AI-Checkers Game Instructions and Writeup

Compile Instructions: The program consists of 3 source files (main.cpp, initialize.cpp, backend.cpp), 2 header files (initialize.h, backend.h), and a makefile. The makefile can be used to compile an executable (checkers.exe).

Implementation and features:

The structure of the game consists of a 2d int array [8][8] as the board and pieces as objects of a custom “piece” class. The “0” on the board represents an empty or non-plyable square, “1” represents team 1 pawn, “2” represents team 2 pawn, “3” represents team 1 king, and “4” represents team 2 king. Team 1 is the black team, which also represents max player, and team 2 is the red team, which represents min player.

Each piece object has the attributes of int currentPos[2], which keeps track of the position of the piece on the board[row][column], int team, which keeps track of the team (1 or 2) that the piece belong to, type, which is the same value as the piece digit on the board (1, 2, 3, or 4), and int ID, which links the piece to its position index on the team vector for constant access- each team of pieces is contained in a vector <class piece>. When a board of standard setup is initialized or a game from a file is loaded, it is initially parsed, and the piece digits on the board (1,2,3, or 4) are used to initialize unique piece objects, which are loaded into their respective team vectors. Structuring pieces as objects and storing them in vectors allows constant access (especially through ID attribute – discussed more later) to each piece on the board for applications such as checking for legal moves and changing the digit values on the board (deleting a piece (ie ‘1’ -> ‘0’, moving piece, or updating pawn to piece), which is part of applying a move.

For legal moves, each legal move is an object of the class “move.” The move class has attributes of “piece curPiece”, which is the piece that the move is centered around, and vector <int> “opp\_IDs”, which collects the IDs of captured opponent pieces. For obtaining legal moves, the parameters passed in include the board, team vectors, and int team turn (1 or 2). Prior to collecting moves, a vector <class move> was declared. For each piece in the team vector of respective turn, possible directions that the piece can travel based on piece type and position are initially found (to minimize computations) – for example, a black pawn on the leftmost column can only travel to the upper right square. Then, for each valid direction, if there is an open square that the piece can move into, a move with the piece attribute whose currentPos is equal to the position of the open square is initialized and added to the move vector. If there is an opponent that could be taken, a recursive process takes place (creates copies of the board and team vectors) to find all of the possible legal moves and all the moves get pushed into the moves vector. If an opponent is captured, the opponent piece’s ID gets pushed into the move’s “opp\_IDs” vector. After all moves have been added, a loop is used to check if forced moves exist. If so, the loop is broken, and all moves with empty opp\_IDs vector are removed.

When a move is applied, the ID of the piece that the move is centered around is first used. The ID is used to access the specific piece in constant time within the team vector. Then, the position of the piece on the board is set to zero, then the piece at the ID index is replaced with the move’s piece. After, the digit at the position of the board correlating to the new position of the piece is replaced with the digit of the position type. If there are opponent pieces that were captured as part of the move, the vector of opponent IDs is first sorted. Then from the highest to lowest value, each opponent piece is accessed in the opponent team vector through its ID, and the digit on the board at the opponent piece’s location is equated to ‘0’. After, the piece is deleted from its team vector. After all the deletions, the remaining pieces’ IDs are updated to match its index with respect to its team vector.

For minimax and iterative deepening, the board and team vectors were copied for each move and the time at which the search should end was determined to be at 99.5% of the input time limit. As for the heuristic/evaluation, points were given according to each position type, protecting the end row for each team, and being backed up by another piece of the same team.