Final Project: Othello AI Bots

Due: 8am, May 26, 2008

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 "FinalProject-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, "FinalProject-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) Othello

You will create an AI to play Othello for your final project. Othello (sometimes known as Reversi) is a fairly simple game to learn but can become quite sophisticated in higher levels of play. For the rules and some basic strategy, see Wikipedia's entry for Reversi. I highly suggest that you play a number of games to get an understanding of the game before you try to implement any real strategy.

Tournament Gameplay

The tournament will be a complete round-robin, in which every AI plays every other AI. At the end of the round robin, the AI with the best record wins. In the event of a tie, the two AIs will play a tiebreaker game. AIs will have 30 seconds to make each move. If they exceed the 30 seconds, they will be disqualified. Complete transcripts of every game will be posted to the website.

Class Structure

You will create four new classes, `LastnameBot`, `LastnameOthelloState`, `LastnameOthelloNode`, and `LastnameHeurWeights`, where (obviously) `Lastname` stands for your last name.
A summary of all of the classes in this game is given below.
The classes you must create are:

`LastnameOthelloState.java` This class inherits from the abstract class `OthelloState.java`, which has some of the utility methods (like printing the board and converting between `String` and `int` representations of the move, etc), but many abstract methods that you will have to implement in your `LastnameOthelloState` class.

`LastnameOthelloNode.java` This class inherits from the`LastnameOthelloState` class and implements the `GameNodeI` interface and must implement the im-

portant methods for constructing the game tree and getting the heuristic value of a node.

`LastnameBot.java` This class inherits from the `AIBot` class and implements the big method, `getMove`, which creates a new `LastnameGameNode` object to construct the game tree and then get the best move.

`LastnameHeurWeights.java` This class inherits the `HeurWeights` class and implements its abstract methods. The major method in this class is the `getHeurVal` method, which returns an integer heuristic score given a certain board.

The classes you'll inherit from are:

`OthelloState.java` This class is responsible for much of the game's properties, like displaying the current GameState, calculating the score, and getting the current player. It contains a number of abstract methods that you will implement in `LastnameGameState.java`.

`AIBot.java` This class inherits the `Player` class and provides the protected data and `getMove` template that you will implement in your `LastnameBot.java` class.

`HeurWeights.java` This class class is entirely a wrapper class with a single method template, `getHeurVal`, which you will implement in you `LastnameHeurWeights` class.

Interfaces you'll implement are:

`GameNodeI.java` The `LastnameOthelloNode` class implements this interface, which contains methods for getting the best move and the heuristic value of that move.

There are a number of other classes and interfaces necessary (and included) in running the game, but you will not directly interact with them, so their descriptions are not listed. Feel free to look at their code and ask questions in class about how they work, but keep in mind that the only files you will submit are the first four listed here.

Running the Game

You will need to add the `OthelloStateProcessor.jar` to your Project. In Eclipse, select Project –¿ Properties –¿ Java Build Path –¿ Add External JARs. Browse to the `OthelloStateProcessor.jar` file and open it.

Once you have tested your classes thoroughly in your own tester class, you should test them integrated into the full game. To "plug" into the GameRunner, simply instantiate one or both of your players with an instance of your `LastnameBot`.

Approaching this Project

This project is obviously quite challenging, but it need not be intimidating. This project is quite similar to the Tic Tac Toe AI you implemented in the third quarter except that the heuristic calculations for this game will be much more sophisticated. Here are a few steps you could follow if you're having trouble getting started.

1. Test out many of the more complicated methods you will implement from the abstract `OthelloState` class. If these don't work, your game won't work at all, so test these thoroughly before you get into the GameTree creation and other complicated aspectso of the game.

2. You should simply get the game running with a vary simple heuristic method, perhaps involving just the current score at a particular GameState.
3. Practice a number of times playing (as a User) against your simple AI so that you can both catch any hard-to-find bugs in the AI (wrong moves, etc) and so you can get a feeling for the weaknesses in the the AI's strategy.
4. Read a good bit about Othello strategy. There are a number of useful guides out on the internet.
5. Once you get into sophisticated heuristics, you should make a new class that inherits the `abstract` class `HeurWeights` and includes all of the helper methods (for determining your corners weight, for example) that you'll use to generate the final, single heuristic score of a particular GameState.
6. Once you get a fairly functional heuristic algorithm, you can then delve into optimization techniques to make the most of the time you have to make your moves.

Optimization

There are a number of ways that you can optimize your AI to yield better, or at least more forward-looking, results. Since you are working within a time limit, you want to make the most of that time but not go over your limit (resulting in disqualification). Here are a few strategies you may want to try out once you get a basic, working version of your AI running.

1. Employ alpha-beta pruning, in which moves that a player will never make are eliminated before they are tested. This technique results in significant time-efficiency, thus allowing your AI to probe deeper in the game tree than it otherwise would have been able to do.
2. Use iterative deepening, in which you create multiple trees of increasing depth (so, for example, you might start with a tree of depth 5 and then make one of 6, 7, etc). The advantage of iterative deepening is that it allows you to make the best use of your time. At the beginning of the game, for example searching a tree to depth 7 might take 15 seconds, whereas at the end of the game, it might only take 7 seconds. Therefore, you never know the highest (and most beneficial) depth to search for, so you start out at a starting depth and iterate through subsequent depths until your time is up.
3. You can use Java's `Timer` class to keep track of how much time you have spent processing your tree. This will allow you to continue processing until the very last minute, possibly gaining a small (but perhaps significant) advantage. You should also look into Java's multithreading capabilities, which allow you to basically run two different bits of code at the same time. (The timing aspect of the GameRunner uses multithreading. Feel free to look at the GameRunner code if you like to see an example.)
4. Test many, many, many different ways of calculating your heuristic score for a particular GameState. Remember, a more complicated heuristic calculation isn't always better if it takes a long time since it may mean you don't get to probe the game tree as deep as you otherwise might have.
5. You may find it useful to set up some internal battles of different AIs against each other to determine if one technique is more effective than another.