Assignment 2: Othello

Acknowledgments: Much of this assignment is copied from an assignment written by Professor Grant Braught. The code framework derives from an Othello implementation by Evan A. Sultanik of Drexel University, with certain additions by Professor Braught.

This assignment consists of both code and written components. The criteria for grading code are posted on the course website. The criteria for grading the written components are described separately for each component below.

Question 1. (30 points) A MiniMax Othello Player

Using the starter code provided on the course web page, implement a player for the Othello game that uses the MiniMax algorithm (with cutoff at a fixed depth) to select its move. You must create a new class called `MMOthelloPlayer` player, which extends the `OthelloPlayer` class and implements the `MiniMax` interface.

Comments:

- None of the given code may be modified.
- Your player must have 2 constructors:
  - `public MMOthelloPlayer(String name)`
  - `public MMOthelloPlayer(String name, int depthLimit)`
- Both constructors must start with the statement: `super(name);`
- The one-argument constructor may initialize the depth limit as you see fit.
- The search must not generate nodes beyond the depth limit.
- Your player must correctly report all of the values described in the `MiniMax` interface.
- Your static evaluation function must be completely contained in the `staticEvaluator` method specified by the `MiniMax` interface.
- You may ignore the `deadline` parameter to `getMove` at this point.
- Use the following simple static evaluation function: the number of pieces owned by MAX.

In writing your OthelloPlayer and later your static evaluation function you may find useful the methods of the `GameState` class that are listed below. You will need to examine the source code for specific information on how to invoke these methods and exactly what they return.

- `getValidMoves` – get a list of all valid moves for the player whose turn it is.
- `isLegalMove` – returns true if the specified move is legal in the current state.
- `applyMove` – returns the state reached by applying a specified move to this state.
- `getSuccessors` – get the states that can be reached from this one via valid moves.
- `getPreviousState` – get the state from which this one was reached.
- `getPreviousMove` – get the move that led to this state.
- `getCurrentPlayer` – get the player whose turn it is.

- `getOpponent` – get the player whose turn it is not.
- `getSquare` – find out which player owns a given square.
- `getScore` – get the score for a specified player.
- `getWinner` – get the player who won the game.
- `getStatus` – get the status of the game (has someone won? Was it a tie? Still playing?)

You can run the Othello program with or without a GUI interface. To run the program with a GUI use a command line similar to:

```
java Othello HumanOthelloPlayer RandomOthelloPlayer
```

where `HumanOthelloPlayer` and `RandomOthelloPlayer` are the names of classes that implement the OthelloPlayer interface.

To run the Othello program with a console interface use the command line:

```
java Othello –nw HumanOthelloPlayer RandomOthelloPlayer
```

The GUI interface is a bit nicer for human players but is a bit flaky. Thus, when performing the experiments later in this assignment it is recommended that you use the console version.

It is also possible to place a limit on the amount of time a player may use to choose its move. If the player does not move within the time limit, its piece will be placed randomly. To specify a time limit, a `–d` flag is included in the command line. For example, the command line:

```
java Othello –nw –d 2 HumanOthelloPlayer RandomOthelloPlayer
```

specifies a time limit of 2 seconds for each move.

Finally, you may find that Java runs out of heap space when running your program. This problem can be put off (though perhaps not eliminated) by setting the maximum heap space for Java. The following command line runs Othello with a maximum heap space of 1GB:

```
java –Xmx1000m Othello –nw –d 2 HumanOthelloPlayer RandomOthelloPlayer
```

Question 2. (30 points) An alpha-beta Othello player

Create a copy of your OthelloPlayer from question 1 in a new class, called `ABOthelloPlayer`, and modify it so that it uses alpha-beta pruning during search.

Question 3. (20 points) Comparing minimax and alpha-beta

(a) Design and conduct an experiment to determine the following values:
- the number of nodes per second that your MiniMax player can generate
- the number of nodes per second that your Alpha-Beta player can generate
- the approximate average branching factor of the search tree in an Othello game

- the approximate effective average branching factor for your implementation of alpha-beta search

Provide a written description (1-2 pages) of the experiment that you performed containing sufficient detail that one of your classmates could repeat your work. At the top of the report, clearly state your estimates of the four values listed above. The grade for this question will be based on the elegance and suitability of the experimental design, as well as the clarity, correctness, spelling and grammar of the description.

(b) Answer the following questions using your data from part (a). Justify your answers.

1. Approximately how long would your MiniMax and Alpha-Beta players take to explore the game tree to a depth of 12 plies?
2. If your MiniMax player can explore to a depth of D in a given time T, approximately what depth would your Alpha/Beta player be able to explore in the same time T?

Question 4: (20 points) Othello Steel Cage Death Match

(a) (3 points) Create an OthelloPlayer, including an improved static evaluation function, to participate in our Othello Steel Cage Death Match. Your player must be implemented in a class called `XXXOthelloPlayer`, where `XXX` is the name you have chosen for your entry into the Death Match. The only limitation is that your player must use some variant of the MiniMax algorithm (perhaps with Alpha/Beta pruning, perhaps with other optimizations that you develop). In class we will run an Othello tournament in which your OthelloPlayer implementations will play against each other. This tournament will be played using the console version of the game and the players will have a maximum of 3 seconds per move. The following line of code will tell you how many milliseconds remain before the time for the current move expires:

```
long msLeft = deadline.getTime() - System.currentTimeMillis();
```

Your `XXXOthelloPlayer` must be self-contained, as it will be copied and pasted into the example framework in order to run the tournament. Therefore, if you need additional helper classes, include them as nested classes within `XXXOthelloPlayer`.

You are encouraged to research Othello-playing strategies for this part of the assignment, and it is permissible to copy any *ideas* your research reveals, provided they are appropriately cited in part (b) below. However, you may not copy any code whatsoever, from any source. All code submitted (except for the framework provided, of course) must be your own work.

The grade for this part of the assignment will be based purely on performance in the Othello Steel Cage Death Match: first place gets 3 points, second place gets 2.5 points, third place gets 2 points, and so on.

(b) (17 points) Write a description of the strategy you implemented (0.5-1.5 pages). Remember to cite any sources on which your ideas were based. The grade for this question will be based on the originality and appropriateness of the ideas, as well as the clarity, correctness, spelling and grammar of the description.

Submission instructions

Please submit exactly 2 files to Moodle. The first file will be a .zip file of the three relevant .java files: `MMOthelloPlayer.java`, `ABOthelloPlayer.java`, and `XXXOthelloPlayer.java`. The second file will be a single document in any reasonable format (e.g. PDF, OpenOffice, Microsoft Word) containing the written answers to questions 3 and 4(b).