CSCI-561 - Fall 2019 - Foundations of Artificial Intelligence
Homework 2

Due October 21, 2019 23:59:59

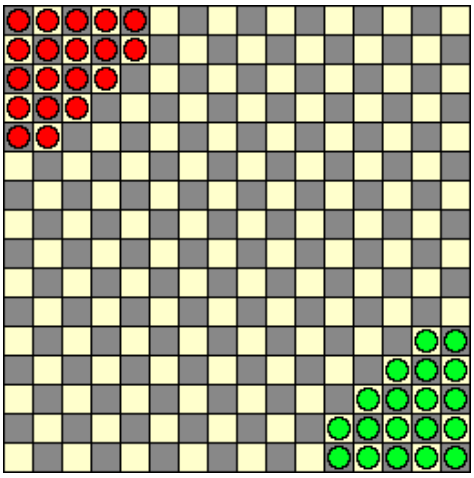


Image from cardboard_box @ stackexchange

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 to check that you can correctly parse the problem definitions and generate a correctly formatted output. The samples are very simple and it should not be assumed that if your program works on the samples it will work on all test cases. There will be more complex test cases and 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 to check how your program would behave in some complex special case 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. You should upload and test your code on vocareum.com, and you will submit it there. You may use any of the programming languages 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 a problem definition. It should write a file "output.txt" with your solution 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 cøpy** 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

In this project, we will play the game of ***Halma***, an adversarial game with some similarities to checkers. The game uses a 16x16 checkered gameboard. Each player starts with 19 game pieces clustered in diagonally opposite corners of the board. To win the game, a player needs to transfer all of their pieces from their starting corner to the opposite corner, into the positions that were initially occupied by the opponent. Note that this original rule of the game is subject to *spoiling*, as a player may choose to not move some pieces at all, thereby preventing the opponent from occupying those locations. Note that the spoiling player cannot win either (because some pieces remain in their original corner and thus cannot be used to occupy all positions in the opposite corner). Here, to prevent spoiling, we modify the goal of the game to be to occupy all of the opponent's starting positions which the opponent is not still occupying. See http://www.cyningstan.com/post/922/unspoiling-halma for more about this rule modification.

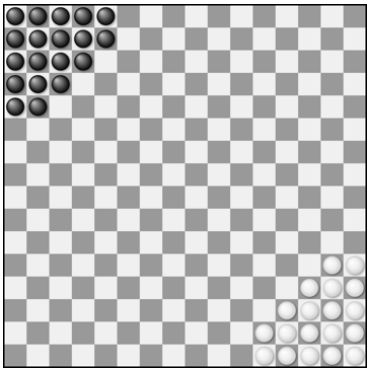
In more details (from https://en.wikipedia.org/wiki/Halma):

Setup for two players:

Note: we only consider the two-player variant here; this game can also be played by four players but we will not explore this here.

- Simple wooden pawn-style playing pieces, often called "Halma pawns."
- The board consists of a grid of 16×16 squares.
- Each player's camp consists of a cluster of adjacent squares in one corner of the board. These camps are delineated on the board.
- For two-player games, each player's camp is a cluster of 19 squares. The camps are in opposite corners.
- Each player has a set of pieces in a distinct color, of the same number as squares in each camp.
- The game starts with each player's camp filled by pieces of their own color.

The initial setup is shown below for black and white players. We will always use this exact initial setup in this homework.



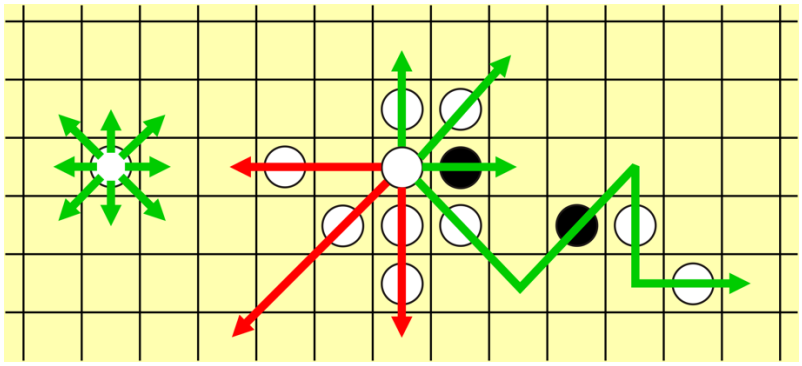
Play sequence:

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 move first.
- Pieces can move in eight possible directions (orthogonally and diagonally).
- Each player's turn consists of moving a single piece of one's own color in one of the following plays:
    - One move to an empty square:
        - Move the piece to an empty square that is adjacent to the piece's original position (with 8-adjacency).
        - This move ends the play for this player's turn.

