Submitting your project

Submit the file

- `gomoku.py`

on Gradescope.

The `if __name __== __"main"__` block

All your testing code should be inside the `if __name __== __"main"__` block. You must not use global variables in this project.

Clarifications and discussion board

Important clarifications and/or corrections to the project, should there be any, will be posted on the ESC180 Piazza. You are responsible for monitoring the announcements there.

Hints & tips

- Start early. Programming projects always take more time than you estimate!
- Do not wait until the last minute to submit your code. You can overwrite previous submissions with more recent ones, so submit early and often—a good rule of thumb is to submit every time you get one more feature implemented and tested.
- Write your code incrementally. Don't try to write everything at once, and then compile it. That strategy never works. Start off with something small that compiles, and then add functions to it gradually, making sure that it compiles every step of the way.
- Read these instructions and make sure you understand them thoroughly before you start—ask questions if anything is unclear!
- Inspect your code before submitting it. Also, make sure that you submit the correct file.
- Seek help when you get stuck! Check the discussion board first to see if your question has already been asked and answered. Ask your question on the discussion board if it hasn't been asked already. Talk to your TA during the lab if you are having difficulties with programming. Go to the instructors' office hours if you need extra help with understanding the course content.

  At the same time, beware not to post anything that might give away any part of your solution—this would constitute plagiarism, and the consequences would be unpleasant for everyone involved! If you cannot think of a way to ask your question without giving away part of your solution, then please drop by office hours or ask by private Piazza message instead.
- If your message to the TA or the instructor is "Here is my program. What's wrong with it?", don't expect an answer! We expect you to at least make an effort to start to debug your own code, a skill which you are meant to learn as part of this course. And as you will discover for yourself, reading through someone else's code is a difficult process—we just don't have the time to read through and understand even a fraction of everyone's code in detail.

  However, if you show us the work that you've done to narrow down the problem to a specific section of the code, why you think it doesn't work, and what you've tried to fix it, it will be much easier to provide you with the specific help you require and we will be happy to do so.

How you will be marked

We will mark your project for the correctness of the functions that you are required to write. **Make sure that you follow the specifications exactly.**

Correctness

We will run your functions using a Python 3 interpreter. Please ensure that you are running Python 3 as well. To check what version of Python you are running, you can run the following in your Python shell:

```
import sys
sys.version
```

Syntax errors in your code will cause you to lose most of the marks for this project.

Note that **your functions must be implemented precisely according to the project specifications.** Their signatures should be exactly as in the project handout, and their behaviour should be exactly as specified. In particular, make sure that functions do not print anything unless the project specifications specifically demand that, and that the functions return exactly what the project handout is asking for.

The Gomoku Game

In this assignment you will implement a simple (and imperfect) AI engine for the game Gomoku, played on a 8×8 board. In Gomoku, there are two players. One player plays with black stones and the other player plays with white stones. A player moves by placing a stone on an empty square on the board. The player who plays with black stones always moves first. After the first move, the players alternate. A player wins if she has placed five of her stones in a sequence, either horizontally, or vertically, or diagonally. Please read

`http://en.wikipedia.org/wiki/Gomoku`

for more information about Gomoku. We will be playing the `standard` variant.

1 The starter code

The file `gomoku.py` contains the starter code for the Gomoku game and AI engine. The program works as follows. The computer (which plays with the `black` stones) always moves first. After the first move, the user's and the computer's moves alternate. The computer determines its move by finding the move that maximises the return value of the `score` function (which is provided).

Functions in `gomoku.py` accepts `board` as one of its arguments. This is the representation of the Gomoku board. The square `(y,x)` on the board is stored in `board[y][x]`. The value of the square is:

- `" "`, if the square is empty,
- `"b"`, if the square has a black stone on it, and
- `"w"`, if the square has a white stone on it.

See the function `printBoard` for an example of using `board`.

An important part of the Gomoku AI engine is finding contiguous sequences of stones of the same colour on the Gomoku board. There are four possible directions for a sequence: left-to-right, top-to-bottom, upper-left-to-lower-right, and upper-right-to-lower-left. Note that we do not consider, for example, the direction right-lo-left, since a right-to-left sequence can be represented as a left-to-right sequence. The direction of a sequence can be represented by a pair of numbers `(d_y, d_x)` as follows:

- `(0,1)`: direction left-to-right. For example, the sequence of stones of the same colour on coordinates `(5,2), (5,3), (5,4), (5,5)` is a sequence in direction left-to-right. Note that we say that the last stone in the sequence is at location `(5,5)`, not at location `(5,2)`.
- `(1,0)`: direction top-to-bottom. For example, a sequence of stones of the same colour on coordinates `(3,1), (4,1), (5,1)` is a top-to-bottom sequence. Note that we say that the last stone in the sequence is at location `(5,1)`, not at location `(3,1)`.
- `(1,1)`: direction upper-left-to-lower-right. For example, a sequence of stones of the same colour on coordinates `(2,3), (3,4), (4,5)` is an upper-left-to-lower-right sequence. Note that we say that the last stone in the sequence is at location `(4,5)`, not at location `(2,3)`.
- `(1,-1)`: direction upper-right-to-lower-left. For example, a sequence of stones of the same colour on coordinates `(5,5), (6,4), (7,3)` is an upper-right-to-lower-left sequence. Note that we say that the last stone in the sequence is at location `(7,3)`, not at location `(5,5)`.

