Mathematics 121: Computer Science I (Frachtenberg, Spring 2016)

Project 3: Reversi

Goal

The goal of this project is to implement the game of Reversi (also known as Othello). You will implement the game mechanics, the user interface (UI), and at least one computer strategy using the Minimax algorithm.

There are many implementations of this game online that will help you familiarize yourself with the rules and strategy of the games. Check out these links, for example: othello, reversi, reversi. Take an hour or so playing online to understand the rules of the game and some insights on successful strategies. You are provided a few Python files to get you started and to preserve a uniform interface among all implementations.

Tasks

You will be required, as a minimum, to accomplish the following tasks:

- Implement all of the missing methods in the Board class (board.py). These methods compute things such as which moves are legal for each player and how to change the board state after a legal move was taken.
- Implement a simple text-based user interface for a human player.
- Implement a graphic user interface.
- Implement a brute-force game strategy using the Minimax algorithm.

In addition, you will have plenty of opportunity for extra credit by improving your strategy and your graphical user interface.

The following is a breakdown of the steps (and relative grade) you should take. You can change the order of the steps as you see fit (or divide them between two people), but it's designed to be the most logical order.

1. Text UI (10%)

The UI class (ui.py) defines a base class for presenting the board state. Create a new class (TextUI) in a file called text_ui.py that derives from UI and presents the board on the console. You will have to implement at least two methods, in addition to __init__: showBoard() and showLegalMoves() (implementing showMove() is optional but recommended). Read up their documentation in ui.py to figure out exactly what's required from each method.

2. Text (human) player (10%)

Next, implement the text-based interface for a human player. You start from the Player class (in player.py) and create a new derived class called TextPlayer in a file called text_player.py. There is only one new method to implement, chooseMove(). Ensure your implementation won't let any illegal moves be selected, and that it parses the user's input string correctly.

3. Reversi mechanics (40%)

It's now time to implement the game rules. By the end of this step, you should be able to play (against yourself, another human player, or the provided random player).

To accomplish this, you have to implement all the methods of the Board class in the board.py file. You do not need to derive this class (but you can)--you can simply replace all the "pass" implementations with real ones. There are 6 methods to implement + __init__(). Of these, only two are relatively complicated:

- legalMoves() returns a list of all legal moves for a given player, so you have to figure out very carefully what makes a move legal or illegal. This is potentially a place for lots of bugs, so plan this method carefully, break it up into small step, and keep it clean and simple.

- makeMove() changes the board state after a move was taken (you may crash if the move isn't legal, but that must never happen--Player classes should always pick a legal move).

Finally, you can now play a full game of Reversi using the reversi() function in reversi.py. Just past to it a list with two Player objects (at this point, of class TextPlayer or class RandomPlayer) and a user interface (at this point, of class TextUI), and have some fun playing and debugging!

Your game might look something like this (you can tweak the appearance to your tastes):

```
  01234567

0 ········ 0

1 ········ 1

2 ········ 2

3 ···●○··· 3

4 ···○●··· 4

5 ········ 5

6 ········ 6

7 ········ 7

  01234567

Player 1: 2 Player 2: 2
```

```
Total moves: 0

Available moves: [(4, 2), (5, 3), (2, 4), (3, 5)]

Type a y,x pair (no parens):
```

4. Minimax strategy (20%)

The next step is to write a computer strategy that's actually halfway decent. (An example of a rudimentary computer strategy that's pretty easy to beat is provided in random_player.py.) The strategy you'll be implementing is a brute-force search, meaning that it will evaluate every possible move and counter move up to a predetermined depth, and pick the best possible outcome. This strategy is very powerful for board games such as Chess, Checkers, and Reversi, and only has one major flaw: it is of exponential complexity. Since it would take too long to compute a full game tree, typical implementations limit the actual searched tree depth (how many moves forward) to a reasonable value that represents a trade-off between computation time and move quality.

