CSCI-561 – Spring 2023 - Foundations of Artificial Intelligence
Homework 2

Due March 7, 2023 23:59:59

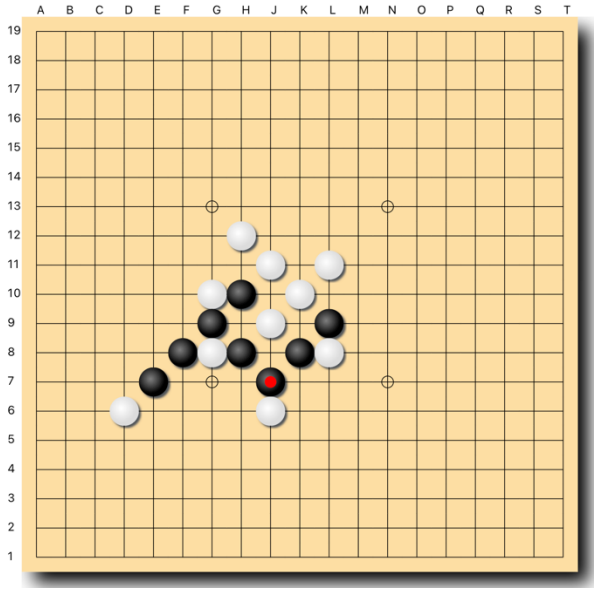


Image from Pente.org

Guidelines

This is a programming assignment. You will be provided sample inputs and outputs (see below). Please understand that the goal of the samples is only to check that you can correctly parse the problem definitions and generate a correctly formatted output that contains a valid, but not necessarily good (or the only possible) move. In most situations, several moves will be possible, so your move may differ from our example and still be perfectly valid. You should not assume that if your program works on the samples, it is a correctly implemented game-playing agent. It is your task to make sure that your program will work correctly on any valid input. You are encouraged to try your own test cases or have your program play against itself to check how your program would behave in some complex situations. Since **each homework submission is checked by a set of programs**, your output should match the specified format ***exactly***. Failure to do so will most certainly cost some points. The output format is simple and examples are provided. You should upload and test your code on vocareum.com, and you will submit it there. You may use any of the programming languages and versions thereof provided by vocareum.com.

Grading

Your code will be tested as follows: Your program should not require any command-line argument. It should read a text file called “input.txt” in the current directory that contains the current state of the game. It should write a file “output.txt” with your chosen move to the same

current directory. Format for input.txt and output.txt is specified below. End-of-line character is LF (since vocareum is a Unix system and follows the Unix convention).

Note that if your code does not compile, or somehow fails to load and parse input.txt, or writes an incorrectly formatted output.txt, or no output.txt at all, or OuTpUt.TxT, **you will get zero points**. Anything you write to stdout or stderr will be ignored and is ok to leave in the code you submit (but it will likely slow you down). Please test your program with the provided sample files to avoid any problem.

Academic Honesty and Integrity

All homework material is checked vigorously for dishonesty using several methods. All detected violations of academic honesty are forwarded to the Office of Student Judicial Affairs. To be safe you are urged to err on the side of caution. Do not copy work from another student or off the web. Keep in mind that sanctions for dishonesty are reflected in *your permanent record* and can negatively impact your future success. As a general guide:

> **Do not copy** code or written material from another student. Even single lines of code should not be copied.
> **Do not collaborate** on this assignment. The assignment is to be solved individually.
> **Do not copy** code off the web. This is easier to detect than you may think.
> **Do not share** any custom test cases you may create to check your program's behavior in more complex scenarios than the simplistic ones considered below.
> **Do not copy** code from past students. We keep copies of past work to check for this. Even though this problem differs from those of previous years, do not try to copy from homework submissions of previous years.
> **Do not ask on piazza** how to implement some function for this homework, or how to calculate something needed for this homework.
> **Do not post code on piazza** asking whether or not it is correct. This is a violation of academic integrity because it biases other students who may read your post.
> **Do not post test cases on piazza** asking for what the correct solution should be.
>
> **Do** ask the professor or TAs if you are unsure about whether certain actions constitute dishonesty. It is better to be safe than sorry.