A sequence can be:

- *open*: a stone can be put on a square at either side of the sequence.

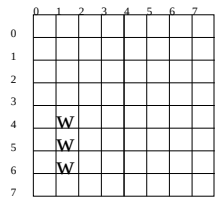

Figure 1: (1,0)

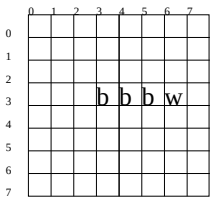

Figure 2: (0,1)

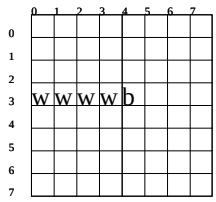

Figure 3: (0,1)

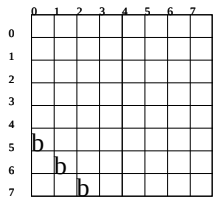

Figure 4: (1,1)

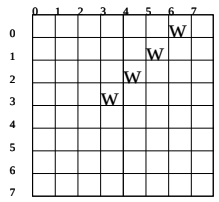

Figure 5: (1,-1)

- *closed*: the sequence is blocked on both sides, so that no stone can be placed on either side of the sequence. This can occur either because the sequence begins/ends near the border of the board, or because there is a stone of a different colour in the location immediately next to the beginning/end of the sequence.
- *semi-open*: the sequence is neither open nor closed.

Here are some examples of sequences and their classifications.

- Figure 1: An open sequence with 3 white stones. The direction is `(1,0)`. The last stone is at location `(6,1)`.
- Figure 2: A semi-open sequence with 3 black stones. The direction is `(0,1)`. The last stone is at location `(3,5)`.
- Figure 3: A closed sequence with 4 white stones. The direction is `(0,1)`. The last stone is at location `(3,3)`.
- Figure 4: A closed sequence with 3 black stones. The direction is `(1,1)`. The last stone is at location `(7,2)`.
- Figure 5: A semi-open sequence with 4 white stones. The direction is `(1,-1)`. The last stone is at location `(3,3)`.

The file `gomoku.py` contains implementations of the following functions:

- `print_board(board)`
  This function prints out the Gomoku board.
- `score(board)`
  This function computes and returns the score for the position of the board. It assumes that black has just moved.
- `play_gomoku(board_size)`
  This function allows the user to play against a computer on a board of size $board\_size \times board\_size$. This function interacts with the AI engine by calling the function `searchMax()`, which you will write.
- `put_seq_on_board(board, y, x, d_y, d_x, length, col)`
  This helper function adds the sequence of stones of colour `col` of length `length` to `board` , starting at location `(y,x)` and moving in the direction `(d_y, d_x)`. This function facilitates the testing of the AI engine.
- `analysis(board)`:
  This function analyses the position of the board by computing the number of open and semi-open sequences of both colours.

Do not modify these functions!

Part 1. The AI engine

Write an AI engine for the Gomoku game by implementing the following functions.

- `is_empty(board)`
  This function returns `True` iff there are no stones on the board `board`.
- `is_bounded(board, y_end, x_end, length, d_y, d_x)`

  This function analyses the sequence of length `length` that ends at location `(y_end, x_end)`. The function returns `"OPEN"` if the sequence is open, `"SEMIOPEN"` if the sequence if semi-open, and `"CLOSED"` if the sequence is closed.

  Assume that the sequence is complete (i.e., you are not just given a subsequence) and valid, and contains stones of only one colour.
- `detect_row(board, col, y_start, x_start, length, d_y, d_x)`
  This function analyses the row (let's call it **R**) of squares that starts at the location `(y_start,x_start)` and goes in the direction `(d_y,d_x)`. Note that this use of the word *row* is different from "a *row* in a table". Here the word *row* means a sequence of squares, which are adjacent either horizontally, or vertically, or diagonally. The function returns a tuple whose first element is the number of open sequences of colour `col` of length `length` in the row **R**, and whose second element is the number of semi-open sequences of colour `col` of length `length` in the row **R**.

  Assume that `(y_start,x_start)` is located on the edge of the board. Only *complete* sequences count. For example, column 1 in Fig. 1 is considered to contain one open row of length 3, and no other rows.

  Assume `length` is an integer greater or equal to 2.
- `detect_rows(board, col, length)`
  This function analyses the board `board`. The function returns a tuple, whose first element is the number of open sequences of colour `col` of length `length`**on the entire board**, and whose second element is the number of semi-open sequences of colour `col` of length `length` **on the entire board**.

  Only *complete* sequences count. For example, Fig. 1 is considered to contain one open row of length 3, and no other rows.

  Assume `length` is an integer greater or equal to 2.
- `search_max(board)`:
  This function uses the function `score()` (provided) to find the optimal move for black. It finds the location `(y,x)`, such that `(y,x)` is empty and putting a black stone on `(y,x)` maximizes the score of the board as calculated by `score()`. The function returns a tuple `(y, x)` such that putting a black stone in coordinates `(y, x)` maximizes the potential score (if there are several such tuples, you can return any one of them). After the function returns, the contents of `board` **must** remain the same.
- `is_win(board)` This function determines the current status of the game, and returns one of `["White won", "Black won", "Draw", "Continue playing"]`, depending on the current status on the board. The only situation where `"Draw"` is returned is when `board` is full.

Part 2.

Test your code thoroughly, and make sure that all your functions work as required by the spec. While you do not earn points directly for thoroughly testing your code, doing that is the only way to make sure your functions work.