal moves are added into the dictionary of moves, which is returned after all of the pieces have been through the for loops. The same process is done with regular pawns, but without evaluating the backwards diagonals.

2. Running a full turn of checkers - game_turn

Ignoring the global calling at the start (we will come back to it later), the function checks if the chosen move is a capture move and will then execute the move. The pawn position is edited to be empty and the piece value is set at the final position, “moving the piece”. If the move was a capture, we check if there is another capture move that is available with the same piece. If so, we cut the function short by returning (with the second capture) which is then used as the “h_move” input to rerun a game_turn. The H_CAPTURE and h_capt_moves global variables are used in the UI implementation to rerun a game turn with only that piece and that move being legal if there are captures in a row. If the move

is not a capture or there are no further captures, it is the computer's turn. The computer chooses a move using one of the three functions which is then executed in the same way as the HUMAN's move. If the move is a capture, we check if there are further captures directly in the function, which is why the whole computer move code is in a while loop. When the move is not a capture or there are no further captures, the function is done running, and a turn of the game is played.

3. Choosing a computer move - 3 algorithms

For the random move selection, all of the moves of each piece are appended in a new dictionary (excluding pieces with no moves). Then a random move from a random piece is chosen using the randint method from random.

For the greed move selection, capture moves are prioritized (the moves of length 3). If there are none, we choose a random move.

For the recursive move selection, a BFS approach of a graph with all possible board states is used to find the optimal move. Each state is assigned a value depending on some factors like if there is a capture or if the next move will lead to a capture or if the move leads to no possible moves from the opponent (which is a win). The optimal move is returned through multiple layers of recursion. At depth 0 (the deepest layer or base case), we choose the best move using the greed algorithm. If there are no more moves, (-1,-1) is returned which is used to tell that game_over needs to be evaluated.

1. Ending a game - Checking for possible game ending conditions

   The two conditions for a game to end are implemented in the game_over function:

   1. One of the players has no more moves: In this case, the other player wins

   2. One of the players has no more pieces: In this case, the other player wins

   Game_over is evaluated after every move.

User Interface

Summary

The user interface (UI) is built using the Python toolkit Tkinter by creating an interactive board through the canvas widget. The interface registers the user's mouse clicks and finds the corresponding board matrix coordinates to a chosen move. It then inputs these board coordinates into the game's main program which will modify the current game's board matrix. To update the UI with the new board, the program refreshes the displayed board with the new board matrix

.

2. Creating the board

To open an application window, we first create a Tk object using Tkinter, which we will run in a mainloop. We then created a canvas object of the size of the board in said application window.

To display a board on the canvas widget, we wrote a function to iterate over the board matrix and create corresponding dark squares and checker pieces onto the blank canvas. The function first iterates over a simplified version of the board matrix with only dark squares to create a rectangle representing the square in the correct coordinates. It then iterates over a user inputted board matrix to create ovals in the correct coordinates on the canvas. The oval color is determined by the type of piece (1, 2, 3, 4) as identified in the given matrix. When we first start the game, we simply display the board with pieces at their initial positions in the matrix.

3. Playing the game

Once run, the game waits for the user's input through mouse clicks, which are bound to a function and will run said function. The function first uses coordinates of the clicked point to identify the clicked square on the board. It will then use if and elif statements to differentiate whether the user has clicked on a checker piece to view possible moves or clicked on a possible move to input their chosen move.

In the first case, the function will input the selected checker piece's board matrix coordinates into the legal_moves() function to obtain the list of potential moves for that selected piece, which we refer to in the code as LEGAL_MOVES. As this list contains the board matrix coordinates of potential new positions of the selected piece, we iterate over this list, use a separate function to find the specific square at those coordinates and fill it in green to display the possible moves to the user.

In the case where the user clicks on one of the squares listed in the list LEGAL_MOVES, we make sure the green squares are filled again with their original color and register the board coordinates of the selected square. These coordinates, along with the coordinates of the selected piece and the matrix of the current board are inputted into the game function, game_turn(), which will update the board matrix. We then update the user interface by calling the function to display this new updated board matrix.