CSE 150 Programming Assignment #2

Due: 5/6 at 11:59 PM

1 Overview

In this project, you will develop agents to play the m,n,k-games that includes "Gomoku" and "tic-tac-toe". You will create a simple minimax agent, an alpha-beta pruning minimax agent and a custom agent of your own design that should outperform your minimax agents.

2 The m,n,k-game

The m, n, k-game is a two-player deterministic game where two players alternatively place stones on a game board. The rules of the generic m, n, k-game is simple:

- There are two players, usually denoted as "black" and "white" (or "X" and "O" for tic-tac-toe).
- The game is played on an $m \times n$ game board. Like the game of "Go", the stones are usually placed on the intersections of lines drawn on the board. (This doesn't really matter for a computer simulation - we just have $m \times n$ locations to place "stones".)
- Black plays first and the players take turns alternatively. At each turn, a player places a stone of his color on an unoccupied location on the board.
- The player who is the first to get k or more stones in a row (horizontally, vertically or diagonally) wins. (This is a departure from the game of Gomoku, where it must be *exactly* five stones in a row.) We are calling these "connected" stones as *streaks* in this assignment. If neither player has streaks of lengths k and the board is full, the game is a draw.

The goal of this assignment is to develop an agent that can play on a large board (up to $19, 19, 6$-game), but we will also develop players for smaller games in the process.

3 Provided Code

We have provided code to deal with the basic mechanics of the game, and some stub code for each problem already. The game state, player and actions are implemented in the `State`, `Player` and `Action` classes in `src/assignment2.py`, respectively. While you will not be making modifications to the files under `src`, you will be implementing and submitting various "agents" in the `solutions` folder.

The `board` attribute of the `State` object represents the game board in a tuple of M tuples of N elements (M rows and N columns). In this array, 0 represents the empty location, and 1 and 2 represent the stones placed by players 1 and 2, respectively.

Testing Your Agent

We have also provided two `Player` implementations:

- The `RandomPlayer` in `random_player.py` is a player that places stones at random unoccupied location on the board.
- The `HumanPlayer` in `human_player.py` is a player that takes stone placements from the console. You can use this player to play against the various agents for testing. You input the locations by entering the two numbers separated by a space. For example, entering `0 3` would place a stone on the first row, 4th column.

There is a command-line game UI in `run_game.py` that you can use to make the agents play against each other (or against you!) The syntax is:

```
$ python run_game.py [M] [N] [K] [timeout] [PlayerClass1] [PlayerClass2]
```

Here, `[M]`, `[N]`, `[K]` are the board sizes and the streak length needed to win the game. `[timeout]` specifies the maximum amount of time allowed to make a move, in seconds; if it is -1, there'll be no timeout. `[PlayerClass1]` and `[PlayerClass2]` are the class names of the first and second players.

For instance, to play a game of tic-tac-toe against the `RandomPlayer` AI without time limit:

```
$ cd src
$ python run_game.py 3 3 3 -1 HumanPlayer RandomPlayer
```

You can also perform a subset of automated testing by running `test_problems.py` in the `tests` directory:

```
$ cd tests
$ python test_problems.py
```

This executes the test corresponding to problems 1 to 4 by giving some input in the `in` directories and comparing the output against ones in `out` directories. The tests will be reported as a "failure" if the output of your code does not match the text files the `out` directories. A good practice is to run the tests before doing the problems and observe that they fail. Then, once you implement the problems correctly, your tests should pass. There will be more test cases in the actual online submission site, and you are encouraged to add more of your own inputs and outputs in the `problems` directory!

4 Problems

Problem 1

Implement a minimax search algorithm in `p1_minimax_player.py`. The game tree can expand quickly with the board size, but you do not need to worry about the efficiency for this problem - this will only be tested with small boards.

When there are moves with the same values, choose the move that comes earlier in "right-left then top-down" order, with respect to the board. (Note that this should happen automatically if you iterate the actions returned by the `action()` method.) So, for instance, if placing stones at $(0, 1)$ and $(1, 0)$ yield same values, $(0, 1)$ should be returned, regardless of the last move.

Examples

input1.txt

```
State(3,(
    (0,2,1),
    (0,1,2),
    (0,0,0),
), last_action=Action(2, (0,1)))
```

output1.txt

```
Action(1, (0, 0))
```

Note that there are multiple moves for Player 1 to win, but the *first* winning action returned by `state.actions()` is at $(0, 0)$. (Think about why the algorithm does not return the "obvious" winning action at $(2, 0)$.)

input2.txt

```
State(5,(
    (9,9,9,9,0,0),
    (9,9,9,9,0,0),
    (9,9,9,0,0,9),
    (9,9,1,9,1,9),
    (9,1,9,9,1,9),
    (0,9,9,9,1,9),
    (9,9,9,9,2,9)
), last_action=Action(2, (6,4)))
```

output2.txt

```
Action(1, (1, 4))
```

In this larger board, "9" is used as a dummy placeholder where neither of the player can place the stone. This is the "four-three" situation in the game of Gomoku; here, after the player 1 places at $(1, 4)$, player 2 cannot stop her from connecting 4 stones in the next move. (The 4 stone chain without ends blocked cannot be stopped.)

