Computer Science CS136 (Fall 2021)
Duane Bailey & Aaron Williams
Laboratory 7
Playing Gardner's Hex-a-Pawn

Objective. To use trees to develop a game-playing strategy.

Discussion. In this lab we will write a program to play the game, Hex-a-Pawn. This game was developed in the early sixties by Martin Gardner. Three white and three black pawns are placed on a 3×3 chess board. On alternate moves they may either move forward one square or capture an opponent, diagonally, ahead. The game ends when a pawn is promoted to the opposite row, or if a player loses all his pieces, or if no legal move is possible.

In his article in the March 1962 *Scientific American*,[1] Gardner discussed a method for teaching a computer to play this simple game using a relatively small number of training matches. The process involved keeping track of the different states of the board and the potential for success (a win) from each board state. When a move led directly to a loss, the computer forgot the move, thereby causing it to avoid that particular loss in the future. This *pruning* of moves could, of course, eventually cause an intermediate state to lead to a resignation, in which case the computer would be forced to prune an intermediate move relatively high in the original tree.

Gardner's original "computer" was constructed from match boxes—one per board state—containing colored beads. Each bead corresponded to a potential move, and the pruning involved disposing of the *last* bead played. In a modern system, we might represent each board state by a node of a tree stored in a computer. The children of each node would correspond to states entered by each of the possible next moves.

This Week's Tasks. Our hope is to develop a toolbox of classes that will help us explore Hex-a-Pawn more deeply. Here are the expectations for this week's investigation:

1. Clone your `lab7` repository:

   ```
   cd ~/cs136
   git clone https://evolene.cs.williams.edu/cs136-labs/22xyz3/lab7.git
   ```

   replacing 22xyz3 with your CS username. This will create the directory `~/cs136/lab7`.

2. Available for your use are several Java classes:

   `HexBoard`. This class describes the state of a board. The default constructor builds the 3×3 starting `HexBoard`. You can ask a board (with its `moves(color)` method) to return a `Vector` of the `HexMoves` that are possible for a particular color (`HexBoard.WHITE` or `HexBoard.BLACK`) from the position. The `win(color)` method allows you to ask a `HexBoard` if the current position is a win for a particular color. A static utility method, `HexBoard.opponent(color)`, takes a color and returns the opposite color.

   The `main` method of this class allows a human to play Hex-a-Pawn against a computer that moves randomly. It is a demonstration of how `HexBoards` are manipulated and printed.

[1] Gardner, Martin, *Mathematical Games*, Scientific American, March 1962, pp. 138–144.

`HexMove`. This class describes the movement of a pawn. The result of `HexBoard.moves` is a `Vector` of `HexMove`. Given a `HexBoard` and a `HexMove` one can construct the resulting `HexBoard` using a `HexBoard` constructor. These two classes—`HexBoard` and `HexMove`—are vital in exploring the state-space of the Hex-a-Pawn game.

`GameTree`. This is one of the classes *you* will construct. The `GameTree` nodes will form a large tree of `HexBoard` states, related by player moves. At the root is the starting position, ready for `WHITE` to move. The next level of the tree describes `HexBoard` positions that are the result of a `WHITE` move, ready for `BLACK` to move. The 3×3 game leads to a tree with 252 nodes. We expect that players will traverse a single tree recursively and, if they wish, *prune* the tree to learn from losses.

`Player`. The `Player` interface describes the methods that must be provided by agents that play the game. Every `Player` must have a name and color, accessible through `getName()` and `getColor()`, respectively. In addition, they must support a `play(node,opponent)` method takes a `GameTree` node and an opposing `Player`. This method plays the game by traversing one level of the `GameTree`—if it can—and checking for a win. If the player's move does not lead to a win, it passes control of the game to its opponent. The result of the `play` method is the `Player` who ultimately wins the game.

Read these class files carefully. Please do not modify the classes `HexBoard`, `HexMove`, or `Player`.

3. Compile `HexBoard.java` and run it as a program. Play a few games against the computer. You may wish to modify the size of the board. Very little is known about the games larger than 3×3.

4. Construct a tree of Hex-a-Pawn board positions. Each node of the tree is represented by a `GameTree` object. The structure of the class is of your own design.

   Here are some observations about this class:

   - The `GameTree` class should have a constructor that, given a `HexBoard` and a color (a `char`, `HexBoard.WHITE` or `HexBoard.BLACK`), recursively generates the entire tree of all boards reachable from the specified board position during normal game play.
   - The `GameTree` node that encapsulates the starting `HexBoard` position, ready for `WHITE` to move, will be the root of the tree that describes this game's state-space.
   - The children of the root `GameTree` node describe the `GameTree`s that result from different move choices by `WHITE`. The positions described by these `GameTree` nodes are ready for `BLACK` to move.
   - Alternate levels of the tree represent boards that are considered by alternate players.
   - The leaves of the tree are positions where a player loses because they cannot move.
   - Typically, a program will construct a single `GameTree` and then traverse the tree many times.
   - Keep in mind that different `Player`s will traverse the tree. Think about the information that a `Player` might need and make sure to include methods that will allow them to access (and possibly modify) the `private` `GameTree` instance variables.

5. We now begin to construct different kinds of Players that play the game of Hex-a-Pawn by traversing the GameTree. These classes may interact in pairs to play one or more games.

   The first Player class is called HumanPlayer. This player prints the board and then, if it hasn't already lost (*i.e.*, if there moves that seem possible), it presents the moves and allows a human to select a move using input from a Scanner. (It may be instructive to look at the human side of the game played in the main method of HexBoard.) If the move does not lead to a win for the human, the play is then handed off to the opponent by calling the play method recursively. Remember: the play method should return the Player who ultimately won the game.

   Use the main method of the HumanPlayer class to allow two human players to explore the GameTree as they play a game.

6. The second player, RandPlayer, should traverse the GameTree by selecting random moves. Make sure you check for a loss before attempting a move. At this point it might be interesting to pit two RandPlayers against each other to play, say, 1000 games. Who would you expect to win most of those games?

   If you have made it this far, you have earned 9 points. The remaining portion of the lab, however, investigates the most interesting experiments posed by Gardner.

7. The third player, called CompPlayer, plays as intelligently as possible. When it loses, as the recursion unwinds, it prunes *only its most recent move* from the game tree. In future play, this small modification will cause it to consider other, possibly more rewarding moves. Of course it may be possible that further pruning causes the CompPlayer to resign very early, with knowledge that there is no win ahead.

   In the main method, pit two CompPlayers against each other. Each game leads to a loss and possible pruning of the tree. Very quickly the advantage of playing WHITE or BLACK will become obvious. In a comment, document the results of your investigation.

8. If you've made it this far, you have the potential to earn all 10 points. Review your code, make sure it is well designed and commented. If you ran any experiments that allowed you to draw conclusions, include those as part of the documentation.

Submitting Your Work. To get credit for this week's lab make sure that you've added, committed, and pushed your version of GameTree.java, the various players, and a signed honorcode.txt. As usual, remember that git status and git push will help to verify that everything is up-to-date.

⋆