Project description

In this project, we will implement agent that plays the game of ***Pente***, the two-player version of the abstract strategy board game. It is in the **m,n,k-game family** from which tic-tac-toe (3,3,3) and Connect Four also come from, where the players are trying to connect k pieces on the board before the other player does. Pente stands out from these games because it includes a piece capture mechanic, where the player can *sandwich* **a pair** of their opponent's pieces and capture them. It is played on a 19x19 board, where pieces are placed on the intersection of the

lines (like the game Go). White always opens the game (like in Chess). The players take turns putting pieces on the board until:

1) A player connects 5 of their pieces in a horizontal, vertical or a diagonal line, OR
2) A player makes 5 total captures (equals to 10 pieces of their opponent's since pieces can *only* be captured in pairs).

The custodial capture mechanic, where a player flanks the opponent's pieces with their own to capture them, only applies to pairs of the opponent's pieces. Therefore, if the current board formation is XOO_ and player X plays their piece as XOOX, the O pieces are captured and the board becomes X_ _X. Note again that this only works for pairs of pieces, therefore X cannot capture their opponent's pieces from a board like XOOO_ or XO_.

Captures only happen when a capturing piece is placed. Therefore, if the board is XO_X and O places a piece in the free intersection, the board formation becomes XOOX and no capture happens by X.

More details on the game can be found at https://en.wikipedia.org/wiki/Pente and we will also go over the gameplay for you below (be careful, Wikipedia images show a 13x13 board). If you would like to try the game of Pente to understand the general gameplay, you can utilize the "Play the AI" option without having to log in at https://pente.org/join.jsp. Note that these resources are just for you to familiarize yourself with the game, you should strictly follow the rules we outline below while coding your game-playing agent (there are several variants of the game).

The original Pente game is known to favor the first player. This is called the First Player Advantage (FPA) and there has been many suggestions to level the playing field for the second player. A discussion about this can be found in:
https://en.wikipedia.org/wiki/Pente#First_Player_Advantage
We will be employing certain rules that have been used in the past in Pente tournaments to make this game fairer for the second player. More details of these can be found below. These will also help us determine how winners are determined in student competitions.

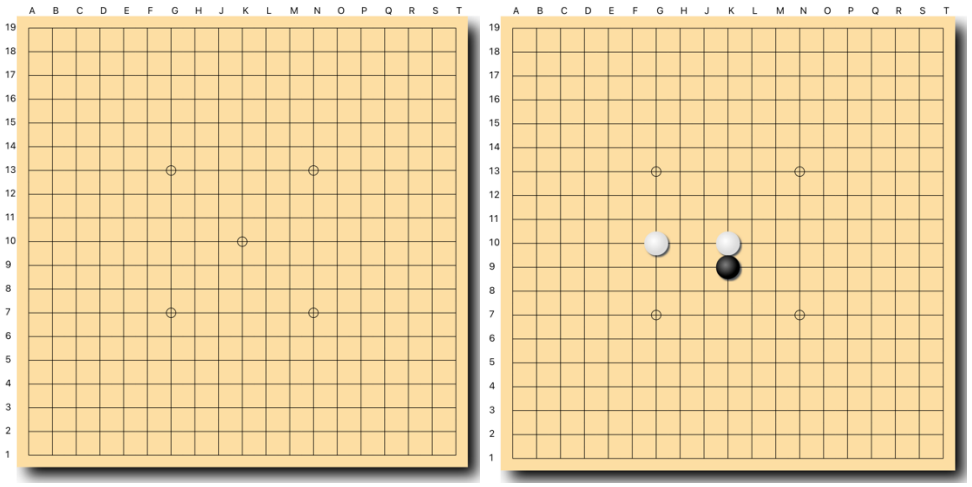
Setup of the game:

The setup of the game is as follows:

- Each player plays as white or black.
- The board consists of an 19x19 grid of squares.
- Before the game starts, the board is empty.
- White always opens the game.
- First piece (White) must be placed in the middle of the board.
- The second move of White (first player) cannot be less than 3 intersections away from their first piece (center). This is one of the rules that alleviates the first player advantage (FPA).

