Lab 6

Due 4 March

Handout 6
CSC 062: Spring, 2007
22 February

Hex-a-Pawn

This week's lab uses trees to play a small chess-like game. You will build a tree representing all possible states that the game board can be in. You will then implement several different players that consult this tree to make moves as they play Hex-a-Pawn. The players include: a human player that asks the user for moves to make; a random player that picks possible moves at random; and a computer player that improves its strategy by learning from past mistakes. In the end, you will be able to run the different players against each other.

Hex-a-Pawn was developed in the early sixties by Martin Gardner. Three white and three black pawns are placed on a 3×3 chessboard. On alternate moves they may be either moved forward one square or they may capture an opponent on the diagonal. The game ends when a pawn is promoted to the opposite rank, or if a player loses all of his pieces, or if no legal move is possible.

In an article in the March, 1962, *Scientific American*, 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, cause an intermediate state to lead indirectly to a loss, in which case the computer would be forced to prune out an intermediate move.

Gardner's original "computer" was constructed from matchboxes that contained colored beads. Each bead corresponded to a potential move, and the pruning involved disposing of the last bead played. In a modern system, we can use nodes of a tree stored in a computer to maintain the necessary information about each board state. The degree of each node is determined by the number of possible moves.

During the course of this project you are to:

1. Construct a tree, `GameTree`, of Hex-a-Pawn board positions. The structure of the class is up to you, but it is likely to be similar to the `LinkedTree` implementation. I suggest defining a subclass of `LinkedTree<HexBoard>` that contains the extra methods you will need to build the game tree and to manipulate it. Alternatively, you may find it handy to have a similar implementation to `LinkedTree`, but using an `ArrayList` to represent the children. If you do this, then you will have to start from scratch in building `GameTree`, but you need only write the methods you actually need.

2. Construct two classes of `Player`s that play the game of Hex-a-Pawn. I will provide a third for your use. These three classes may interact in pairs to play a series of games.

The start-up folder for this lab contains four classes:

HexBoard This class describes the state of a board. The default board is the 3×3 starting position. You can ask a board to print itself out (`toString`) or to return the `HexMoves` (moves) that are possible from this position. You can also ask a `HexBoard` if the current position is a win for a particular color – `HexBoard.WHITE` or `HexBoard.BLACK`. A static utility method, `opponent`, takes a color and returns the opposite color. The `main` method of this class demonstrates how `HexBoard`s are manipulated.

HexMove This class describes a valid move. The components of the `ArrayList` returned from the `HexBoard.moves` contains objects of type `HexMove`. Given a `HexBoard` and a `HexMove` one can construct the resulting `HexBoard` using a `HexBoard` constructor.

Player When one is interested in constructing players that play Hex-a-Pawn, the `Player` interface describes the form of the `play` method that must be provided. The `play` method takes a `GameTree`, a `TreePosition<HexBoard>`, and a `Player`. It checks for a loss, plays the game according to the `GameTree`, and then turns control over to the opposing player.

HumanPlayer This is a player that gets input from the console. It should provide a good sense of the structure needed for your other two players, though they will make moves randomly rather than getting them from the console.

Read these class files carefully. You should not expect to modify them.

There are many ways to approach Hex-a-Pawn. One series of steps might be the following:

1. 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.

2. Implement a `GameTree` class. It should have a constructor that, given a `HexBoard` and a color (really, a char: `HexBoard.WHITE` or `HexBoard.BLACK`), generates the tree of all boards reachable from the specified board position during normal game play. Alternate levels of the tree represent boards that are considered by alternate players. Leaves are winning positions for the player at hand. The references to other nodes are suggested by the individual moves returned from the `moves` method. A complete game tree for 3×3 boards has around 370 nodes.

   I strongly suggest that your `GameTree` class include the following methods (I use the first in the code for `HumanPlayer`):

   ```
   public TreePosition<HexBoard> child(TreePosition<HexBoard> node, int num);

   public void removeChild(TreePosition<HexBoard> node, int num);
   ```

   The first of these returns the num*th* child of the given node, while the second removes the num*th* child of a node. This will be needed for the computer player class.

3. Examine the class `HumanPlayer`. If it hasn't already lost (i.e., if the opponent hasn't won), this player prints the board, presents the moves, and allows a human (through a `Scanner`) to select a move. The play is then handed off to the opponent.

4. The second player, `RandPlayer`, should play randomly. Make sure you check for a loss before attempting a move.

5. The third player, `CompPlayer`, should attempt to modify its game tree, as described above, to remove losing moves.

Clearly `Players` should be able to play against each other in any combination.

For extra credit, create a nice graphic user interface (including a window with a picture of the board, a natural way to select moves, etc.) to make it more pleasant to play the game.

Using Scanner

The `Scanner` class is part of the standard Java libraries and provides simplified I/O from the keyboard (and elsewhere if desired). To use it, you must import `java.util.Scanner`, and then proceed as follows:

```
Scanner r = new Scanner(System.in);
...
int num = r.nextInt();   // gets next int typed by the user
```

More information is available in the javadoc for `Scanner`, but this is all you should need for this program.

Collaboration

This is a fairly complicated program that will be a challenge to debug. I encourage you to use `JUnit` to debug individual methods, and to work together with a partner and use "pair programming" in writing this program. If you do so, you must follow the rules from the last assignment.

What to hand in

Your final program will be due on Friday, March 9, at 5 p.m. As usual, place all of your code in a folder and drop it off in the CS 062 dropbox folder. Please be aware that I will be leaving town right after class on Wednesday, March 7, so will not be able to help you on your program after that time.

Acknowledgment This assignment is taken (with permission) from Duane Bailey's "Java Structures" text.