Assignment 3.4: Creating a Tic Tac Toe AI using Trees

Due: 8am, March 3, 2009

Directions

The problems for this assignment are described below. You will submit this assignment by emailing it do Mr. Wulsin. The file will be named “Assignment 3.4-LastName.zip”, where [LastName] is obviously your last name. All of your submitted assignments will be named in this fashion. The Zip file will contain a folder of the same name (so, “Assignment 3.4-LastName”), which will contain a folder for each of the subsequent problems. Each of these subfolders will have the name of its particular problem. Each of these problem subfolders will contain all of the necessary files for that particular problem. It’s very important that you follow all of these naming conventions exactly as specified.

Problems

1) TicTacToeAI

In this assignment, you will create an artificial intelligence (AI) that plays tic tac toe. Almost all of the background code required to run the game is provided to you. The game-running code is set up for you such that almost any two-player game can be run with it. Much of the background code is in the form of interfaces and abstract classes whose specific tic tac toe implementations you will provide. Setting up the code in this way makes programming future games (and their AIs) much easier. Our final project after the AP exam will be to create Othello AIs that will battle in a bracket-style tournament. Instead of having to re-code all of the gameplay, we’ll be able to use almost exactly the same template as we are here.

Given that you will receive all of the game-running template, you will simply need to create the tic tac toe specific classes. First, though, it would be helpful to review the essentials files you will work with.

GameRunner.java This class handles the basics of a two person gameplay, including displaying the game every turns and prompting each player for his move. It is generalized, and specific classes for a particular game are passed in through the constructor.

GameStateI.java This interface defines methods for a specific game state like displaying the game, printing the score, getting a list of valid moves for a given player, processing a specific move, and others. Implementations of this interface are used by the `GameRunner` class.

GameNodeI.java This interface, which extends `GameStateI`, defines methods used for a node in a game tree like those involved in generating the children for each valid move from the game state represented by that node and getting the heuristic value of that node (and its children).

Player.java This abstract class is a wrapper for any type of player, human or computer. Its most important (abstract) method gets a play from the player given a certain game state.

User.java This class extends `Player` and most importantly provides the prompting for the human user to enter a move.

AIBot.java This class extends `Player` and controls the AI calculation of what move to make by constructing a game tree.

HeurWeights.java This class is a wrapper for heuristic weights for an AI's heuristic evaluation of a game state. `HeurWeights` is meant to be inherited by a game-specific class with weights for various aspects of the game to consider.

TicTacToeRunner.java This class is the entry point for our program. It initializes the two players as well as a `GameRunner` object and starts the game.

TicTacToeState.java This class is the game-specific implementation of the `GameStateI` interface. You will implement the required methods.

TicTacToeNode.java This class both extends `TicTacToeState` and implements `GameNodeI`. You will implement the required methods.

A few conventions

a) We will identify each player by a `playerCode`, which—for a two player game—will be -1 (X) and 1 (O). The negative/positive number makes it easy to switch back and forth between players by multiplying the current player by -1.

b) We will represent the tic tac toe board (or crosshatch) as a 3x3, 2D `int` array.

c) We will represent a move as an integer. Although the user will enter a move like "a1" for the top, left-hand corner spot, we will represent that as a move of 0 (zero). The move codes are as follows:

```
     1   2   3

a    1 | 2 | 3
    -----------
b    4 | 5 | 6
    -----------
c    7 | 8 | 9
```

One can calculate the row and column indices from the move number by the following equations:

```
row = move / 3;

column = move % 3 - 1;
```

This relationship is quite usefull given that we represent the tic tac toe board as a 3x3 `int` array.

d) How you display the game is up to you, but it should look something like this

```
     1   2   3

a    X | O | O
    -----------
b      | X |
    -----------
c      | O | X
```

with letters representing each row and numbers each column, to make it easier for the user to enter a move like b2 instead of having to calculate in his head what the `int` representation of that move would be. You will make the methods `moveToInt(String move)` and `moveToString(int move)` for converting between `String` ("b2") and `int` (5) representations of a move.

How to approach this assignment

Although you're given much of the code necessary to run the game, you still have a lot of code to generate. I'd start working on the `TicTacToeState` class and get those methods working. Test them in a tester (you may want to comment out the lines in **TicTacToeRunner.java** or simply create a separate tester file), make sure they work as expected, and then move on the `TicTacToeNode` class, which has fewer but more complicated methods (that's where the real "thinking" of the AI occurs).

A helpful hint

One tricky part of generating the tree is creating the new `int` arrays from a given move. Let's say you want to create a child board from a the existing board and add a move to it. For example,

```
// board is of type int[][]
int[][] childBoard = board;
// if you wanted to place a -1 (or X) at position r,c
childBoard[r][c] = -1;
```

would actually just create a new `int` array that points to the same memory as `board`. After this code is executed, `board` now also has a `-1` at position r,c. This pointing to the same memory is not conducive for making a child node *different* from the parent. You must make a completely separate `int` array, like

```
int[][] childBoard = new int[3][3];
```

and then go through every cell (which have type `int`) and equating `childBoard[r][c] = board[r][c];`, since `int`s are primitive data types and thus assigning creates an actual copy of the value in the new memory address rather than just a pointer to the same memory address.

This copying process is called *cloning* because it creates a complete copy of some non-primitive data type but in a different memory address, so when a clone is modified, it doesn't also modify the original. It might be useful to create a method like `cloneBoard(int[][] board)` that returns a complete clone of an array. See me if you have questions or need further clarification.