- Pieces can be placed in any empty intersection (apart from the first player restrictions explained above).
- Placed pieces cannot be removed from the board unless they are captured.

Here's the visualization of an empty board and an example start, just after White has placed their second piece following the anti-FPA restriction:

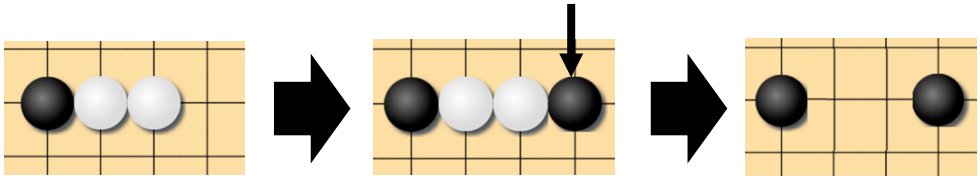


As can be seen in the right image above, White has placed their 2nd piece 3 intersections away which is the maximum possible amount for their second move. In following turns, pieces can be placed in any empty intersection, but your agent should be able to figure out which is best.

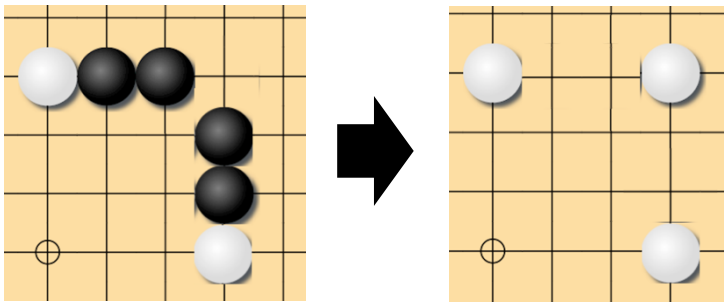
Play sequence:

We first describe the typical play for humans. We will then describe some minor modifications for how we will play this game with artificial agents.

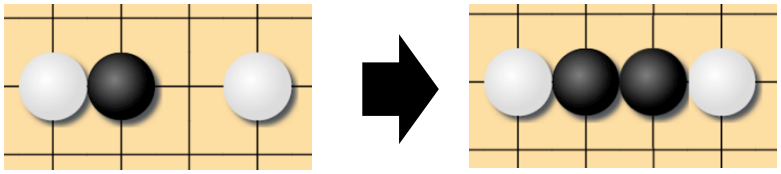
- Create the initial board setup according to the above description.
- Players randomly determine who will play White/Black. White will play first.
- During their turn, each player places a single piece of one's own color on the board:
    - Once placed, the pieces cannot be removed from their intersection unless they are captured by the opponent.
    - A simple move:
        - Can be played on any empty intersection (including border intersections).
        - If playing White, player must place their 1st piece in the center of the board and their 2nd piece no less than 3 intersections away from the 1st piece.
    - A capture move:
        - Is played next to a pair of the opponent's pieces such that they are flanked (sandwiched) by one of the player's pieces. The opponent's pieces are captured by the player when this happens. A sample capture can be seen below:



- The capture can be made for pieces that are horizontally, vertically, or diagonally oriented.
- When capture happens, the pair of pieces belonging to the opponent are removed from the board and added to the player's capture pile.
- One capture move can be used to capture one or more pairs of pieces from the opponent. Sample of this can be seen below:



- Note that captures only happen when both of the opponent's pieces are on the board when a capturing move is made. Therefore, if the board state is XO_X and O moves to make it XOOX, the pair of O pieces are not captured by X. An example for this is below.



- If the current play results in a board where the active player has 5+ connected pieces on the board OR has 5 pairs (or 10 pieces) total captured from their opponent, the game ends. Otherwise, play proceeds to the other player.
- If above conditions are not met, and there is no more room left on the board to play a piece, the game ends in a draw.

Playing with agents

In this homework, your agent will play against another agent, either implemented by the TAs, or by another student in the class.

