CSC 242 Project 1
Maryfrances Umeora, Kelley Foley, Kharissa King
mumeora, kfoley6, kking33

## Program Files Used

- main.java
- Board.java
- Piece.java
- Player.java
- Square.java
- TreeNode.java
- checkersState.java
- checkersGame.java
- checkersMove.java

## How To Run Our Project

We wrote our project in Java, so it should be able to run on any Java compiler and/or terminal. To compile the files from the terminal, first navigate to the folder where they are saved. Then, use the command “javac *.java”. To run the code, type “java main”.

## Set Up

As per the teacher’s instruction, we started by designing our data structures.
↓
### State
We represent our state object with the class `checkersState.java`. A state in a checkers game consists of the current configuration of the board, the location of each player’s pieces which helps additionally with locating valid moves, the current player (represented with the variable `whosUp`), and the number of moves that have been made so far in the game.
Our State class has multiple different constructors for different situations. It also has various methods to print the board, make copies of the board, etc.
↓
### Action
An action is represented in our class `checkersMove.java`. A move has a string moveDescription ie `"C4-B5"` and an integer value to simulate a tree. Moves are comparable. Jumps are represented with an “x” in place of the dash “-”, ie `"C4xA6"`. Please ensure that you use capital letters for the board positions when entering moves.
↓

## Transition Model

As a transition model is just the *actions* (that checks applicability of actions given states) and *results* (that returns state that results from making action in certain state) functions, our transition model is represented in the same class that we run our game in: `checkersGame.java`. We have

- `testValidMove(String move, checkersState s)` and
- `result(checkersState w, checkersMove m)`

to represent these functions.

↓

## Cost and Utility Function

The perceived cost for each action is constant and represented best in our utility function which can also be found in `checkersGame.java`. It returns -20 or 20 as the utility for the given player. It returns 0 if it ends in a draw.

We determined draws using Ferguson's suggestion, which is after 10 moves for a 4x4 and after 50 moves for an 8x8.

## Other classes we have include...

- Square: which represents a physical square on a game board. A square may be occupied or empty. A square also stores the color and status of the piece occupying it (whether it is a king or not).
- Board: which is a two-dimensional array of Square objects. A board has a size.
- Piece: which represents a physical game piece. A piece has a color, a status, a row and a column.
- Player: which represents a player, be it human or MINIMAX.

**Note:** If you type r when it is the human's turn to play, a random legal move will be generated. This makes it easy to test our code :)

## **How Our Code WORKS**

Everything starts in the class `main.java` where we prompt the player for information about what type of game they want, what color they want to play, etc. After they have made all these choices and we configure those choices to our game, we begin a while loop

```
while(!game.isTerminal(game.currentState))
```

which will alternate the player's turns with the computer's until a terminal state is reached.

This takes us to our game engine class, known as `checkersGame.java`.
Associated with a game is a board size, a current state, a count of moves taken, and the max and min players.

If the current player is the user (human player), a check is made to see if they typed "r" instead of a standard move. If they did, a random move is played from a list of the player's valid moves in the current state. Otherwise, our `result()` method is called, and it returns the resulting state produced by the user's move (action) if it was valid. If the move is invalid, however, the user is prompted to make another move.

If the current player is an AI agent (Minimax, H-Minimax, or Alpha-Beta Minimax), the appropriate algorithm is run, the valid action(s) is/are found, and a move is made. The resulting state of the chosen action is returned. Of course, the move count is incremented for each action taken.

So how do we check if a move is valid? Well, if it is not in the list returned by getValidMoves(), then it is not valid. How getValidMoves works is that we check all directions to see if the individual moves are valid. If they are, we add them to a linked list which we return.

To implement minimax, we introduced an additional data structure TreeNode that consists of a checkersState, its parent node, and the move that got us to this TreeNode's state along with its minimaxValue, alpha, and beta. Given the current state of the game, we begin minimax by creating a TreeNode with its state property set to be a copy of that state (so as not to manipulate the actual state of the game).

For normal minimax `minimaxSearch()` with helper functions `maxValue()` and `minValue()`, we adapted the pseudocode in the AIMA textbook. We search the entire gameTree by generating the successor states (children nodes) of whatever node we are expanding, starting with the currentState TreeNode as the root. When we get to a terminal node, we apply our utility function, then work our way back up the tree to assign minimaxValues to all nodes. The best move– the one that gets us from the root TreeNode to its child with the highest minimax value– is then selected

We also have minimax with alpha-beta search `alphabetaSearch()` with corresponding helper functions. This acts essentially the same as minimax, but now we keep track of upper & lower bounds (alpha and beta) for the minimaxValue of each TreeNode, so as to allow for pruning of subtrees that won't affect the outcome of the game. We adapted the alpha beta search pseudocode featured in the AIMA textbook.

For a heuristic, we evaluated how many pieces of each color are on the board and whether they are kings. Our heuristic minimax function `hminimaxSearch()` and corresponding helper functions already implement alpha-beta pruning. It is the same as the `alphabetaSearch()` function, however once **we reach a depth of 6** in the tree, we cut off the search rather than going all the way down to terminal states. Then, we apply our heuristic to these TreeNodes at the cutoff level with our `evaluation` function (in `checkersGame` class) rather than using the utility function.

Note that our heuristic minimax does not allow the player to choose the depth at which to cutoff.

~***~

THANK YOU FOR READING OUR README :)

*(if you read it at all)*

~***~