B551 Assignment 2: Games and Bayes

Fall 2019
Due: Thursday November 7, 11:59PM
(You may submit up to 48 hours late for a 10% penalty.)

This assignment will give you practice with adversarial games and with Bayes Nets.

Guidelines for this assignment

Coding requirements. For fairness and efficiency, we use a semi-automatic program to grade your submissions. This means you must write your code carefully so that our program can run your code and understand its output properly. In particular:

1. You must code this assignment in Python 3, not Python 2.
2. Make sure to include a #! line at the top of your code.
3. You should test your code on one of the SICE Linux systems such as burrow.sice.indiana.edu.
4. Your code must obey the input and output specifications given below.
5. You may import standard Python modules for routines not related to AI, such as basic sorting algorithms and data structures like queues, as long as they are already installed on the SICE Linux servers.
6. Make sure to use the program file name we specify.

These requirements were discussed in more detail in the Assignment 0 handout.

For each of these problems, you will face some design decisions along the way. Your primary goal is to write clear code that finds the correct solution in a reasonable amount of time. To do this, you should give careful thought to the search abstractions, data structures, algorithms, heuristic functions, etc. that you use. To encourage innovation, we will conduct a competition among programs to see which can solve the hardest problems in the shortest amount of time.

Submissions. Unlike previous assignments, we won't be running an automated output tester on your programs. For Part 1, our skeleton code already handles output for you. For Part 2, we will provide test code to make sure that your program satisfies our formatting requirements (which you can run on your own).

Groups. You'll work in a group of 1-3 people for this assignment; we've already assigned you to a group (see details below) according to your preferences. You should only submit **one** copy of the assignment for your team, through GitHub, as described below. All the people on the team will receive the same grade on the assignment, except in unusual circumstances; we will collect feedback about how well your team functioned in order to detect these circumstances. The requirements for the assignment are the same whether you have 1 person, 2 people, or 3 people on your team, but we expect that teams with more people will submit answers that are more "polished" — e.g., better documented code, faster running times, more thorough answers to questions, etc.

Coding style and documentation. We will not explicitly grade based on coding style, but it's important that you write your code in a way that we can easily understand it. Please use descriptive variable and function names, and use comments when needed to help us understand code that is not obvious.

Report. Please put a report describing your assignment in the Readme.md file in your Github repository. For each problem, please include: (1) a description of how you formulated each problem; (2) a brief description of how your program works; (3) and discussion of any problems you faced, any assumptions, simplifications,

and/or design decisions you made. These comments are especially important if your code does not work as well as you would like, since it is a chance to document how much energy and thought you put into your solution.

Academic integrity. We take academic integrity very seriously. To maintain fairness to all students in the class and integrity of our grading system, we will prosecute any academic integrity violations that we discover. *Before beginning this assignment, make sure you are familiar with the Academic Integrity policy of the course, as stated in the Syllabus, and ask us about any doubts or questions you may have.* To briefly summarize, you may discuss the assignment with other people at a high level, e.g. discussing general strategies to solve the problem, talking about Python syntax and features, etc. You may also consult printed and/or online references, including books, tutorials, etc., but you must cite these materials (e.g. in source code comments). We expect that you'll write your own code and not copy anything from anyone else, including online resources. *However, if you do copy something (e.g., a small bit of code that you think is particularly clever), you have to make it explicitly clear which parts were copied and which parts were your own. You can do this by putting a very detailed comment in your code, marking the line above which the copying began, and the line below which the copying ended, and a reference to the source.* Any code that is not marked in this way must be your own, which you personally designed and wrote. You may not share written answers or code with any other students, nor may you possess code written by another student, either in whole or in part, regardless of format.

Part 0: Getting started

For this project, we are assigning you to a team. We will let you change these teams in future assignments. You can find your assigned teammate(s) by logging into IU Github, at `http://github.iu.edu/`. In the upper left hand corner of the screen, you should see a pull-down menu. Select `cs-b551-fa2019`. Then in the box below, you should see a repository called *userid1*-a2, *userid1-userid2*-a2, or *userid1-userid2-userid3*-a2, where the other user ID(s) correspond to your teammate(s). Now that you know their userid(s), you can write them an email at userid@iu.edu.