For grading, your agent will play against two different agents implemented by the TAs. 10 full games will be against a random agent (this should be easy for your agent to beat), and another 10 full games will be against a simple minimax agent with no alpha-beta pruning. There will be a limited total amount of play time available to your agent for the whole game (e.g., 100 seconds), so you should think about how to best use it throughout the game. This total amount of time will vary from game to game. Your agent must play correctly (no illegal moves, etc.) and beat the

reference agents to receive 5 points per game. Your agent will be given the first move on 12 of the 20 games. In case of a draw, the agent with more remaining play time wins. Note that, while playing games, you should think about how to divide your remaining play time across possibly many moves throughout the game.

In addition to grading, we will run a competition where your agent plays against agents created by the other students in the class. This will not affect your grade, but it would look very good on your Resume if you finish in the top 10, or are the grand winner!

Agent vs agent games:

Playing against another agent will be organized as follows (both when your agent plays against the reference minimax agent, or against another student's agent):

A master game playing engine will be implemented by the grading team. This engine will:

- Create the initial board setup according to the above description.
- Assign a player color (Black or White) to your agent. The player who gets assigned White will have the first move.
- Then, in sequence, until the game is over:
    - The master game playing engine will create an input.txt file which contains the current board configuration, which color your agent should play, and how much total play time your agent has left. This file will also contain number of pieces captured by each agent until that point in the game. More details on the exact format of input.txt are given below.
    - We will then run your agent. Your agent should read input.txt in the current directory, decide on a move, and create an output.txt file that describes the move (details below). Your time will be measured (total CPU time). If your agent does not return before your time is over, it will be killed and it loses the game.
    - Your remaining playing time will be updated by subtracting the time taken by your agent on this move. If time left reaches zero or negative, your agent loses the game.
    - The validity of your move will be checked. If the format of output.txt is incorrect or your move is invalid according to the rules of the game, your agent loses the game. (Reminder: Any empty spot on the board is a valid move, except for the rules for the first two moves of the white player.)
    - Your move will be executed by the master game playing engine. This will update the game board to a new configuration.
    - The master game playing engine will check for a game-over condition. If one occurs, the winning agent or a draw will be declared accordingly.
    - The master game playing engine will then present the updated board to the opposing agent and let that agent make one move (with the same rules as just described for your agent; the only difference is that the opponent plays the other color and has its own time counter).
    - Game continues until an end condition is reached.

Input and output file formats:

Input: The file input.txt in the current directory of your program will be formatted as follows:

First line: A string BLACK or WHITE indicating which color you play. White will always start the game.

Second line: A strictly positive floating point number indicating the amount of play time remaining for your agent (in seconds).

Third line: Two non-negative 32-bit integers separated by a comma indicating the number of pieces captured by White and Black players consecutively. Caution, it will always be ordered as first captured by White, then by Black, irrespective of what color is given in the first line.

Next 19 lines: Description of the game board, with 19 lines of 19 symbols each:

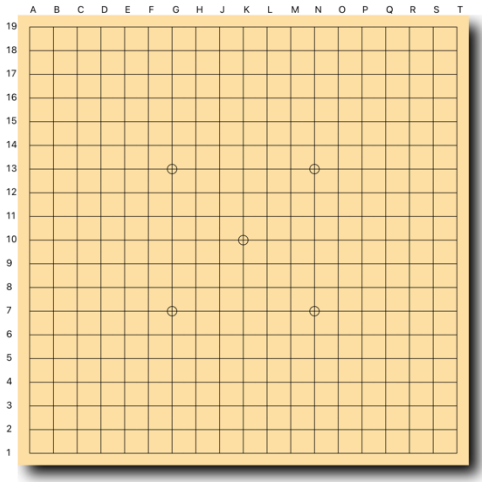
- `w` for a cell occupied by a white piece
- `b` for a cell occupied by a black piece
- `.` (a dot) for an empty intersection

For example:

```
BLACK
100.0
0,0
...................
...................
...................
...................
...................
...................
...................
.........w.........
..........b........
.........w.bw......
............*......
...................
...................
...................
...................
...................
...................
...................
...................
```

In this input.txt example, your agent should play a move as the Black agent and has 100.0 seconds. The board configuration is 5 turns into the game. There's a capture condition for White, so your agent could likely choose to block that by putting their piece in the red highlighted position on the board.

