CSCI-561 - Fall 2017 - Foundations of Artificial Intelligence
Homework 2

Due October 17, 2017 23:59:59

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).

Additional details are provided below under **Grading**.

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 this.

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.
>
> **Do** ask the professor or TA if you are unsure about whether certain actions constitute dishonesty. It is better to be safe than sorry.

Project description

This project will provide you an opportunity to practice what you have learned about Game Playing in the class. In a typical zero sum two player game, players are generally competing for a certain common resource, and their gain is a function of their share of the resource. Often players have other challenges such as satisfying other constraints on other personal resources such as time, energy or computational power in the course of the game. In this homework we will introduce ***The Fruit Rage!*** a game that captures the nature of a zero sum two player game with strict limitation on allocated time for reasoning.

Your task is creating a software agent that can play this game against a human or another agent.

Rules of the game

The Fruit Rage is a two player game in which each player tries to maximize his/her share from a batch of fruits randomly placed in a box. The box is divided into cells and each cell is either empty or filled with one fruit of a specific type.

At the beginning of each game, all cells are filled with fruits. Players play in turn and can pick a cell of the box in their own turn and claim all fruit of the same type, in all cells that are connected to the selected cell through horizontal and vertical paths. For each selection or move the agent is rewarded a numeric value which is the square of the number of fruits claimed in that move. Once an agent picks the fruits from the cells, their empty place will be filled with other fruits on top of them (which fall down due to gravity), if any. In this game, no fruit is added during game play. Hence, players play until all fruits have been claimed.

Another big constraint of this game is that every agent has a limited amount of time to spend for thinking during the whole game. Spending more than the original allocated time will be penalized harshly. Each player is allocated a fixed total amount of time. When it is your turn to play, you will also be told how much remaining time you have. The time you take on each move will be subtracted from your total remaining time. If your remaining time reaches zero, your agent will automatically lose the game. Hence you should think about strategies for best use of your time (spend a lot of time on early moves, or on later moves?)

The overall score of each player is the sum of rewards gained for every turn. The game will terminate when there is no fruit left in the box or when a player has run out of time.

Game setup and examples

Figure 1 depicts a sample 10 x 10 game board with 4 types of fruits denoted by digits 0, 1, 2 and 3 in the cells. By analyzing the game, your agent should decide which location to pick next. Let's assume that it has decided to pick the cell highlighted in red and yellow in figure 1.

