CSCI-561 - Spring 2024 - Foundations of Artificial Intelligence
Homework 2

Due March 6, 2024 23:59

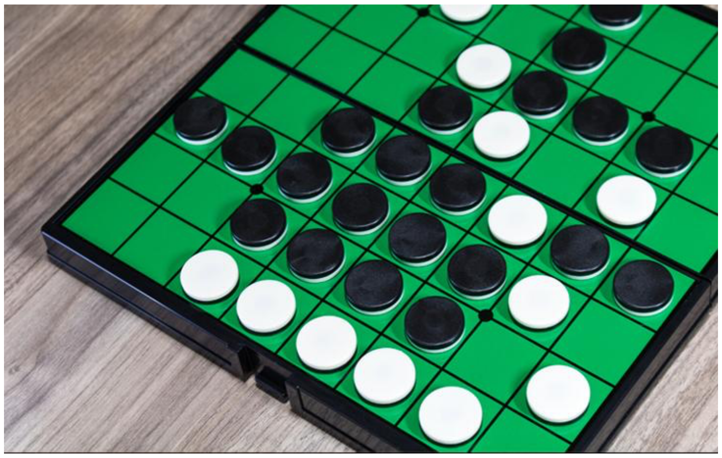


Image from https://www.coolmathgames.com/blog/how-to-play-reversi-basics-and-best-strategies

Guidelines

This is a programming assignment. You will be provided sample inputs and outputs (see below). The goal of the samples is to check that you can correctly parse the problem definitions and generate a correctly formatted output which is also a correct solution to the problem. For grading, your program will be tested on a *different* set of samples. It should not be assumed that if your program works on the samples it will work on all test cases, so you should focus on making sure that your algorithm is general and algorithmically correct. You are encouraged to try your own test cases to check how your program would behave in some corner cases that you might think of. Since **each homework is checked via an automated A.I. script**, 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 below. You should upload and test your code on vocareum.com, and you will also submit it there.

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, or runs out of memory or out of time (details below), **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 homeworks 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

The rules of the Reversi game can be found at http://en.wikipedia.org/wiki/Reversi [1] and interactive examples can be found at http://www.samsoft.org.uk/reversi/ [2]. In the Othello version of this game, the game begins with four disks placed in a square in the middle of the grid, two facing light-up, two pieces with the dark side up, with same-colored disks on a diagonal with each other. However, in this assignment, the starting position will be specified in the input file and will be different.

In this project, we define a new game that we call **Duo-Othello**. It is similar to the classic game of Reversi / Othello, which is explained below, but with three twists:

1) We play on a 12x12 board instead of the standard 8x8 Reversi board.
2) The start state has 8 pieces on the board in the exact configuration shown later below, instead of the standard 4 pieces.
3) Once game is over, which is when neither player can place new pieces on the board anymore, the winner is determined by counting the pieces of each color. To mitigate first-player advantage, the second player is given a bonus of +1. In case of ties (equal

scores for both players after bonus), the winner is the one with the most time remaining. If remaining times are equal, the winner is the player that started second.

Playing with agents, grading

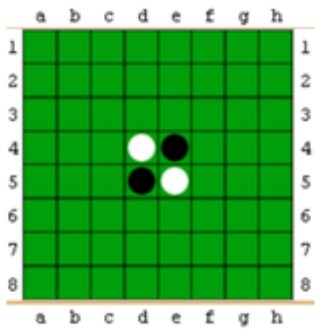
In this homework, your agent will play against another agent, either implemented by the TAs, or by another student in the class.

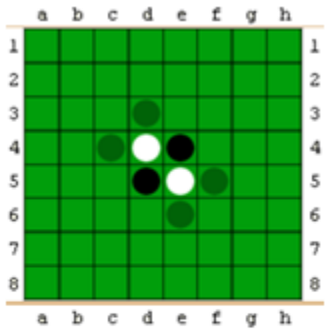
For grading, your agent will play against two different agents implemented by the TAs. 10 full games will be against a ***random agent*** (this should be easy for your agent to beat), and another 10 full games will be against a ***simple minimax agent*** with no alpha-beta pruning. There will be a limited total amount of play time available to your agent for the whole game (e.g., 300 seconds), so you should think about how to best use it throughout the game. This total amount of time may 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 10 of the 20 games. The conditions for determining a winner when the game is over are detailed below. 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, with a prize. 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!