Output: The format we will use for describing the square positions is borrowed from the notations from Pente.org, where every column is described by a letter and every row is described by a number. The position for a given square is given as the concatenation of these. Here's a useful visualization on how we identify each intersection for the 19x19 Pente board:



Using the above image as reference, in the input.txt sample given above, White has pieces on 10K, 10N and 12K, while Black has pieces on 11L and 10M. Using this type of notation for the cells on our gameboard, the file output.txt which your program creates in the current directory should be formatted as follows:

1 line: **`PIECE_POS`** which describes your move with an integer (1-19) and an uppercase letter (A-T) concatenated.

For example, for the red highlighted move in the input sample, output.txt may contain:

```
9N
```

The resulting board would look like this, given the above input.txt (the master game playing engine will compute this and it is not part of output.txt):

```
...................
...................
...................
...................
...................
...................
...................
.........w.........
..........b........
.........w.bw......
............b......
...................
...................
...................
...................
...................
...................
...................
```

Notes and hints:

- Please name your program "**homework.xxx**" where 'xxx' is the extension for the programming language you choose ("py" for python, "cpp" for C++11, and "java" for Java).
- The board you will be given as input will always be valid and will have w and b letters, as well as **.** (standing for empty cells).
- Likely (but not guaranteed), total play time will be 5 minutes (300.0 seconds) when playing against another agent.
- Play time used on each move is the total combined CPU time as measured by the Unix **`time`** command. This command measures pure computation time used by your agent, and discards time taken by the operating system, disk I/O, program loading, etc. Beware that it cumulates time spent in any threads spawned by your agent (so if you run 4 threads and use 400% CPU for 10 seconds, this will count as using 40 seconds of allocated time).
- If your agent runs for more than its given play time (in input.txt), it will be killed and will lose the game.
- You need to think and strategize how to best use your allocated time. In particular, you need to decide on how deep to carry your search, on each move. In some cases, your agent might be given only a very short amount of time (e.g., 5.2 seconds, or even 0.01 seconds), for example towards the end of a game. Your agent should be prepared for that and return a quick decision to avoid losing by running over time. The amount of play time that will be given in input.txt will always be >0, but it could be very small if you are close to running out of time.
- To help you with figuring out the speed of the computer that your agent runs on, you are allowed to also provide a second program called **calibrate.xxx** (same extension conventions as for homework.xxx). This is optional. If one is present, **we will run your calibrate program once (and only once)** before we run your agent for grading or against another agent. You can use calibrate to, e.g., measure how long it takes to expand some fixed number of search nodes, or to benchmark the CPU speed in any other way you like. You can then save this into a single file called **calibration.txt** in the current directory. When your agent runs during grading or during a game, it could then read calibration.txt in addition to reading input.txt, and use the data from calibration.txt to strategize about search depth or other factors. Please aim for no more than 5 minutes to run your calibrate program. A few seconds (e.g., expand 10,000 nodes) is usually enough to get a good estimate of the CPU speed.
- You need to think hard about how to design **your eval function** (which gives a value to a board when it is not game over yet).
- You are allowed to maintain persistent data across moves during a game, by writing such data to a single file called **playdata.txt** in the current directory. Before a new game starts, the master game playing engine will delete any playdata.txt file. So, on your first move, this file will not exist, and you should be prepared for that. Then, you can write some data to that file at the end of a move and read that file back at the beginning of the next move.
- As mentioned, there is some first player advantage in this game for two agents with perfect play (even with our rules for the first two White moves). Therefore, when playing

against a reference agent, we will give your agent the first move for 6 of the 10 games. In the competition, we will play two agents against each other for an even number of games giving each the first player for half of the games, and advance both agents to the next round of the competition if they both win or draw on half of the games. If an agent loses more than half of the games, it will be eliminated and only the other agent will move to the next round of the competition. We may end up with several equivalent winners of the competition.

- The random agent created by the TAs will likely not be uniformly random over the whole board, but may choose randomly between several candidate locations, for example all locations adjacent to pieces already on the board. The minimax TA agent will not use alpha-beta and will likely be capped at a low lookahead depth; but it will do adaptive depth choice on every move to avoid running out of time (e.g., use depth 3 when >50s remains, depth 1 when < 3s, otherwise depth 2).