Checkers User Interface Played with Artificial Intelligence

Ryker Frohock

May 6, 2019

Contents

1 Abstract **1**
1.1 The Game . . . 1
1.2 History . . . 1
1.3 Necessary Components . . . 1

2 The Approach **2**
2.1 Player vs. Player . . . 2
2.2 Player vs. Computer . . . 2
2.2.1 Easy AI . . . 3
2.2.2 Medium AI . . . 3
2.2.3 Hard AI . . . 3

3 Final Deliverable **4**
3.1 Player vs. Player . . . 4
3.2 AI vs. Player . . . 4
3.3 AI vs. AI . . . 5

4 Discussion **5**
4.1 Problems Encountered . . . 5
4.2 Future Work . . . 5

5 Citations **6**

1 Abstract

1.1 The Game

Checkers, or draughts (British English) is a two-player strategic board game involving diagonal moves of game pieces and mandatory captures by jumping over opponent pieces. While there are many versions of checkers, the most common is American checkers and Russian Draughts, both played on an 8x8 board. Other versions include international draughts and Canadian checkers, played on a 10x10 board and a 12x12 board, respectively. For the sake of consistency, the game will be referred to as checkers for this report.

Checkers is played by two opponents on opposite sides of the board, one player being able to move only the dark pieces and the other may only move the light pieces. Only the dark squares are used, and players make moves in alternating turns. A player performs a move by moving only their piece diagonally by an adjacent unoccupied square. If the adjacent diagonal is held by an opponent piece and the square immediately beyond it is empty, the piece may be captured and removed from the game by jumping over it. In most official settings, capturing a piece is mandatory. It is mostly universal that once a player is unable to make a move or runs out of pieces, the game is over and that player has lost.

1.2 History

The game of checkers has been played in similar variants for thousands of years. The earliest version that has been found is currently in the British Museum and dates back to ancient Egypt, and has been mentioned by Plato and Homer in their writings. The capture method was loosely derived from the Trojan War era, and can be found in some examples of ancient Roman games. The game developed further with the crowning method during the 13th century[1]. Since then, it has had tweaks to make it the game we know today, namely the size of the board and forced capture.

In recent history, the game of checkers has been involved in the arena of game artificial intelligence. In 2007, computer scientists at the University of Alberta developed the Chinook program far enough to make it unbeatable[2]. Unfortunately, the game I have created is far from that. Using brute force, the game was solved after two decades and will always end in a draw if played "perfectly" (i.e. no player makes a mistake)[3].

For the sake of computational complexity, checkers has been generalized and can be played on any N x N board. To determine whether a specific player has a winning strategy is the set of all decision problems that can be solved using a polynomial amount of space, meaning it is *PSPACE-hard*[4].

1.3 Necessary Components

The pieces used in checkers consist of two ranks for each team, Men and Kings. Both have specific uses and movement abilities.

Each player starts a standard game of checkers with 12 men. These pieces may only move forward diagonally, and are used to create strategic traps, blocks, and jumps during the game. The opening move generally belongs to black, and each piece can be moved forward one space at a time at the beginning of the game. Although men may be moved

forward, if a jump can be made then that piece must be captured according to section 1.1.

If a piece successfully makes it to the opposing players backmost row, then a piece that has been captured by the opposing player is placed on top of that piece and is elevated to king status. In standard checkers, when a piece has king status, it is allowed to move to any adjacent black space on the board forward or backward. It may also jump an opposing player's piece forward or backward. Like men, it may also make successive jumps in a single turn if the jump captures an enemy man or king. It should also be noted that the king piece also adheres to the rules of mandatory capture.