Problem 2

Implement a minimax search algorithm with **alpha-beta pruning** and **transposition table** in `p2_alphabeta_player.py`. With these two optimizations, the code should be able to handle slightly larger branching factors than the simple minimax agent.

Examples

input2.txt

```
State(5,(
    (9,9,0,0,0,0),
    (9,9,9,0,0,0),
    (9,9,9,0,0,9),
    (9,9,1,9,1,9),
    (9,1,9,9,1,9),
    (0,9,9,9,1,9),
```

```
    (9,9,9,9,2,9)
), last_action=Action(2, (6,4)))
```

output2.txt

```
Action(1, (1, 4))
```

This is essentially the same situation as in **Problem 1**, but with slightly more empty locations where the stones could be placed. The `MinimaxPlayer` would take too long to evaluate, but the `AlphaBetaPlayer` should be able to handle this case.

Problem 3

Implement a simple evaluation function in `p3_evaluation_player.py`. In contrast to the previous two problems, you don't have to implement the `move` method, but you will implement the *state evaluation function*; the agent will then play at the location that would yield the best evaluation.

The `evaluate` method in this problem should return **the length of the longest streak on the board** (of the given stone color), divided by K. Since the longest streak you can achieve is K, the value returned will be in range $[1/K, 1]$.

Examples

(In these examples, the `color` is assumed to be 1, *i.e.* the evaluation is for the player 1.)

input1.txt

```
State(4,(
    (0,2,1,0),
    (0,1,2,0),
    (0,0,0,0),
    (0,0,0,0)
), last_action=Action(2, (0,1)))
```

output1.txt

```
0.5
```

The longest streak is 2, so the `evaluate()` method should return $2.0/4.0$.

input2.txt

```
State(5,(
    (0,0,0,0,0,0),
    (0,0,0,0,1,0),
    (0,0,0,1,0,0),
    (0,0,1,0,1,0),
    (0,1,0,0,1,0),
    (0,0,0,0,1,0),
    (0,0,0,0,2,0)
), last_action=Action(2, (6,4)))
```

output2.txt

```
0.8
```

Problem 4

Implement a custom agent that can play any m, n, k-game, up to $K = 6$ on 19×19 board ($M, N = 19, K = 6$) in the `p4_custom_player.py` file. Rename the class and override the `name()` method to your liking. After the homework submission deadline, we'll also have **a tournament among all submitted agents** with different per-move time limits. There'll be prizes for the winning agents and extra credits for especially clever / innovative agents!

A good start will be to implement an iterative deepening minimax search with alpha-beta pruning, transposition table and move-ordering based on the evaluation function of **Problem 3**. Opening heuristics may also be needed, especially when the board is so large. However, you're free to improve on these.

The code should be written so that it can search to arbitrary depths depending on the board size and allowed time. To do this, follow these guidelines:

1. Check for the `self.is_time_up()` condition often in your main computation loop (searching through the game tree, for example).
2. Once the `self.is_time_up()` becomes `True`, your code should finish up and return the move as quickly as possible. There will only be about one second allocated for this portion, so it should only do "quick" operations to finish up (such as calculating the best move from the tree you've searched.)

Again, make sure `self.is_time_up()` is checked somewhat frequently, so that your code will be less likely to be terminated abruptly. If the method does not return a move in time, a random move will be played instead! You can test the agents under time limit by specifying the number of seconds per move in the fourth parameter of the `run_game.py`. For example, to play a $19, 19, 6$-game with a 10-second per move limit,

```
$ cd src
$ python run_game.py 19 19 6 10 HumanPlayer YourCustomPlayer
```

Problem 5

Submit a write-up for this project in PDF. You should include the following:

- Description of the problem and the algorithms used to solve problems 2 - 4.
- Describe the approach you used in **Problem 4** and other approaches you tried / considered, if any. Which techniques were the most effective?
- Evaluate qualitatively how your custom agent plays. Did you notice any situations where they make seemingly irrational decisions? What could you do/what did you do to improve the performance in these situations?
- What is the maximum number of empty squares on the board for which the minimax agent can play in a reasonable amount of time? What about the alpha-beta agent and your custom agent, in the same amount of time?
- Create multiple copies of your custom agents with different depth limits. (You can do this by copying the `p4_custom_player.py` file to other `*_player.py` and changing the class names inside.) Make them play against each other in at least 10 games on a game with $K > 3$. Report the number of wins, losses and ties in a table. Discuss your finding.
- A paragraph from each author stating what their contribution was and what they learned.

Your writeup should be structured as a formal report, and we will grade based on the quality of the writeup, including structure and clarity of explanations.