The Minimax search algorithm assumes two players, the "maximizer" (current player, who seeks to maximize its own ranking) and the "minimizer", the opponent. Another parameter is an evaluation function that, given a certain board, evaluates it from the maximizer's point of view and returns a score for how good it is for this player. Finally, It also receives a given depth to search the game tree. So, for example, if the depth is 1, Minimax evaluates all current moves for the maximizer and returns the one with the highest evaluation score. If the depth is two, then for each one of the maximizer's moves, it recursively checks all of the minimizer's moves and returns the one with the lowest score (since the minimizer is trying to foil the maximizer, that is, to lower its score).

A detailed explanation of Minimax (as well as a pseudo-code example) can be found here.

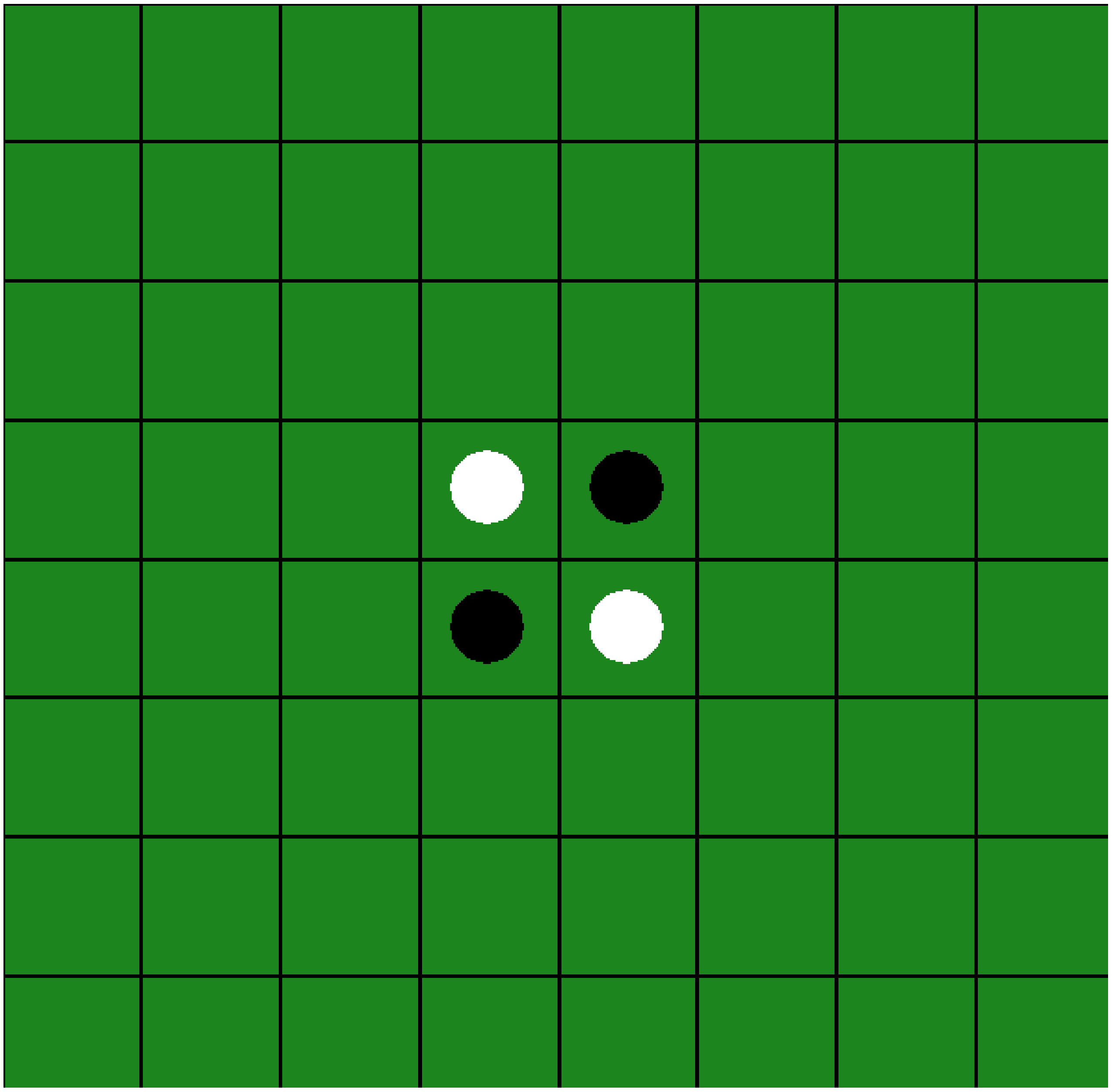
Your goal is to implement a new MinimaxPlayer class (in minimax_player.py) that derives from Player. It should receive an additional init parameter for the depth of the tree to search (and depth of 3 should result in a reasonably snappy player and depth of 4 should still be quite playable. Once debugged, try to play against your new strategy or pit it against itself (possibly with different tree depths). Ensure that higher depth results in higher win rates as well.

