CS B551 - Assignment 2: Games

Fall 2020

Due: Sunday November 1, 11:59:59PM Eastern (New York) time

This assignment will give you practice with game playing and probability. Please read the instructions below carefully; we cannot accept any submissions that do not follow the instructions given here. Most importantly: please **start early,** and ask questions on Piazza or in office hours.

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. You should test your code on one of the SICE Linux systems such as burrow.sice.indiana.edu. 3. Your code must obey the input and output specifications given below. 4. 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. 5. Make sure to use the program file name we specify.

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. To encourage innovation, we will conduct a competition to see which program can solve the hardest problems in the shortest time, and a small amount of your grade (or extra credit) may be based on your program's performance in this competition.

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, 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 no matter how many teammates you have, but we expect that teams with more people will submit answers that are significantly 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-fa2020`. 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-fa2020/your-repo-name-a2
```

If that doesn't work, instead try:

```
git clone https://github.iu.edu/cs-b551-fa2020/your-repo-name-a2
```

where *your-repo-name* is the one you found on the GitHub website above.

Part 1: The Game of Sebastian

Sebastian[1] is one-player game of luck and skill. Each turn has four steps:

1. The player rolls five dice.
2. The player inspects the dice and chooses any subset (including none or all) and rolls them.
3. The player inspects the dice again again choose any subset and rolls them.
4. The player must assign the outcome to exactly one category on their score card, depending on which five dice are showing after the third roll.

Here are the categories on the score card:

- Primis: The player can add the number of dice that show 1 to his or her score.
- Secundus: The player can count the number of dice that show 2, multiply by 2, and add to the score.
- Tertium: The player can count the number of dice that show 3, multiply by 3, and add to the score.
- Quartus: The player can count the number of dice that show 4, multiply by 4, and add to the score.
- Quintus: The player can count the number of dice that show 5, multiply by 5, and add to the score.
- Sextus: The player can count the number of dice that show 6, multiply by 6, and add to the score.

[1]Yes, Sebastian is the name of one of David's pet birds and yes this is yet another bird-themed problem.

- Company: If the five dice are either 1, 2, 3, 4, 5 or 2, 3, 4, 5, 6, the player can add 40 points to their score. (Note that for this and all other categories, the order of the dice is not important.)
- Prattle: If four of the five dice are either 1, 2, 3, 4, or 2, 3, 4, 5, or 3, 4, 5, 6, the player can add 30 points to their score.
- Squadron: If three of the dice show the same number, and the other two dice are also the same, the player can add 25 points to their score.
- Triplex: If three of the dice are the same, the player can add up the values of all five dice and add this to their score.
- Quadrupla: If four of the dice are the same, the player can add up the values of all five dice and add the sum to their score.
- Quintuplicatam: If all five dice are the same, the player can add 50 points to their score.
- Pandemonium: The sum of all five dice, no matter what they are.

Players can choose which category to fill ~~with any particular roll~~ at the end of each turn, but **each category may be filled only once per game.** A player can also choose to assign a roll to a category that does not match the requirements, but then a zero is entered into that category.

For example, here's what a few moves of the game might look like, with a single player:

- Player rolls dice and gets 1, 2, 3, 4, 5 on the first roll. They decide not to reroll any dice. Player assigns it to Company and gets 40 points. (They could have decided to assign it instead to Prattle for 30 points, or to one of the first 5 categories to get a score of 1 for Primis, 2 for Secundus, 3 for Tertium, 4 for Quartus, or 5 for Quintus, or 15 for Pandemonium. Or they could have chosen to get a 0 for any other category).
- Player rolls dice and gets 1, 3, 3, 4, 4. They decide to reroll the 1, and they get a 2. They reroll the 2, and get a 6. So the final dice are 6, 3, 3, 4, 4. They could assign this to ~~Primis for 1 point,~~ Tertium for 6 points, or Quartus for 8 points, or 6 for Sextus, or 20 for Pandemonium, or any other category for 0 points. They choose to assign to Quartus, and now have 48 points (i.e., 40 from the first turn plus 8 from this turn).
- Player rolls dice and gets 1, 2, 2, 3, 3. They reroll the 1 and get 4, and reroll the 4 and get 5. So the final dice are 5, 2, 2, 3, 3. Their choices now are: Quintus for 5 points, ~~Primis for 1 point,~~ Secundus for 4 points, Tertium for 6 points, or 15 for Pandemonium, or 0 in another category. They choose Tertium and now have 54 points.
- Player rolls dice and gets 1, 2, 3, 4, 5 on the first roll. Player can't assign it to Company since they've already used this category already, so they count it as a Prattle instead, and now have 84 points.
- Player rolls dice and gets 1, 2, 3, 4, 5 after their 3 rolls. The player can't take either Company or Prattle, or Tertium or Quartus since they have been used, but they could take Primis for 1 point, Secundus for 2 points, or Quintus for 5 points, or 15 for Pandemonium. However, the player decides it's unlikely they'll ever manage to get a Quintuplicatam, whereas it's much more likely they'll be able to get a better score for one of those other categories, so they assign to Quintuplicatam and get a score of 0.
- And so on...

After 13 turns, all categories are full, and the game ends. If the player managed to get a score of at least 63 totaled across the first six categories (Primis through Sextus), they get a bonus of 35 points added to their score.

Implement a program that plays Sebastian as well as possible, i.e. getting the highest possible score. Your goal is to achieve as high an average score as possible. As with Assignment 1, a small portion of your grade will be based on how well your program works with respect to the rest of the class. To get you started, we've implemented some skeleton code that should be in your cloned repository. You can run it like,