2 The Approach

2.1 Player vs. Player

In order to set up the board and the rules properly, I found it best to set up a standard game of checkers with an original option of player vs. player. The most crucial part of the game is having the pieces move. Getting the pieces to move depended on a data structure that contained an array of a grid class, where each grid object contained the button being clicked (which was necessary to move images around), a character "mark" being either *'b'*, *'r'*, *'-'* (unused square), or *' '* (empty square), and a boolean variable that indicates the king status. Pieces are able to be moved by clicking the piece that the player wants to move and then clicking the square where the player wants to move it to. After developing the ability to move pieces, validation was created by first checking whether or not an actual piece was clicked, then by checking whether or not the piece is the players piece simply by using a boolean turn variable. Once a proper piece has been clicked, it is highlighted, then the player is able to take the rest of their turn by clicking on an empty square. If the move is legal, the image of the piece is moved to the second square clicked, and the computer does the rest of the work setting up the turn for the next player. This repeats until the game is finished and the winner is announced. The PvP game also takes valid king moves.

The foundation of this game was difficult to develop due to the fact that it must check if the second square clicked is both adjacent according to how the original game defines adjacency and whether or not a piece is there. If a piece is there, it must be determined whether it is an opposing player's piece, or the current player's piece. If the player wants to perform a capture, it must be checked if the second piece clicked is an empty square, and the piece meaning to be jumped is an opposing player's mark. There must also be a separate logical path for a king piece to make its move, which is a combination of the current players forward moves and jumps, and the opposing players' forward moves and jumps tailored to the current players jump restrictions.

2.2 Player vs. Computer

Originally, I thought due to the time constraints, that the intended AI vs. AI could not be completed. Therefore, in order to get something working, the user of this program may choose difficulty levels of the computer to play. The three levels of AI are dependent upon different methods of choosing moves. The sections below go in depth as to how moves are chosen by the computer. In general, the player makes the first move, then

immediately goes into the computer's turn. The code path the computer takes allows the computer to scan the entire board for its legal moves in the same way it determines if the user chooses a legal move. It scans the board for all of the pieces that coincides with the turn it is taking and stores them in a list. If there is a move to be made, it stores the current grid object in a class called Move as the current piece and it stores the grid square that the piece is being moved to in the same move object, and if there is a jump, the opponents piece is stored as the adjacent square, and the square being jumped to is stored as such. If there is just a forward move (no jump) then the jump variable is set to null. The logic path for the king piece was created as a culmination of the player version of a king move plus the version of a normal AI move plus how the opponent would move if it was an AI.

Determining legal moves for the AI was extremely time consuming to code in itself because it has to check each piece's surroundings to find any and all legal forward move or jump, and the king even more so because it has to check backwards legal moves as well as its own forward legal moves. I had a numerous amounts of occurences where it would potentially check off the board, so the first, second, second-to-last, last row, and the rows in between had to be checked for legal moves without going off the board, same had to be done for the columns of the board.

2.2.1 Easy AI

Once all legal moves have been found and stored in a list, the list is returned to the calling function, where the computer determines which move to take. The easy AI portion chooses randomly from the list of moves and makes the move on the board that it chooses. Unfortunately, due to the rule of forced capture, this method had to be reworked to first check for jumps. Since this method of choosing moves chooses randomly, the chances of it choosing the best move randomizes the bell-curve. Ultimately, this section of the program was reworked as such, and the medium level was removed because it essentially did the same thing, aside from choosing randomly compared to choosing the first available.

2.2.2 Medium AI

The medium-level AI is intended to prioritize any move where there is a potential for double-jumps. There are a few logic errors in the code that causes another move to be made if there is a move found before the second jump of the piece that just moved. This can be easily reworked to set the piece that was just jumped to be the next selected piece, and the second jump to be the next jump in the move object. I digress, the medium-level AI will prioritize the first successive-jump, then the first jump, then the first move to be found. This will cause a new user to catch on to the pattern rather quickly, so this can be reworked by making new lists of the moves, then to check each list, and take the moves randomly. As stated in 2.2.1, this entire section was removed in the final submission as the easy version essentially does the same thing.

