ECE469 - Checkers AI Report

Arvind Nagalingam

November 5, 2017

1 Introduction

For the game-playing AI project, I chose to implement an AI for Checkers. The program is written in Java, but the code uses very little Java specific functionality. Therefore, the code could be rewritten in other faster languages like C for better performance. I mainly chose Java to get more comfortable with the language.

The code for the program is contained in two files, checkers.java and board.java. The checkers.java code simply contains the "main" function of the program with game setup instructions like game mode, AI time limit, etc. Once the game setup is parsed from the user, a board object is created and a while loop is started which just calls the board.play() method while incrementing the turn counter. This main function could have been placed in the board.java file as well, but I kept it in a separate file for ease of development and to keep the board.java file as small as possible.

2 Code Explanation

The core functionality of the Checkers program lies in the board.java file. This file has multiple sections: creating the board object, interacting with the board, finding moves, determining the AI's moves, and outputting what happens on every turn of the game. In other software projects, I would have focused more on segmenting the functionality of this code into multiple class files. Though this would make the code maintainability and readability easier, I steered away from this solution since the goal was performance. Adding a more complicated class hierarchy on top of the project as is could slow down the AI's search speed, thus making the project worse on the performance side.

The primary performance boosts that my implementation has over others comes from the storage/manipulation of the board and the game tree. Rather than using an 8x8 dynamically allocated matrix of ints for storing a board, I compacted the representation to a single static array of 4 ints. In Java, each int is guaranteed to be 32 bits. Furthermore, each of the 32 possible squares on the board can take a value from 0-4, requiring 3 bits each. So the board can ideally be referenced in 3 ints = 96 bits. I included 4 ints for ease of implementation, with each int corresponding to 2 rows of the checkers board. Obviously this added a necessity for more complicated viewing and interacting with the board, but I deemed the memory savings of a board to outweigh these simple operations like bit shifts, ands, and adds/subtracts. I later heard of an idea called "bit boards" from other students which may be slightly more efficient, but I did not think that the performance enhancement warranted rewriting this base code under the time constraints.

The other major performance boost was in the representation of the game tree for the minimax search. Since the minimax algorithm essentially uses a post-order depth-first search of the game tree, a stack would be necessary to keep track of the nodes. Rather than dynamically allocating memory for this stack, I created a static array at instantiation of the board which I use as a "validMoves" stack. With proper manipulation of the "numValidMoves" stack pointer and ensuring that moves are "deleted" by assigning zeros to all elements, a more efficient stack was created. Each valid move was essentially a ply of the game tree. When the valid move was applied to the game board, the game board became one of its children. And when the valid move was later reverted, the game board returned to the parent state. This search therefore uses no dynamic memory other than the variable instantiations in each call of board.alphaBeta(). To allow for this, a smaller static sqVals stack was needed the kept track of the critical square values from the parent board state at each depth level. The efficiency of this system could have been further improved by compacting

the validMoves array from 11 to 3 ints, but again I did not know if the performance enhancement would have been worth the rewriting of the moves code.

The final place where my code is unique is in the heuristic function. In any non-terminal state, the heuristic function returns an integer based on the number of pieces and kings of each player and a piece advantage bonus (for encouraging trades when in the lead). In the early and middle game, there are square bonuses for advancing pieces, maintaining the back row, and guarding other pieces. In the late game, which is defined as 3 or more kings on the board, a player with a piece advantage is incentivized to get closer to enemy pieces based on a Manhattan distance penalty to the closest enemy piece. Furthermore, a player in the disadvantage gets a square bonus for occupying either double corner. These end game provisions allow for dealing with the two kings to one king double corner situation.

3 Results

Overall, my AI plays extremely well. In its final state, I have yet to beat it. Though I am not an extraordinary checkers player, I do think that the program is so good because of its deep searches and a well-designed heuristic. The main problem for the program is in finishing out a two kings to one king double corner situation. The program wins once it forces a board similar to the one below.

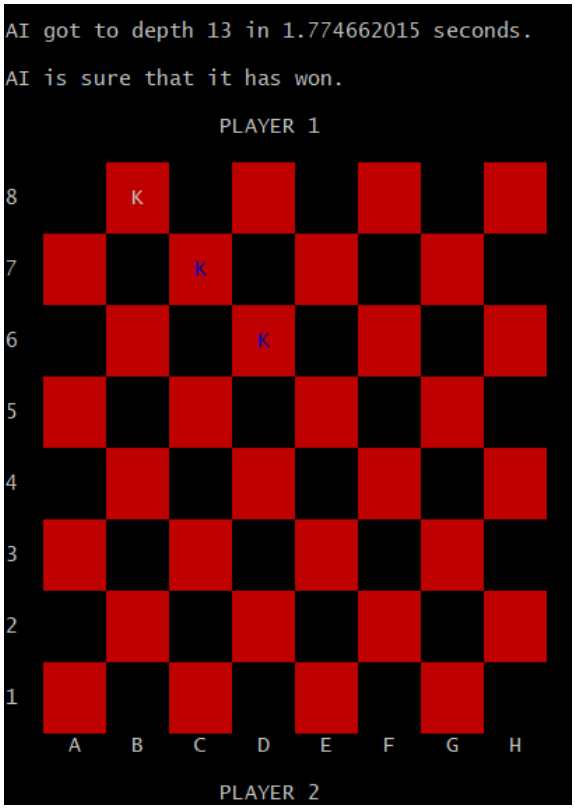

Figure 1: The AI's winning board state for a two king to one king double corner situation

Though the program closes in on this position thanks to the Manhattan distance penalty, it may sometimes not get a deep enough search to see this position is a guaranteed win. Since many moves could result in the same heuristic at this point in the game with a deep (but not deep enough) search, the AI sometimes makes random moves away from the double corner. However, I have seen that even in the 3 second AI timer situation, if the AI moves to some positions like both kings near the boundaries of the board, enough pruning can occur to find the pictured winning solution at around depth 15. If the AI keeps playing, eventually it will win, but it may take a while for it to randomly step to a board state where it can see that win.