Checkers Project Report

Instructions to compile and run the program:

- Install Python 3.
- Install the packages 'random', 'copy' and 'numpy' using 'pip install numpy'(and others in a similar way).
- The easiest way to have these packages pre-installed is by downloading Anaconda with Python 3 version.
- In your command prompt, change to the directory where the file is present. Run the command 'python simple_checkers.py'. (Basically, the command is 'python filename.py')

Design Description:

I have designed the game using 5 classes:

Board: This class when instantiated, creates a default board of size 6 x 6 with Blacks and Whites' positions as mentioned in the project question. Basically, I created a Multi-dimensional array (list of lists) to represent the board squares. They are indexed with (row, col) numbers. Blacks are represented with a 0, whites with 1 and empty spaces with -1.

This class has methods like showBoard – displays the board; legalMoves – calculates the legal moves available; areJumpsAvailable – checks for capture moves; moveFromTo – moves a piece from a position to another position; calculatePositions – returns the (row,col) positions of black and white pieces.

Checkers: This class has few member variables like turn – maintains the players' turns; difficulty – takes difficulty level; leftOnBoard – count of blacks and whites on the board; depth limit – limits the depth search of algorithm based on level of difficulty.

This class has few member functions like play – main control of computer and human's turn is here; isGameOver – checks if all the pieces of any player are captured/ or if no legal moves are available; utility – returns the utility values of terminal states; alpha_beta_search – returns the best action; max-value and min-value – these two functions check if terminal state is reached, or if depth limit is reached, or calculate the value of v and return the best action to alpha_beta_search function; evaluation function – returns a score as described in the previous page.

AB_Properties: This class maintains the value v, max_depth, nodes generated, #prunings in max-value and min-value functions for each action.

AB_Board_Player: Maintains the board state, current player and original player values during the recursive calls of max-value and min-value functions.

Move: Maintains the start, end positions of each move, whether a jump has taken place, and the pieces that got jumped over information.

Program Description:

When the program is run, you will have the option of choosing the difficulty level. The board positions are displayed. You will always play Black and you get the option of moving first or second. Then, the legal moves are displayed to you(from and to board positions). When jumps are available, only they are displayed as legal moves. Once you make a move, then its Compuer's turn. The program then calls the alpha_beta_search function, calculates utility values for each available action. If depth limit is reached, it uses evaluation function to estimate the utility values and returns a best action. Then Computer makes a move and displays the statistics as mentioned in the project question.

If all pieces of any player are captured, program stops and displays a winner. If there are no legal moves for both the players, program declares a winner based on the # pieces remaining. It's a draw if same # pieces are left.

Terminal States and Utility values:

- Max wins (i.e) Computer, Utility value = 1000
- Min wins (i.e) Human, Utility value = -1000
- Draw, utility value = 0

Evaluation Function:

eval_score = (70*(black_at_white_end - white_at_black_end)) + (50*(black_at_white_half - white_at_black_half)) + (30*(black_at_self_half - white_at_self_half)

The variables in the above formula explain themselves.

Basically, at each board state, the program counts the # blacks and #whites in 3 kinds of positions(opponent's farther end, opponent's half of the board and self's half). Then the evaluation function returns a score with weights multiplied to scores at each positions. It returns eval_score when the player is black and negative of eval_score when the player is white. The returned score is always between (-1000,1000).

I have chosen higher weight if the player reaches his oppponent's farther end because that piece can no longer be captured.

Difficulty Levels:

- **Level 1:** For level 1, I have limited the depth search to 4. Also, the evaluation function returns a random_score (i.e) random integer between (-500,500) as its score. It doesn't use the eval_score mentioned above. Hence this level is easy to win.
- **Level 2:** Here, I have limited the depth search to 10. Also, the evaluation function returns a score either from random_score or eval_score with a probability of 0.5. Hence this level is moderately difficult. There are good chances to make it a draw.
- **Level 3:** Here I have limited the depth search to 13 taking into consideration the time limit of 15 seconds by which computer has to make a move. The evaluation function returns eval_score from the formula mentioned above. Hence, it is highly difficult to win here.