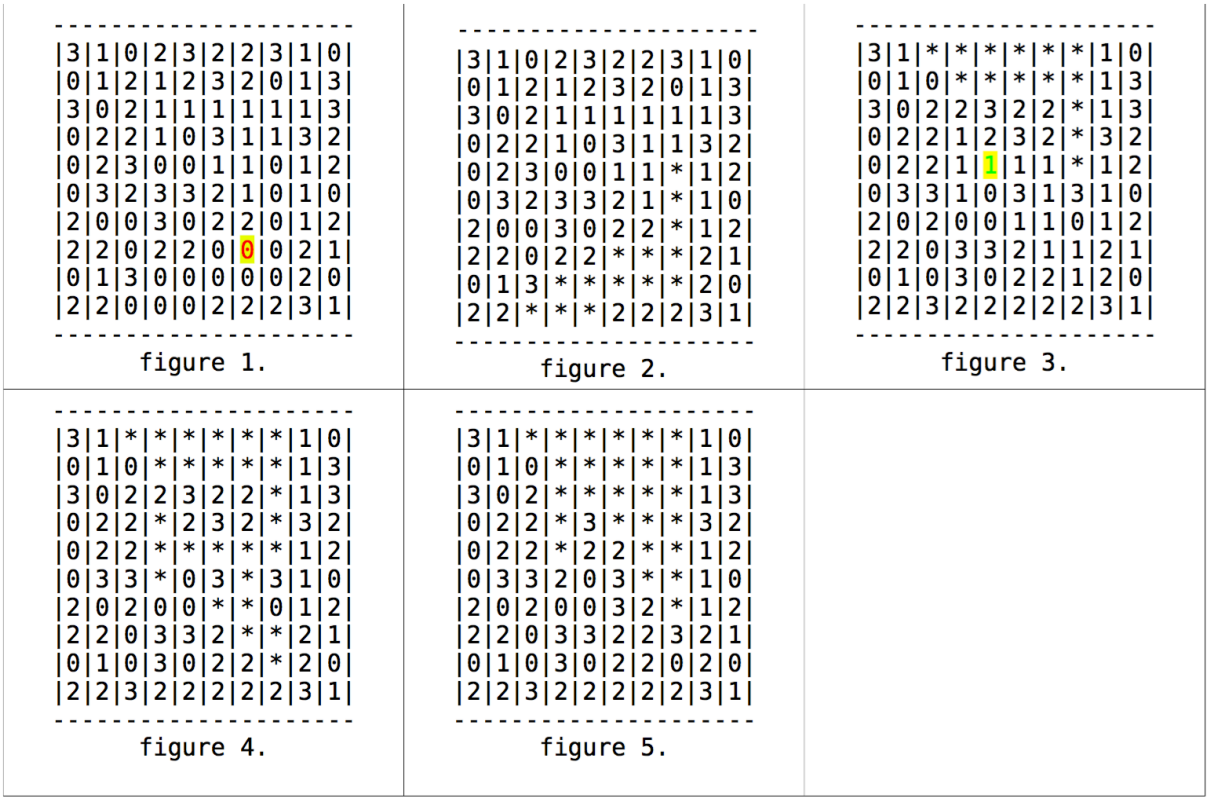

Figure 2 shows the result of executing this action: all the horizontally and vertically connected fruits of the same type (here, the selected fruit is of type 0) have been replaced by a * symbol (which represents an empty cell). The player will claim 14 fruits of type 0 because of this move and thus will be rewarded 14^2 = 196 points.

Figure 3 shows the state of the game after the empty space is filled with fruits falling from cells above. That is, for each cell with a * in figure 2, if fruits are present above, they will fall down. When a fruit that was on the top row falls down, its previous location is marked as empty (i.e., it becomes a * symbol). That is, no new fruits are injected to the top of the board. In addition to returning the column and row of your selected fruit, your agent will also need to return this resulting state after gravity has been applied. The game is over when all cells are empty, and the winner is determined by the total number of points, that is, sum of [fruits taken on each move]^2 (it is possible to end in a draw if both players score the same).

In figure 3, the opponent player then decided to pick the location highlighted in green and yellow. Upon selecting this cell, all 12 fruits of type 1 connected to that cell will be given to the opponent player and thus the opponent player will gain 12^2 = 144 points. In figure 4, cells connected to the selected cell are marked with * and in figure 5 you see how some of those picked fruits are replaced with the contents of cells above (fruits above fell down due to gravity).

To succeed, you should implement the **minimax algorithm with alpha-beta pruning**, as studied in class. While implementing minimax only (no pruning) might work in some cases, your agent is highly likely to run out of time for more complex cases unless you also implement alpha-beta pruning.

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

First line:	integer n, the width and height of the square board ($0 < n \leq 26$)
Second line:	integer p, the number of fruit types ($0 < p \leq 9$)
Third line:	strictly positive floating point number, your remaining time in seconds
Next n lines:	the n x n board, with one board row per input file line, and n characters (plus end-of-line marker) on each line. Each character can be either a digit from 0 to p-1, or a * to denote an empty cell. Note: for ease of parsing, the extra horizontal and vertical lines shown in figures 1 – 5 will not be present in the actual input.txt (see below for examples).

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

First line:	your selected move, represented as two characters: **A letter** from A to Z representing the column number (where A is the leftmost column, B is the next one to the right, etc), and **A number** from 1 to 26 representing the row number (where 1 is the top row, 2 is the row below it, etc).
Next n lines:	the n x n board just after your move and after gravity has been applied to make any fruits fall into holes created by your move taking away some fruits (like in figure 3).

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".
- Any additional heuristics on top of minimax + alpha-beta pruning which you can think of are allowed.
- The board you will be given as input could be completely filled with fruits, partially filled with fruits, but it will never be empty (all * symbols). Hence there will always be a valid possible move for your agent and you do not need to worry about passing your turn or detecting when the game is over.
- But note that the game board may become empty at some point during your search. You should then get the winner (or draw) by the number of points collected by each player.
- The board you will be given as input will always have gravity already applied (i.e., there cannot be a fruit above an empty space).
- Likely, total play time will be 5 minutes (300.0 seconds).
- 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 test case or the game (see below).
- The grace period is only so that we do not kill your agent prematurely, and you should not count on 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 considered and you will lose if it exceeds your 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), for example towards the end of a competition run (see below). Your agent should be prepared for that and return a quick decision to avoid losing by running over 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. 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, 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).