Your evaluation function returns a score for every board state that should reflect how "good" this board is toward the maximizer player (it can be negative if the player is losing). You are free to implement any evaluation function you like, even something as simple as counting how many more discs the maximizer player has than the minimizer, or some such approximation of "winning". But remember, the function is critical to the strength of the strategy, so if you want a player that's hard to beat, you'll want to spend more time on improving this function.

5. Graphic UI (20%)

The final step is to implement a graphic game UI, using the same graphics.py library from project 2. This step includes two tasks: a new GraphicUI class (derived from UI) and a new GraphicPlayer class (derived from Player). The former shows the board state (and current player score and number of moves), and the latter lets the user pick its next move with mouse, by clicking on an available (and legal) tile.

The board doesn't have to be fancy or visually show legal moves (see step 6 below for that). It also doesn't need to animate game moves, but the interface should still feel responsive, not sluggish. It can be as simple as this, for example:



6. Enhanced graphic UI (extra credit)

If you do want to improve the look and feel for your game, go for it! You can receive up to 25% extra credit for a really pretty interface. There are no limits to your design--let your creative juices flow! But here are some ideas you might consider:

- Showing the human player the available legal moves, so they don't have to guess or think about it too hard.
- Animating changes in board state (basically, new moves and flipping existing discs).
- 3D or shadow effects.

7. Improved strategy (extra credit)

I will evaluate the quality of your Minimax strategy by running it at depth 3 or so (2 if it's too slow, 4 if it's zippy) against my code, with about 25 different evaluation functions and depth combinations. Any one of those combinations that your strategy beats gives you an extra point.

You have two main paths to make your strategy more competitive:

- You can make your board evaluation function better, so that it leads to more victories. This would require first a good understanding of what leads to winning, and a lot of tweaking and experimentation with different strategies and parameters.
- You make your code faster, so I can run it at a higher depth. That might include some simple performance optimization, parallelizing your code to take advantage of multiple threads (I can use 28 cores to evaluate your code) or some more interesting algorithmic improvements such as Alpha-Beta pruning.

Notes

- Please submit your work here in the Moodle page as an attached zip file. It needs to include your complete source code files, as well as a README file describing your work and your strategy.

- You may (but don't have to) work on this in pairs. If you do, submit your work only once, but include the name of both partners in the README file. You're encouraged to share notes and ideas with other teams, but not code.
- If you search, you will likely find implementations of the game in Python on the Web. Do not use these! They will not help you much, because you have to conform to the classes and interfaces specific to this project. And more importantly, you are expected to submit your own work, not somebody else's, and the graders are made aware of those works as well.
- Coding style matters, so be sure to document your code, choose proper names, maintain class attribute privacy, keep your functions short and to the point, and don't repeat code. In other words, adhere to all the clean programming principles we've covered in class.

pr3_files.zip

Submission status

Submission status	No attempt
Grading status	Not graded
Due date	Wednesday, May 4, 2016, 8:00 PM
Time remaining	19 days 22 hours
Last modified	Sunday, April 3, 2016, 8:24 PM
Submission comments	▸ Comments (0)

Add submission

Make changes to your submission

NAVIGATION

Home

- My home

  Site pages

  My profile

  Current course

  Math121S01S02S16

  Participants

  General

  Introduction to computing and programming

  Elements of programming

  Program control

  More functional functions

  Sequences

  Abstract data types

  Object-oriented programming

  Introduction to computer graphics

  File I/O

  Algorithms on sequences

  Data structures

  The Minimax algorithm

  Project 3: Reversi

  Reading material: CP 2.9

  Introduction to computer architecture

  My courses

ADMINISTRATION

My profile settings