Lab 5

CS 3793/5233 – Fall 2016 assigned November 9, 2016
Tom Bylander, Instructor due midnight, December 7, 2016

In Lab 5, you will program a player for a version of the Gomoku game. A program that performs Minimax search is provided to you. It is up to you to improve this program with your weights for the evaluation function. For better play, more substantial effort is needed to incorporate Alpha-Beta pruning and intelligent time management. All of your programs will be played against each other. Prizes will be given for the top five programs. In general, your grade on the lab will depend on the quality of the player that is implemented.

Gomoku

Gomoku is played on an n-by-n board. We will use $n = 10$. The game is the same as Tic-Tac-Toe except that the goal is to get five of your pieces in a line (row, column, or diagonal) rather than three.

As with Tic-Tac-Toe, the two players (X and O) alternate turns with X playing first. On each turn, a player puts his symbol on an empty square on the board. The first player to get five in a line wins.

Programming Framework

The `lab5.zip` download for this lab contains `Interact2.java`, `Gomoku.java`, `GomokuPlayer.java` and `HumanPlayer.java`.

`Interact2.java` sets up the threads for one `Gomoku` object and two `GomokuPlayer` objects. One `GomokuPlayer` object is agent 1; `Interact2` prints a `you are agent 1` message to this player. The other `GomokuPlayer` object is player 2; `Interact2` prints a `you are agent 2` message to this player.

Any line from an agent is output to the `Gomoku` object with a `1` or a `2` appended in front, indicating agent 1 or agent 2. Any line from `Gomoku` has a `1` or a `2` in front indicating which agent the line should go to.

Each agent inputs and outputs moves according to whose turn it is. The `Gokoku` object runs the game; it will output a `win for agent 1` line when agent 1 wins the game, `win for agent 2` line when agent 1 wins the game, and `draw` if the game is a draw.

The `GomokuPlayer` code is already written to follow the protocol. You should copy this code to another class, naming it `YourNameHerePlayer`, replacing `YourNameHere` with your name. Besides changing the name, the only changes you *must* make is to `PLAYER_WEIGHTS` and `OPPONENT_WEIGHTS`.

The download should be set up to see the game as it progresses; this is done by setting the `debug` variable in `Gomoku.java` to `true`. If you want to play against `GomokuPlayer`, change the initial assignment to `agent1` or `agent2` in `Interact2.java` to an instance of `HumanPlayer`.

Agent

Your task is to write a Gomoku player that will improve on `GomokuPlayer.java`.

The only part you need to change are the values in `PLAYER_WEIGHTS` and `OPPONENT_WEIGHTS`. These weights are used to evaluate positions three *ply* away from the current board (two moves by `player` and one move by `opponent`). The `gomokuEval` method enumerates each possible five in line (row, column, diagonal). If `player` has x pieces in a possible five in line and `opponent` has 0 pieces, then `PLAYER_WEIGHTS[`x`]` is added to the evaluation. If `opponent` has y pieces in a possible five in line and `player` has 0 pieces, then `OPPONENT_WEIGHTS[`y`]` is added to the evaluation.

The weights are easy to change, but choosing good ones will take some insight into Gomoku and a three-ply Minimax search.

The following two items would require significant programming.

1. **Alpha-Beta Pruning**. Alpha-beta pruning can allow the depth of your search to be 1 or 2 moves deeper compared to minimax, depending on how clever you are in programming this method. I was able to search 2 moves deeper by using the evaluation function to order the moves in the first call to *maxNode* from highest to lowest.

2. **Time Management**. Your program gets roughly 30 seconds to make all of its moves. When I run the download, it takes 22 seconds to complete the game on my machine. If `GomokuPlayer` is forced to play the game to a draw, it uses less than 20 seconds.

   Your instructor is not going to time this exactly (although he may be forced to in order to complete a tournament), but a program that consistently takes too much time will not be eligible for extra credit.

   Early in the game, the branching factor is 100. Later in the game as the board fills up, the branching factor decreases rapidly, so you can search to a greater depth, perhaps all the way to the end of the game. One possibility is to modify the depth parameter so it is lower at the beginning and higher at the ending.

Prizes

I will be running your lab against the other students' labs and programs of my own (`GomokuPlayer.java`, a version of `GomokuPlayer.java`, and one more competitive program that searches 4 ply deep). The prizes (my programs are not eligible) for the top five programs will be 100, 80, 60, 40, and 20 points, respectively. In the event of ties, the prizes will be distributed evenly.

Otherwise, your grade on the lab will depend on the performance of your program.

1. A score of 80. Your program wins both games vs. `GomokuPlayer`.

2. Higher scores. Your program wins both games vs. `GomokuPlayer`. In addition, I will create a version of `GomokuPlayer` where no weight is changed by more than 10. A score of 85 is if your program draws one and loses one against this player. A score of 90 is if your program wins one and loses one against this player (or draws twice). A score of 95 is if your program wins one and draws one against this player. A score of 100 is if your program wins both games against this player.