To get started, clone the github repository:

`git clone git@github.iu.edu:cs-b551-fa2019/`*`your-repo-name`*`-a2`

If that doesn't work, instead try:

`git clone https://github.iu.edu/cs-b551-fa2019/`*`your-repo-name`*`-a2`

where *your-repo-name* is the one you found on the GitHub website above. (If neither command works, you probably need to set up IU GitHub ssh keys. See Canvas for help.)

Part 1: IJK

IJK is a sliding tile game played on a 4x4 board by two players, lowercase (also known as `-`) and uppercase (`+`). The players take turns, starting with uppercase. At the start of each turn, a tile labeled `a` is added randomly somewhere on the board if it is lowercase's turn, and an `A` is added if it is uppercase's turn. A player's turn consists of making one of five possible moves: Up, Down, Left, Right, or Skip. Skip means nothing further happens and the player forfeits their turn. Otherwise, all tiles are slid in the direction that the player selected until there are no empty spaces left in that direction. Then, pairs of letters that are the same, of the same case, and adjacent to each other in the direction of the movement are combined together to create a single tile of the subsequent letter and the same case. For example, if an `A` slides down and hits another `A`, the two merge and become a single `B`. Similarly, pairs of `B`'s merge to become a single `C`, and so on. For pairs of letters that are the same but with opposite case, e.g. `A` and `a`, the tiles merge to become the letter `B` if it is uppercase's turn, or `b` if it is lowercase's turn. The game ends when a `K` or `k` appears on

the board, and the player of that case wins the game.

(If you're confused, never fear. Our skeleton code below will let you get experience with the game.)

The game has two specific variants. In *Deterministic IJK*, the new `a` or `A` at the beginning of each turn is added to a predictable empty position on the board (which is a function of the current board state). In *Non-deterministic IJK*, the new `a` or `A` is added to a random position on the board. The rest of the rules are the same for the two variants.

Your goal is to write AI clode that plays these two variants of IJK well. We've prepared skeleton code to help you. You can run the code like this:

```
./ijk.py [uppercase-player] [lowercase-player] [mode]
```

where *uppercase-player* and *lowercase-player* are each either `ai` or `human`, and mode is either `det` for deterministic IJK or `nondet` for nondeterministic IJK. These commands let you set up various types of games. For example, if you (a human) want to play a deterministic game against the ai, you could type:

```
./ijk.py human ai det
```

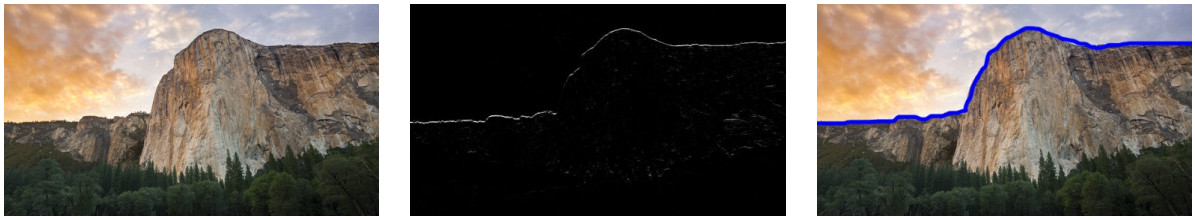
Similarly, you can set up games between AIs or between humans. Currently, the AI logic is very simple – it just plays randomly. You'll want to edit ai_IJK.py to implement more advanced logic. You shouldn't need to edit the other files, although you'll probably want to look at them to understand how the code works.

Hints. You'll probably want to start by implementing the deterministic version first, but it's up to you. Note that you don't have to implement the game itself (or even to fully understand the rules) since our code already does this for you.

The tournament. To make things more interesting, we will hold a competition among all submitted solutions to see whose solution can score the highest across several different games. While the majority of your grade will be on correctness, programming style, quality of answers given in comments, etc., a small portion may be based on how well your code performs in the tournaments, with particularly well-performing programs eligible for prizes including extra credit points.

Part 2: Horizon finding

A classic problem in computer vision is to identify where on Earth a photo was taken using visual features alone (e.g., not using GPS). For some images, this is relatively easy — a photo with the Eiffel tower in it was probably taken in Paris (or Las Vegas, or Disneyworld, or ...). But what about photos like Fig 2a? One way of trying to geolocate such photos is by extracting the horizon (the boundary between the sky and the mountains) and using this as a "fingerprint" that can be matched with a digital elevation map to identify where the photo was taken.