Grading

To discourage random agents and to motivate you to write the best possible agent, grading will be as follows:

- 25 test cases (similar approach to homework 1), but one quickly discovers that even a very bad agent can pass those by just issuing legal (though sub-optimal) moves and by correctly computing the next state (including applying gravity to make fruits fall into holes). This will hence only count for **25% of the grade** (1% per correct test case).
- Your agent will play against two reference agents implemented by the grading team:
    - **Random agent**: selects a random legal move on each turn.
    - **Minimax agent with no alpha-beta and lookahead depth limited to 3**: runs only plain minimax and does not search very deep. It will search even shallower if running out of time, but it will not lose because of having completely run out of time (see below for how your agent will play against another).
- On 11 test games, if your agent wins a majority against our random agent, you get **25%**
- On 11 test games, if your agent wins a majority against our minimax agent, you get **50%**

The criteria for winning a game, and how the game will be played, are detailed below.

If time allows, we will also run every agent in the class against every other agent and determine the best agents, in a competition. There will be a prize, to be discussed in class.

Running one agent against another

This will be run as follows, to the best of 11 games for each pair of agents:

- Decide randomly who plays first on the first of the 11 games between agent 1 and agent 2. On subsequent games, alternate who plays first.

- Create a random initial board. Initialize play times to 5 minutes (estimated, could be changed later) for agent 1 and 5 minutes for agent 2. Run calibrate (if provided) on agent 1 and also (if provided) on agent 2.

- Loop until game over:
    - Run agent 1 on it
    - Subtract the time taken by agent 1 from its total time. If zero or below is reached, agent 1 loses this game. Otherwise, check that the move returned by agent 1 is valid. If not, agent 1 loses. Otherwise, apply the move of agent 1, and apply gravity to the board to make any fruits fall down into empty cells.
    - If the board is empty (no more fruits), go to end of game (see below).
    - Now run agent 2 on that new board
    - Subtract the time taken by agent 2 from its total time. If zero or below is reached, agent 2 loses. Otherwise, check that the move returned by agent 2 is valid. If not,

agent 2 loses. Otherwise, apply the move of agent 2, and apply gravity to the board to make any fruits fall down into empty cells.

- If the board is empty (no more fruits), go to end of game (see below), otherwise go back to agent 1's turn to play.

End of game: when the board is empty (no more fruits), the game is over. The score of each agent is the total points accumulated during the whole game (the sum of [fruits taken on each move]^2). The agent with the highest score wins. If both agents scored the same, the one with the most remaining time wins (to avoid draws during the competition). If remaining times are also exactly equal (highly unlikely), discard this run and do a new one with a different initial condition.

Example 1:

For this input.txt:

```
2
3
123.6
01
21
```

one possible correct output.txt is:

```
B1
0*
2*
```

(remember: B1 means second column, top row, i.e., the agent picked the top right corner cell and got 2 fruits of type 1. It received 2^2 = 4 points for that move).

Example 2:

For this input.txt:

```
3
1
257.2
***
***
*0*
```

one possible correct output.txt is:

```
B3
***
***
***
```

Example 3:

For this input.txt:

```
3
2
24.345
***
*10
000
```

one possible correct output.txt is:

```
C2
***
***
*1*
```

Example 4:

For this input.txt:

```
3
10
7.24
444
444
444
```

one possible correct output.txt is:

```
A1
***
***
***
```

Example 5:

For this input.txt (same layout as in Figure 1):

```
10
4
1.276
3102322310
0121232013
3021111113
0221031132
0230011012
0323321010
2003022012
2202200021
0130000020
2200022231
```

one possible correct output.txt is (same move as in Figure 1, and same resulting layout as in Figure 3):

```
G8
31******10
010*****13
3022322*13
0221232*32
0221111*12
0331031310
2020011012
2203321121
0103022120
2232222231
```