Ethan Yang and Jaeden Hufnagle eyang13@u.rochester.edu & jhufnagl@u.rochester.edu
CSC 242: Intro to AI **Project 1: Game Playing**

How to Run:

==========

Things to Note:

- The AI algorithms are in the Game class, **Lines 223-389**.
- When inputting a move, please use this format: **A2-B1** (5 characters exactly). Not case-sensitive.
- We also assumed the user wouldn't try to make moves off the board as in normal checkers it would break the game so it wouldn't be any different in our game
- We made it so that you can use all 4 Search strategies in both the 4x4 and 8x8 board, but for the 8x8, min-max and for alpha-beta pruning would not be very effective at the beginning because we have the maximum depth of 50, which is not enough for there to be a terminal nodes in the tree. Thus, the AI will simply choose the first action it can. As the game progresses, however, it will reach terminal states and the minimax and alpha-beta pruning will begin to work as expected.
- We also assumed the user wouldn't try to move on top of another piece that already exists
- We decide a draw for a 4x4 after 10 moves, and for the 8x8, after 50 moves.

Needed for Compile:

- To actually compile the program you may use the command line going to the directory then calling *javac Main.c* to compile the program then use *java Main* to actually execute the program.

Explanation of Code:

==========

Files needed to run the program:

1. Main.java- Simply where game is initialzied
2. Game.java - where the game logic and AI functions are
3. StateNode.java - where the Nodes of the search tree are created and manipulated
4. State.java - where the states of the game are created and manipulated
5. Board.java- handles the orignial setup of the game as well as actually changing values of a game
6. Piece.java- Creates pieces with boolean if they're white or not and also handles their positons

Description of Functions:

Main.java{

main()- Simply creates the initial Game instance and calls the game loop function

}

Game.java{

Game()- Constructor for game and initially calls setUpGame to get the game started

setUpGame()- Initially prompts the user for which game they desire then from there prompts the user to pick their choice of AI and if they pick H-Minimax they must also choose how deep they wish it to go to. Sets which AI is chosen (max depth too if H-Minimax), and also handles which color player and subsequentially AI will be.

GameLoop()- Reads input from the user and makes the move if it's a valid move also tasks the AI with tree building and takes input as well to allow moves. Keeps track of moves too for sake of draws (10 moves for 4x4 and 50 for 8x8 as instructed). Prints move made and the board after the move as well (General game things).

AIChoice()- Calls the appropriate algorithm based on what kind of AI the user wants to face and actually handles the random AI in the method by randomly selecting a valid child state.

MiniMax()- Calls Max of pushed utility values if white because our utility function gives values in terms of white so if black calls the minimum as it wishes to minimize whites' outcomes. Also is able to return a node by figuring out which node gave the root the optimal value and saving it.

Max()- Returns utility value if terminal loops through the children of the current node then call Min and get the max of the mins (setting the current best value to return to its prior call)

Min()- Returns utility value if terminal loops through the children of the current node then call Max and get the min of the maxs (setting the current best value to return to its prior call)

AlphaBeta()- Sets the initial Alpha to negative infinity and beta to positive works similar to MiniMax of its calls based on which color it is

Alpha_Max()- Works the same as Max() but if the v is greater than beta it will return v to prune the branches and gets the max of alpha and v

Alpha_Min()- Works the same as Min() but if the v is less than alpha it will also return v to prune and will get the min of beta and v

H_MiniMax()- Works identically to MiniMax() but instead of getting the general utility value at terminal states it gets the heuristic value of the set depth states (Not gonna include H_Min and H_Max because they're identical to Min and Max).

}

Board.java{

Board()- Initializes a 2D array with the size given and also saves the size so it can be called at any time.

makeMove()- Builds the string to make a move

setupBoard()- Creates the initial board for the 4x4 and the 8x8 based on the size of the board constructed meaning it adds the pieces to their proper places

printBoard()- loops through the board printing the current position of the pieces allowing the user to keep track of the board

}

NodeStates.java{

NodeStates()- Constructor for tree nodes there are two versions as the one that takes max depth handles H-Minimax giving it the proper cutoff at user discretion. Otherwise sets max depth for trees at arbitrary sizes to handle draws properly and prevent infinite trees.

FindVal()- Returns zero if max depth reached meaning draw otherwise checks for white-black to show how many white pieces are left this acts as our heuristic function.

Utility()- If max depth then returns 0 because it's a draw otherwise return 1 for white winning and -1 for black as this also favors white.

findChildren()-Loops through possible results of a state and then finds possible states to create new children for the children and continues this process until max depth or no valid moves.

NodeinChildren()- Checks to see if it's a child node

printNodes()- prints out all the nodes (mostly for debugging)

}

Piece.java{

Piece()- Constructor for piece holding if it's white, king, and it's position

getPosition()-returns position of piece

getSymbol()-returns the symbol

getTeam()-returns what color it is

CopyPiece()- copies the given piece

}

State.java{

State()- Constructor that creates the different states of the game

translateX()-Moves the character to the proper value for it to be read into the array

translateY()-Simply shifts the value for y down 1

ifopponent()- For checking if the diagonal piece is an opponent or the same team to prevent invalid moves.

makeMove()- Checks to see if the inputted move is in possible states then makes the move if it's valid updating the current state to be the new one and also checks for possible kings at the end of the move.

copyBoard()- Copies current board to another array for saving and simulating

inResult()-checks if a state is within the state's resulting states list

printResult()-prints resulting states.

}