2.2.3 Hard AI

The hard-level AI employs basic heuristics for choosing moves 1-ply deep. In order to hold true to the forced-capture rule, the score for a potential jump will take precedence

over anything else. A move is scored on its ability not to result in an opponent capture, the pieces movement to the edge, a potential king-row move (reaches the opposite end-row), a potential king-row jump (takes ultimate precedence), and whether or not a jump will result in another same-turn successive jump. What I intended to do with this is to use a 3-ply alpha-beta pruned search tree to score the current board after the move, which is the same method used in the Reversi board game. The board has a potential to be scored upon where the current player's pieces lie in relation to the edges of the board, as well as their respective closeness to the opponents pieces. As glossed over in section 1.2, the use of brute force over multiple computers over 20-some years caused the game to become unbeatable. This is not the method used in this variant of the game. The intent of this program was to have a user load a separate executable program to take the current state of the board, judge each possible move based on the current state of the board, and let the computer choose the move based on the best possible move it determined using heuristic analysis.

Due to some last minute workings, all of these methods of play were abandoned.

3 Final Deliverable

Ultimately, I was able to create a program that almost exactly like the Reversi game. A user is able to calibrate his or her AI file using the C++ source code to play the game as an executable. The executable from the C++ code included in the checkers game file is only a base, and makes moves based on random selection. Since it is so closely related to the Reversi homework, a user is able to code the c++ source code file however they please. Since I was able to work out this game to operate within the confines of the project specifications, I decided to remove the game option windows form, along with the Vs. Computer mode and the PvP mode. When the user clicks the executable, it will immediately open up the game board, so there is no use for the "Easy", "Medium", and "Hard" level AI's, but the algorithms and heuristics used can certainly be coded in the C++ source file as such.

3.1 Player vs. Player

Two users are able to play Player vs. Player on the same computer if they leave both file spaces blank. It works in the exact same way as the original player vs. player works, nothing had to be changed other than the user(s) have to click the start button to begin the game.

3.2 AI vs. Player

A single player is able to play an AI program in this final development. The user can either play on the black side or the red side based on which box they put the file in. Because black goes first, if the user enters a file on the black side, the AI will take the first move, then pass the turn to the user. If the user enters a file on the red side, the AI will not make a move until the user makes a move.

3.3 AI vs. AI

The most significant development comes from the AI vs. AI mode. The user enters two of the executable files created from the C++ source code on both sides of the board. The executable will take the board passed to it from the C# executable, it will find all legal moves, put them in a list, and then pick randomly from that list of moves unless the list has a jump.

4 Discussion

4.1 Problems Encountered

As far as building the game, developing it was about as straightforward as it could possibly be. Build the foundation, instantiate the rules for play, validate the consistency of gameplay, and build how the computer plays based on the foundation. My biggest issue was the fact that creating a respectable AI is extremely in-depth. With the time that was given for the project, it would be impossible to create an unbeatable checkers AI, even with a full team of people. I feel it absolutely essential to mention that the one teammate I did have for this project, I had to kick off the project. This job included developing a clean user interface, a playable game, and the code necessary for a computer to play against another player as well as for a program to play against another program. A job that required multiple people dropped to a one-person job. There is definitely room for improvement on this project but it should be enough to make a convincing argument that all the necessary pieces are in place, and with more time this program has potential for growth.

4.2 Future Work

The intention of this program is for an AI to play against an AI, and originally I did not think it possible. Luckily I was able to do enough research and rework the program almost from the ground up using the original code I had. There was enough room to retool the UI to include the ability for a user to pit 2 AI's against each other in the same way the Reversi program did. There is also more than enough room to further develop the heuristics used to choose moves, as what I have coded is what amounts to a test file. Because I am very happy with how the UI looks, I would like to leave it alone for now and work out how the C++ code determines moves. Furthermore, I have put my code on my Github repository for others to work on or use.

5 Citations

[1] Masters, J. (n.d.). Draughts / Checkers and the Alquerque Family.
Retrieved from *https://www.tradgames.org.uk/games/Draughts.htm*

[2] Project. (n.d.).
Retrieved from *https://webdocs.cs.ualberta.ca/ chinook/project/*

[3] Schaeffer, J. (2006). "Samuels Checkers Player", *Encyclopedia of Cognitive Science*, doi:10.1002/0470018860.s00001

[4] Horssen, J. V. (2019). "Complexity of checkers and draughts on different board sizes", *ICGA Journal*, 40(4), 341-352. doi:10.3233/icg-190077