```
python3 ./sebastian.py
```

The skeleton code implements a very naive automatic player that simply rolls once and assigns the roll to whichever category is next available. You'll want to modify the file called SebastianAutoPlayer.py. There are helper programs called sebastian.py and SebastianState.py to keep track of the scoreboard and to roll the dice. While you should look at SebastianState.py to understand how these classes work, and you may want to modify SebastianState.py for debugging purposes, your final submission should run **using the SebastianState.py file that we supplied without any modifications**, or else we won't be able to grade your submission correctly.

Part 2: Betsy

Chess has been called the "drosophila of artificial intelligence," since it has long been a convenient yardstick by which to measure progress in AI (just as the fruit fly has been a relatively simple experimental "platform" for biology). Let's consider Betsy, a somewhat simplified version of Chess.

The game is played by two players on a board consisting of a grid of 8×8 squares. Initially, each player has sixteen pieces: 8 Parakeets, 2 Robins, 2 Nighthawks, 2 Blue jays, 1 Quetzal, and 1 Kingfisher. The two players alternate turns, with White going first. On each turn, a player moves exactly one of his or her pieces, possibly capturing (removing) a piece of the opposite player in the process, according to the following rules:

- A Parakeet may move one square forward, if no other piece is on that square. Or, a Parakeet may move one square forward diagonally (one square forward and one square left or right) if a piece of the opposite player is on that square, in the process capturing that piece from the board. If a Parakeet reaches the far row of the board (closest to the opposite player), it is transformed into a Quetzal. On its very first move of the game, a Parakeet may move forward two squares as long as both are empty. (Note that due to the rules of the game and the initial game state (see below), you can tell if a Parakeet has never been moved based on its position: a `P` (white Parakeet) that has not been moved will be in the second row of the board, and a `p` (black Parakeet) will be in the seventh row of the board.)
- A Robin may move any number of squares either horizontally or vertically, landing on either an empty square or a piece of the opposite player (which is then captured), as long as all the squares between the starting and ending positions are empty.
- A Blue jay is like a Robin, but moves along diagonal flight paths instead of horizontal or vertical ones.
- A Quetzal is like a combination of a Robin and a Blue jay: it may move any number of empty squares horizontally, vertically, or diagonally, and land either on an empty square or on a piece of the opposite player (which is then captured).
- A Kingfisher may move one square in any direction, horizontally or vertically, either to an empty square or to capture a piece of the opposing player.
- A Nighthawk moves in L shaped patterns on the board, either two squares to the left or right followed by one square forward or backward, or one square left or right followed by two squares forward or

backward. It may fly over any pieces on the way, but the destination square must either be empty or have a piece of the opposite player (which is then captured).

A player wins by capturing the other player's Kingfisher. (Note some of the differences with traditional Chess: there's no notion of check or checkmate, no en passant, and no castling.)

Your task is to write a Python program that plays Betsy well. Use the minimax algorithm and a suitable heuristic evaluation function. You may want to implement alpha-beta pruning as well. Your program should accept a command line argument that gives the current state of the board as a string of 64 characters, each of which is one of: `.` for an empty square, `P` or `p` for a white or black Parakeet, `R` or `r` for a white or black Robin, `N` or `n` for a white or black Nighthawk, `Q` or `q` for a white or black Quetzal, `K` or `k` for a white or black Kingfisher, and `B` or `b` for a white or black Blue jay, in row-major order. For example, the encoding of the start state of the game would be:

```
RNBQKBNRPPPPPPPP................................pppppppprnbqkbnr
```

More precisely, your program will be called with three command line parameters: (1) the current player (w or b), (2) the state of the board, encoded as above, and (3) a time limit in seconds. Your program should then decide a recommended single move for the given player from the given current board state, and display the new state of the board after making that move, *within the number of seconds specified.* Displaying multiple lines of output is fine as long as the last line has the recommended board state. (This is an easy way of dealing with the time limit: the program can very quickly calculate and print a suggested "rough-draft" move, and then print out better moves as it finds them; our test programs will kill your program after the time limit has passed and look only at the last move.) For example, a sample run of your program might look like:[2]

```
[djcran@macbook]$ python3 ./betsy.py w RNBQKBNRPPPPPPPP................................pppppppprnbqkbnr 10
(Hey, human, in the time it takes you to read this sentence, I'll have considered
5 billion board positions. But it's cute that you're still trying to beat me...)

Hmm, I'll move the Parakeet at row 2 column 3 to row 4 column 3.
New board:
RNBQKBNRPP.PPPPP..........P.....................pppppppprnbqkbnr
```

In your source code comments, explain your heuristic function and how you arrived at it.

The tournament. To make things more interesting, we will hold a competition among all submitted solutions. We will not reveal ahead of time the time limit, but we plan to hold multiple tournaments with different values. While the majority of your grade will be on correctness, programming style, 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.

Note: Your code must conform with the interface standards mentioned above! The last line of the output *must* be the new board in the format given, without any extra characters or empty lines. We will provide an output checker to help you verify this. We will also provide a program that will allow you to play against other teams in the class without having to share your Python code (which would violate the academic integrity policies of the course).

Also, note that your program cannot assume that the game will be run in sequence from start to end; given a current board position on the command line, your code must find a recommended next best move. Your program can write files to disk to preserve state between runs, but should correctly handle the case when a new board state is presented to your program that is unrelated to the last state it saw.

[2]The "trash talk" is optional. :)

What to turn in

Turn in the three programs 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. **Your programs must obey the input and output formats we specify above so that we can run them, and your code must work on the SICE Linux computers.**