(a) (b) (c)

Figure 1: (a) Sample input image, (b) corresponding edge strength map, and (c) highlighted ridge.

Let's consider the problem of identifying horizons in images. We'll assume relatively clean images like that of Figure 2a, where the mountain is plainly visible, there are no other objects obstructing the mountain's ridgeline, the mountain takes up the full horizontal dimension of the image, and the sky is relatively clear. Under these assumptions, for each column of the image we need to estimate the row of the image corresponding to the boundary position. Plotting this estimated row for each column will give a horizon estimate.

We've given you some code that reads in an image file and produces an "edge strength map" that measures how strong the image gradient (local constrast) is at each point. We could assume that the stronger the image gradient, the higher the chance that the pixel lies along the ridgeline. So for an $m \times n$ image, this is a 2-d function that we'll call $I(x, y)$, measuring the strength at each pixel $(x, y) \in [1, m] \times [1, n]$ in the original image. Your goal is to estimate, for each column $x \in [1, m]$, the row s_x corresponding to the ridgeline. We can solve this problem using a Bayes net, where $s_1, s_2, ...s_m$ correspond to the hidden variables, and the gradient data are the observed variables (specifically $w_1, w_2, ...w_m$, where w_1 is a vector corresponding to column 1 of the gradient image).

1. Perhaps the simplest approach would be to use the Bayes Net in Figure 2b. You'll want to estimate the most-probable row s_x^* for each column $x \in [1, m]$,

$$s_i^* = \arg\max_{s_i} P(S_i = s_i | w_1, ..., w_m).$$

   Show the result of the detected boundary with a blue line superimposed on the image in an output file called `output_simple.jpg`.

   Hint: This is easy; if you don't see why, try running Variable Elimination by hand on the Bayes Net in Figure 2b.

2. A better approach would use the Bayes Net in Figure 1a. Use the Viterbi algorithm to solve for the maximum a posterior estimate,

$$\arg \max_{s_1,...,s_m} P(S_1 = s_1, ..., S_m = s_m | w_1, ..., w_m),$$

   and show the result of the detected boundary with a red line superimposed on the input image in an output file called `output_map.jpg`.

3. A simple way of improving the results further would be to incorporate some human feedback in the process. Assume that a human gives you a single (x, y) point that is known to lie on the ridgeline. Modify your answer to step 2 to incorporate this additional information. Show the detected boundary in green superimposed on the input image in an output file called `output_human.jpg`. Hint: you can do this by just tweaking the HMM's probability distributions – no need to change the algorithm itself.

Your program should be run like this:

```
./horizon.py input_file.jpg row_coord col_coord
```

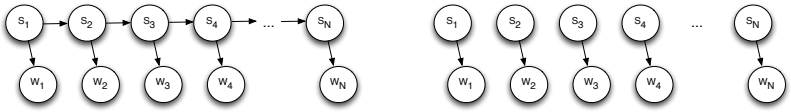


Figure 2: Two Bayes Nets.

where *row_coord* and *col_coord* are the image row and column coordinates of the human-labeled pixel.

Run your code on our sample images (and feel free to try other sample images of your own) and include some examples in your report.

Hints. What should the emission and transition probabilities be? There's no right answer here, but intuitively you'll want the emission probabilities $P(w_i|S_i = s_i)$ to be chosen such that they are high when s_i is near a strong edge according to the gradient image, and low otherwise. The idea behind the transition probabilities is to encourage "smoothness" from column to column, so that the horizon line doesn't randomly jump around. In other words, you'll want to define $P(S_{i+1} = s_{i+1}|S_i = s_i)$ so that it's high if $s_{i+1} \approx s_i$ and is low if not.

What to turn in

Turn in the files required above by simply putting the finished version (of the code with comments and PDF file for the first question) on GitHub (remember to `add`, `commit`, `push`) — we'll grade whatever version you've put there as of 11:59PM on the due date. To make sure that the latest version of your work has been accepted by GitHub, you can log into the github.iu.edu website and browse the code online.