    - One or more jumps over adjacent pieces:
        - An adjacent piece of any color can be jumped if there is an empty square on the directly opposite side of that piece.
        - Place the piece in the empty square on the opposite side of the jumped piece.
        - The piece that was jumped over is unaffected and remains on the board.
        - After any jump, one may make further jumps using the same piece, or end the play for this turn.
        - In a sequence of jumps, a piece may jump several times over the same other piece.
- Once a piece has reached the opposing camp, a play cannot result in that piece leaving the camp.
- If the current play results in having every square of the opposing camp that is not already occupied by the opponent to be occupied by one's own pieces, the acting player wins. Otherwise, play proceeds to the other player.

Below we show examples of valid moves (in green) and invalid moves (in red). At left, the isolated white piece can move to any of its empty 8 neighbors. At right, the central white piece can jump over one adjacent piece if there is an empty cell on the other side. After one jump is executed, possibly several other valid jumps can follow with the same piece and be combined in one move; this is shown in the sequence of jumps that start with a down-right jump for the central piece. Note that additional valid moves exist that are not shown (e.g., the central white piece could move to some adjacent empty location).



Note the invalid moves: red arrow going left: cannot jump over one or more empty spaces plus one or more pieces. Red arrow going left-down: cannot jump over one or more pieces plus one or more empty spaces. Red arrow going down: cannot jump over more than one piece.

Playing with agents

In this homework, your agent will play against another agent, either created by the TAs, or created by another student in the class. For grading, we will use two scenarios:

1) **Single move:** your agent will be given in input.txt a board configuration, a color to play, and some number of seconds of allowed time to play one move. Your agent should return

in output.txt the chosen move(s), before the given play time has expired. Play time is measured as total CPU time used by your agent on all CPU threads it may spawn (so, parallelizing your agent will not get you any free time). Your agent will play 10 single moves, each worth one point. If your agent returns an illegal move, a badly formatted output.txt, or does not return before its time is up, it will lose the point for that move.

2) **Play against reference agent:** your agent will then play 9 full games against a simple minimax agent with no alpha-beta pruning, created by the TAs. 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 minimax agent to receive 10 points per game. Your agent will be given the first move on 5 of the 9 games. In case of a draw, the agent with more remaining play time wins.

Note that we make a difference between single moves and playing full games because in single moves it is advisable to use all remaining play time for that move. While playing games, however, 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 grade. But the top agents will be referred to a contact at Google for an informal introduction. There will also be a prize for 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 agent will be implemented by the grading team. This agent will:

- Create the initial board setup according to the above description.
- Randomly assign a player color (black or white) to your agent.
- When playing against the reference minimax, you will get the opening move. Otherwise who plays first will be chosen randomly.
- Then, in sequence, until the game is over:
    - The master game playing agent will create an input.txt file which lets your agent know the current board configuration, which color your agent should play, and how much total play time your agent has left. 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.

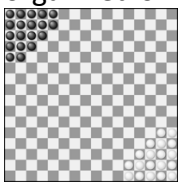
- Your playing time remaining will be updated by subtracting the time taken by your agent on this move. If time left reaches zero or negative, your agent loses the game.
- The validity of your move will be checked. If the format of output.txt is incorrect or your move is invalid according to the rules of the game, your agent loses the game.
- Your move will be executed by the master game playing agent. This will update the game board to a new configuration.
- The master game playing agent will check for a game-over condition. If so, the winning agent will be declared the winner of this game.
- The master game playing agent will then present the updated board to the opponent agent and let that agent make one move (with 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).

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 SINGLE or GAME to let you know whether you are playing a single move (and can use all of the available time for it) of playing a full game with potentially many moves (in which case you should strategically decide how to best allocate your time across moves).

Second line: A string BLACK or WHITE indicating which color you play. The colors will always be organized on the board as follows:



(black starts in the top-left corner and white in the bottom-right).

Third line: A strictly positive floating point number indicating the amount of total play time remaining for your agent.

Next 16 lines: Description of the game board, with 16 lines of 16 symbols each:

- `W` for a grid cell occupied by a white piece
- `B` for a grid cell occupied by a black piece
- `.` (a dot) for an empty grid cell

For example:

```
SINGLE
WHITE
100.0
BBBBB...........
BBBBB...........
BBBB............
BBB.............
BB..............
................
................
................
................
................
................
..............WW
.............WWW
............WWWW
...........WWWWW
...........WWWWW
```

In this example, your agent plays a single move as white color and has 100.0 seconds. The board configuration is just the one from the start of the game.