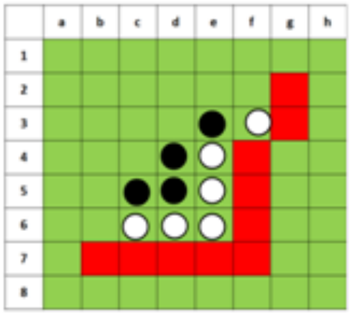
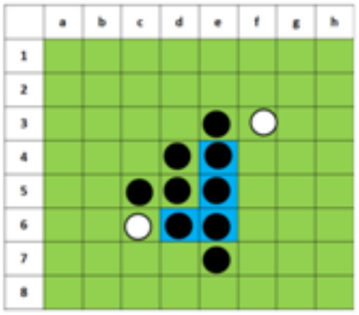
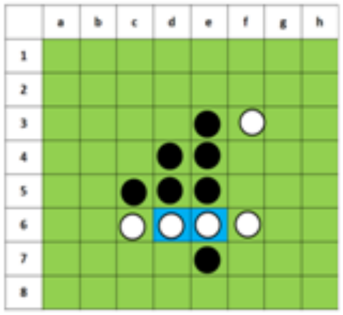
Standard Reversi / Othello game rules

Legal moves. Assume that the current position is as shown in the left image below and that the current turn is Black's. This only for you to understand legal moves. Remember that our board size and starting configuration is different in our game. Black must place a piece with the black side up on the board in such a position that there exists at least one straight (horizontal, vertical, or diagonal) occupied line between the new piece and another black piece, with one or more contiguous White pieces between them. The right image shows all the legal moves available to the Black player as translucent circles. **If one player cannot make a valid move, play passes back to the other player. Our game engine will automatically detect this and pass your turn without even running your agent, until it is your turn again.**

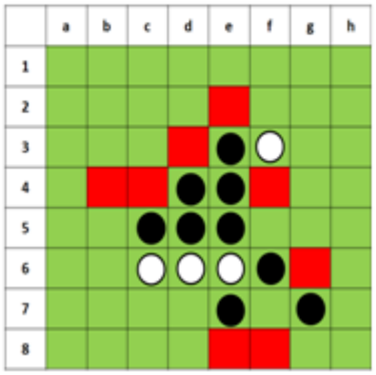
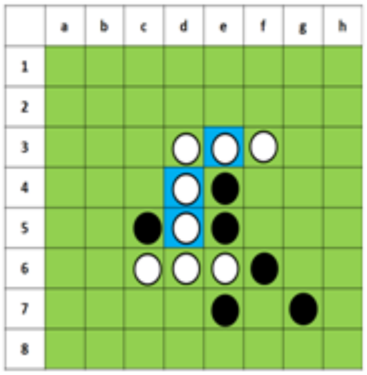
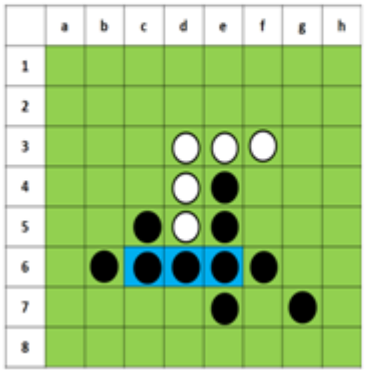




Example of Flips/Captures. In the left image below, the legal Black moves are shown as red cells. After placing the piece at e7, Black turns over (i.e., **flips** or **captures**) all White pieces that are on **any and all** straight line(s) between the newly added piece and any anchoring Black pieces [1]. All reversed pieces are shown in blue cells in the middle image for that Black move to e7. In the right image, White can reverse some of those pieces back on their turn if those Black pieces are on a straight line between the new White piece and any anchoring White pieces (in the image on the right, White played f6).

Another example of Flips/Captures. In the left image below, the available legal White moves are shown as red cells. After placing a new White piece at d3, all captured pieces are shown in blue cells in the center image. Black can use their own turn to reverse some pieces back if they choose so. However, in this example, in the right image, Black makes a different move, to b6, essentially choosing to flip other pieces.

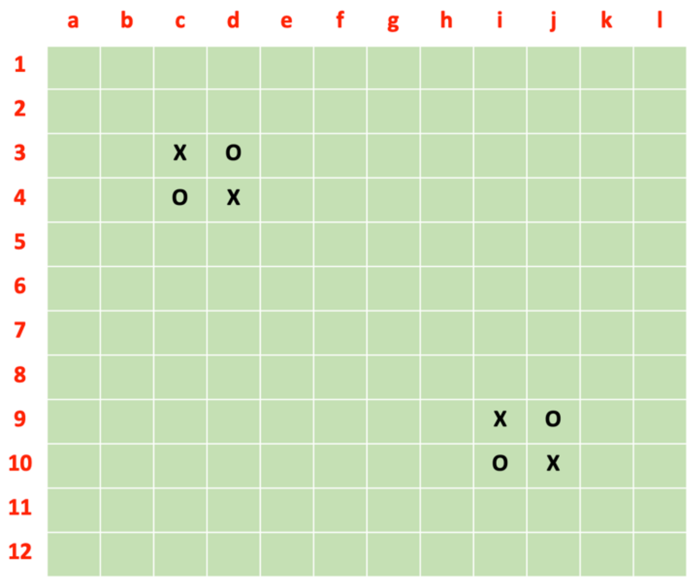




Pass Move and End Game. If one player cannot make a valid move, play **passes** back to the other player. When neither player can move, the game ends. This occurs when the grid has filled up or when neither player can legally place a piece in any of the remaining squares [1]. Once the game is over, how the winner is determined is described in details later below.

