CSE 150 Programming Assignment #2

Due: April 29, 2016, 23:59

1 Overview

In this project, you will develop agents to play the **Mancala Game**. 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 Mancala Game

Mancala is a family of stone-and-pit games with ancient African origins, but now we look at one particular type of the game. At the start of the game, there are 2 rows of M pits with N stones per pit, and 2 goal pits at each end.

The following section uses $M = 6, N = 4$ as the example to illustrate the game rule but you agents **must** handle varying M, N.

2.1 Start

Each player has $M = 6$ **pits** with $N = 4$ **stones** per pit lined up horizontally in front of him or her, and a **"goal" pit** on the side to the right. When you play against the computer, the board will be in the following configuration:

```
  4 4 4 4 4 4
0             0
  4 4 4 4 4 4
```

With your pits at the bottom, your goal pit at the right, and the computer's pits at the top and goal pit at the left. The picture above shows the start position: each pit has 4 stones in it, except for the goal pits which are empty.

For the purposes of this implementation, the pits on the player's side are indexed 0 to 5, with index 6 being the player's goal pit, and the computer's pits are indexed 7 to 12 from right to left (i.e. continuing counter-clockwise) with the computer's goal pit index 13.

2.2 Move[1]

A player chooses one of the 6 pits on his or her **own** side of the board (the chosen pit must be **non-empty**) and redistributes the stones **one-by-one** going **counter-clockwise** around the board, starting with the pit following the one picked. The opponent's goal pit, if reached, is **skipped**.

For example, assuming it is your turn and the board is:

```
  4 4 4 4 4 4
0             0
  4 4 4 4 4 4
```

You choose to move from pit index 4, the resulting board would be

```
  4 4 4 4 5 5
0             1
  4 4 4 4 0 5
```

[1] In the other version of the game, if the player's last stone ends in the player's own goal pit, the player gets another turn. This is NOT considered in this programming assignment to simplify the rules.

2.3 Capture

When moving, if the player's **last** stone ends in an **empty pit** on his or her **own** side, the player "captures" all of the stones in the pit directly across the board from where the last stone was placed (the opponents stones are removed from the pit and placed in the player's goal pit) as well as the last stone placed (the one placed in the empty pit).

There must be stones in the opponent's pit captured; otherwise the capture will not take place.

For example, assuming it is your turn and the board is:

```
  4 4 4 4 5 5
0             1
  4 4 4 4 0 5
```

You choose to move from pit index 0, the resulting board would be

```
  4 4 4 4 0 5
0             7
  0 5 5 5 0 5
```

2.4 End

The game ends when both players cannot move on his or her turn.

2.5 More

For more details on Mancala and its variants visit: http://en.wikipedia.org/wiki/Mancala

3 Provided Code

The game infrastructure is written in Python 2, thus you need to write your solution in Python 2.

3.1 Files

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.

You must go through `src/assignment2.py` to understand the game and call the APIs for your own code.

3.2 Indexing

Pits from index 0 to $M-1$ are the pits on your side (bottom). Pit M is your goal pit. Pits from $M+1$ to $2M$ is the opponent's pits. Pit $2M+1$ is the opponent's goal pit.

Bottom row is represented as row 0 while top row is represented as row 1.

3.3 Break tie

If multiple actions result in the same utility, then you must choose the pit with **smaller** index.

3.4 Testing Your Agents

We have also provided two `Player` implementations:

- The `RandomPlayer` in `random_player.py` is a player that moves stones at random legitimate pit on the board.
- The `HumanPlayer` in `human_player.py` is a player that moves stones from the console. You can use this player to play against the various agents for testing. For example, entering `0` would move stones on the pit index 0.

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:

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

[M] is the number of pits on each side. [N] is the number of stones per pit. [timeout] specifies the maximum amount of time in second allowed to make a move, in seconds; if it is -1, there'll be no timeout. If timeout, the game will take a random move for the player. [PlayerClass1] and [PlayerClass2] are the class names of the first and second players.

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 with reasonable M, N 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

For any player code, you need to implement the `def move(self, state)` method, following the exact signature of the method. It takes a **State** as the input, and returns an **Action**.

You can add other methods you like in the class.

4.1 Problem 1

Implement a minimax search algorithm in **solutions/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 index order.

Example

Input:

```
State.initial(2, 1)
State([1, 0, 1, 1, 1, 0], 1)
```

Output:

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

The board is initialize as $M = 2, N = 1$, with initial board as:

```
  1 1
0     1
  1 0
```

The player is played as upper row (row=1), and the output action for this player at uppper row to move stones in pit index 4.

4.2 Problem 2

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

Example

Input:

```
State.initial(3, 3)
State(None, [3, 3, 3, 0, 3, 3, 3, 0], 0)
```

Output:

```
Action(0, 2)
```

4.3 Problem 3

Implement a simple evaluation function in `solutions/p3_evaluation_player.py`. In contrast to the previous two problems, you don't have to implement the move method since it has been implemented for you, but you will implement the state evaluation function; the agent will then play at the location that would yield the best evaluation (without searching the game play tree).

Use heuristic function as below:

$$\frac{1}{2MN}\Big(\text{stonesInYourGoal}-\text{stonesInOpponentGoal}+\text{stonesOnYourSide}-\text{stonesOnOpponentSide}\Big)$$

Example

Input:

```
State.initial(4, 4)
State([8, 0, 4, 4, 0, 4, 4, 4, 4, 0], 0)
```

Output:

```
Action(0, 2)
```

4.4 Problem 4

Implement a custom agent in the `solutions/p4_custom_player.py` file.

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.

You may wish to consider the number of pieces your agent currently has in its goal pit, the number of blank spaces on its side of the board, the total number of pieces on its side of the board, specific board configurations that often lead to large gains, or anything else you can think of. You should experiment with a number of different heuristics.

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 code (searching through the game tree, for example). You can check it anywhere within the `Player` class methods.
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 (1 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.)
3. If you don't finish, the game will force your agent to take a random action.

Additionally, if you need to store any data for the duration of the game, you can initialize them in the constructor (`__init__(self)` method).

5 Submission

Write a report for this project in PDF. You should include the following:

- Briefly describe of your design to solve problems 1 - 3.

- Describe the approach **in details** 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 pits and stones per pit (i.e. M, N) 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 agent 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. Report the number of wins, losses and ties in a table. Discuss your findings.
- 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.

Submit:

1. Your `solutions` folder containing your player codes.
2. Your report.