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

1 or more lines: Describing your move(s). There are two possible types of moves (see above):

> **`E FROM_X,FROM_Y TO_X,TO_Y`** – your agent moves one of your pieces from location FROM_X, FROM_Y to adjacent empty location TO_X, TO_Y. We will again use zero-based, horizontal-first, start at the top-left indexing in the board, as in homework 1. So, location 0,0 is the top-left corner of the board; location 15,0 of the top-right corner; location 0,15 is the bottom-left corner, and location 15,15 the bottom-right corner. As explained above, TO_X,TO_Y should be adjacent to FROM_X,FROM_Y (8-connected) and should be empty. If you make such a move, you can only make one per turn.
>
> **`J FROM_X,FROM_Y TO_X,TO_Y`** – your agent moves one of your pieces from location FROM_X,FROM_Y to empty location TO_X,TO_Y by jumping over a piece in between. You can make several such jumps using the same piece, as explained above, and should write out **one jump per line** in output.txt.

For example, output.txt may contain:

```
E 11,15 10,15
```

This moves the leftmost white piece on the bottom row of the board to the left. The resulting board would look like this, given the above input.txt:

```
BBBBB..........
BBBBB..........
BBBB...........
BBB............
BB.............
..............
..............
..............
..............
..............
..............
.............WW
............WWW
...........WWWW
..........WWWWW
.........W.WWWW
```

Or it could contain:

```
J 12,15 10,13
```

which is a jump along the left-up diagonal for a bottom piece. Note that here we have only one jump, but there could be several **(one per line in output.txt)** if the board configuration allows (see example 3 below). The board would look like this after this jump, given the above input.txt:

```
BBBBB..........
BBBBB..........
BBBB...........
BBB............
BB.............
..............
..............
..............
..............
..............
..............
.............WW
............WWW
.........W.WWWW
..........WWWWW
..........W.WWW
```

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++, and "java" for Java). If you are using C++11, then the name of your file should be "homework11.cpp" and if you are using python3 then the name of your file should be "homework3.py".
- The board you will be given as input will always have nineteen `W` letters, nineteen `B` letters, and the rest will be `.` (empty space).
- Likely (but not guaranteed), total play time will be 5 minutes (300.0 seconds) when playing against another agent, and 30.0 seconds for single moves.
- 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) + 10 seconds (grace period), it will be killed and will lose the single move or the game.
- The grace period is only so that we do not kill your agent prematurely, and you should not plan on using it. You should aim to write out your output before your allocated time has passed. The actual play time taken by your agent will be subtracted after your agent returns, and you will lose if your agent ends up exceeding its allocated play time.
- 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 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. There is no lower bound on the amount of play time that will be given in input.txt except that it will always be >0.
- 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. 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.
- 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 agent 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.

We expect that simple content in a format of your choice would be sufficient, but if you want to save/load complex data structures, have a look at things like **boost::serialization** or the C++11, header-only **cereal** library at https://github.com/USCiLab/cereal (written by our former students).

- Note that there is an advantage in this game for the player that gets the first move. We give your agent that advantage when playing against the reference minimax agent by giving your agent the first move for 5 of the 9 games. But, in the other 4 games, your agent should still be "smarter" than plain minimax and should still be able to win. In the competition, we will play two agents against each other for an even number of games, and advance both agents to the next round of the competition if they both win 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.

Example 1:

For this input.txt:

```
SINGLE
WHITE
100.0
WWWWW..........
WWWWW..........
WWW............
WWW.W..........
WW.............
...............
...............
...............
...............
...............
...............
..........B...BB
............BBB
............BBB
..........BBBBB
..........BBBBB
```

one possible correct output.txt is:

```
E 4,3 3,2
```

Example 2:

For this input.txt:

```
SINGLE
WHITE
6.6
WWWWW..........
WW.WW..........
WWWW...........
WWW.W..........
WW.............
..............
..............
..............
..............
..............
..............
...........B.BB
..........B.BBB
...........B.B
..........BBBBB
..........BBBBB
```

one possible correct output.txt is:

```
J 4,3 2,1
```

Example 3:

For this input.txt:

```
SINGLE
BLACK
23.33
WWWWW..........
W.W.W..........
WWW............
WW...W.........
WW....W........
...............
.......W.......
...............
...............
.........BW.....
...........B....
............BB
...........BBBB
...........B.BB
............BBB
...........BBBBB
```

one possible correct output.txt is:

```
J 9,9 11,9
J 11,9 11,11
J 11,11 13,13
```