Game setup and initial state

The setup of the game is as follows:

- Each player plays as white (O) or black (X). This will alternate between different games.
- The board consists of a 12x12 grid of squares.
- Before the game starts, the board contains 8 pieces exactly as shown below.
- **White (O) always opens the game. Hence, Black (X) always gets the +1 bonus.**
- Each player starts the game with a given amount of time (e.g., 300 seconds). As we run an agent, we will measure the time taken on each move and subtract it from that agent's total remaining time. If agent runs out of time, it loses the game irrespective of the number of pieces on the board.



File formats

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

First line:	Player that your agent will play on this turn: **X** or **O.** During a given game, your agent will always play either X or O. But in the 20 games used for grading, your agent will play X 10 times, and O 10 times.
Second line:	Two floating point numbers separated by one space to indicate **your remaining play time in seconds**, and your **opponent's remaining play time**. The precision of these numbers is potentially the full prevision of double (e.g., 12.3456789).
Next 12 lines:	Current state of the board, one row per line. Each line will contain one string of 12 symbols (without any space in between) and an end-of-line (LF) marker, where:

- **X** denotes a cell occupied by player X
- **O** denotes a cell occupied by player O
- **.** denotes an empty cell.

For example:

```
X
300 286.46
............
............
..XO........
..OX........
............
............
............
........O...
........OO..
........OX..
............
............
```

In this example, your agent is assigned player X. You are eligible for the +1 bonus at the end of the game, as O is always given the first move (hence X always gets the +1 bonus). This example shows that O just played i8 as their first move after game start, resulting in the board state shown. You have 300.0 seconds of total play time remaining to complete the entire game, and your opponent O has 286.46 seconds left.

You are guaranteed that the format of input.txt will be correct (no missing lines, no missing grid cells, etc). You are also guaranteed that **at least one legal move is possible for your player**, i.e., you do not have to worry about pass. Just beware that sometimes the time remaining may show up as an integer if it just so happens that a player has an exact number of seconds of remaining time (as shown with the figure 300 in the file).

Output: The file output.txt which your program creates in the current directory should be formatted as follows:

First line: Your chosen move, specifying first the column from `a` to `l` (in lowercase), then row from 1 to 12, with no space.

For example, output.txt may contain:

```
c2
```

Game over test

When no more legal move is possible, the game ends, as follows:

- We will count the number of X and O pieces and add +1 bonus to the agent that started second (which is always X). If one score is strictly greater, the agent with the greater score wins;

- Otherwise, if both scores are equal, the agent with more remaining time wins;
- Otherwise, in the highly unlikely event that both remaining times are exactly equal, the player that started second (X) wins.

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 (X) or White (O), to your agent. The player who gets assigned White (O) will always have the first move and no bonus at game end, while X will always play second and get the +1 bonus at game end.
- 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, as described above.
    - 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, as shown above. Your time will be measured (total CPU time). If your agent does not return before your remaining time is over, it will be killed and it loses the game.
    - Your remaining playing time will be updated by the master game engine, 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.
    - 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 will be declared using the game over test described above.
    - 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.

Hence:

1) You do not have to worry about pass conditions. Your agent will not even be called if you cannot make at least one legal move.

2) You should not worry about updating your remaining time. The master game engine will measure the time taken by your agent on each move, and update your remaining time for you (details below).
3) You do not need to let us know what the state will be after your move. The master game engine will compute that, including all captures. Just let us know in output.txt what move you want to play, as described above.
4) You do not need to worry about game over conditions, the master game engine will do that.
5) The master game engine will not end the game unless no valid move exists. Thus, it is possible that your agent will be called many times in a row if the other agent each time has to pass. You should be prepared for that and not time out!

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++17, and "java" for Java).
- Likely (but not guaranteed), total play time for each agent 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). Hence, there is no benefit in using multi-threading in this homework, it will only slow you down because of various synchronization primitives (mutex, semaphore, etc).
- 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 seconds** to run your calibrate program. A few seconds is usually enough to get a good estimate of the CPU speed (e.g., expand 1,000 nodes, or compute the eval function on 1,000 hardwired board configurations).

- 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). Hint: it is well known in Othello that different grid locations have different desirability. You are welcome to search the web for any hint. Just beware that our game is not exactly Othello!
- 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.
- The random agent created by the TAs is not fully random: on every move, it randomly selects one of the available *legal* moves. So, you should not expect that it will lose just by selecting invalid moves. 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 remain, depth 1 when < 3s, otherwise depth 2).