Awakening of Azathoth

FAQ

`EvilChooser` is now optional.

Each assignment will have an FAQ linked at the top. You can also access it by adding “/faq” to the end of the URL. The FAQ for Project 0 is located here.

There are also slides that are complementary to the FAQ at this link.

Introduction

In this project, we will focus on the *client view* of data structures. The idea is to become familiar with how to use the fundamental different types of data structures (lists, maps, sets, etc.). Later in the class, we’ll focus on building data structures.

Awakening of Azathoth

In the game “Awakening of Azathoth”, a player has a limited number of tries to reveal a word by guessing letters, one at a time. Each time a player guesses a correct letter, all occurrences of that letter in the word are revealed. But when a player guesses an incorrect letter, they lose one of their guesses and no letters are revealed. If the player reveals the word before they run out of guesses, Azathoth continues its slumber, but if they run out of guesses, Azathoth is awakened, and the game is lost.

Let’s start by playing the game. A browser based implementation of the game can be found at this link.

Try playing it a couple of times until you've fully understood how the game works. There's a volume meter you can turn up if you want sound effects.

In this project, you'll first create three different "guessers", which are AI players that can play the game.

Then you'll build two different "choosers", which are classes that select the word to be guessed. While this seems like it is a trivial problem, it turns out there is a very clever twist (courtesy of Keith Schwarz) that makes this an amazing data structures problem.

By the end of this project, you'll:

- Be able to play games of Awakening of Azathoth (AoA) using your implementations.
- Have experience working with Java's built-in `List` and `Map` data structures.

> **Note**: You may have played a version of this game called Hangman. AoA is the same game, but we rebranded it since IMO public executions aren't really the best vibe, and there's also a very interesting twist to this project that will come at the end.

> **Note**: There is a **lot** of syntax in the skeleton code for this project that you've never seen before and which we haven't covered in lecture or lab. We'll be going over much of it in more detail in future weeks, so don't panic. Your job will be to focus on filling in the provided functions and constructors according to the specifications provided.

Getting Started

> **Make sure that you have completed Lab 1**. You will not be able to start the project if you have not done the setup in Lab 1.
>
> **This assignment assumes you have completed hw0a and hw0b!**

Starting the Assignment

Follow the instructions in the Assignment Workflow guide to get the skeleton code and open it in IntelliJ. This project is `proj0`.

Restarting

At some point, you might get *really* frustratingly stuck, and want to restart completely. Don't delete your files! Instead, we can use Git to start over. To restart, run this command in your `sp23-s***` directory:

```
git restore --source=skeleton/main proj0
```

⚠ Beware: this command will get rid of all changes to anything in the `proj0` directory that you haven't committed. Be sure to add and commit your code *before* doing this.

Guessers

Your first task will be to build three Guessers. Guessers are essentially AI players that will "play" your game by picking the next letter to guess. As you'll see, each guesser will be an improvement on the previous one, using an increasingly complex and effective strategy to decide which letter to guess.

NaiveLetterFreqGuesser

Open the `NaiveLetterFreqGuesser.java` file. You should see the following class declaration:

```
public class NaiveLetterFreqGuesser implements Guesser {
```

Your task is to fill in the unimplemented functions: `getFrequencyMap` and `getGuess`. Since we haven't implemented them, if you run the main method, you'll see the following output:

```
list of words: [ally, beta, cool, deal, else, flew, good, hope, ibe
frequency map: {}
guess: ?
```

Open up the `tests/aoa/guessers/NaiveLetterFreqGuesserTest.java` file and run the provided tests, similar to what you did in hw0b. You'll see that you're failing all the tests. By the end of this part of the assignment you'll be passing them. Note: The tests for project 0 are the same ones that the autograder runs, so if you pass all the tests we've provided, you'll also get full score on the autograder. In later projects, this will not be the case.

getFrequencyMap

`public Map<Character, Integer> getFrequencyMap()`

This function returns a map that associates each letter to the count of that letter across the entire list of words. That is, after this function is working you should see the following output when running the `main` method:

```
list of words = [ally, beta, cool, deal, else, flew, good, hope
frequency map = {a: 3, b: 2, c: 1, d: 2, e: 7, f: 1, g: 1, h: 1
guess: ?
```

Here, we see that there are, for example, 3 total `a` characters in the file, which we can see by looking at the list of words. These occurrences of `a` are in `ally`, `beta`, and `deal`.

Note: the number of occurrences of a letter is different from the number of words that contain the letter. For example, `secret` has 2 occurrences of the letter `e`.

Task: Implement the `getFrequencyMap` method in `NaiveLetterFreqGuesser` as described above. Run the tests, and check that you are passing the following:

- `testFreqMapSmallFile()`
- `testFreqMapLargeFile()`

---

getGuess

`public char getGuess(List<Character> guesses)`

This method returns the letter with the greatest frequency that has not yet been guessed. The list of previous guesses is given by the `List<Character> guesses` argument.

For example, if the frequency map is `{a=3, b=2, d=2, e=7, l=6, o=5}` and `guesses` is `['e', 'l']`, then the next guess should be `o`.

If there are ties, you should break ties by picking the earliest letter in the alphabet, e.g. if you had to pick between `b` and `d`, you'd choose `b`.

NOTE: The easiest way to resolve ties is by using a `TreeMap` instead of a `HashMap`, since a `TreeMap` stores its keys in sorted order.

If the frequency map is empty, then the question mark character ('?') should be returned.

Task: Implement `getGuess`, then use the provided main method and the tests and confirm that your `NaiveLetterFreqGuesser` is fully correct.

PatternAwareLetterFreqGuesser

When you play AoA, you're provided with information about which letters match. For example, let's assume the secret word is `beta`, and the player picks the letter `e`. In that case, the returned "pattern" will be `-e--`.

Looking at the list of words above, there are only two words that match: `[beta, deal]`. If we count up the remaining letters, the new frequency map is: `{a=2, b=1, d=1, e=2, l=1, t=1}`. Since `e` was guessed, the most common letter (tiebreaking by alphabet order) is `a`. Note that this is different from `NaiveLetterFreqGuesser`, which wouldn't have considered the provided pattern and would have chosen `l` instead. Thus, this guesser will be a better player than the `NaiveLetterFreqGuesser`.

In `PatternAwareLetterFreqGuesser`, you only need to fill in the `getGuess(String pattern, List<Character> guesses)` method. Here, `pattern` is a string like `-e--` or `-ood` or `e--e`.

For this problem, we haven't provided any helper methods, but you're encouraged to write them. For example, in my solution, I wrote a method called `getFreqMapThatMatchesPattern` which returns a frequency map like the one in the example above (`{a=2, b=1, d=1, e=2, l=1, t=1}`). But you could also make a function called something like `keepOnlyWordsThatMatchPattern` that takes as input a list of words and a pattern, and only returns words that match. The magic of helper functions is that you can make up whatever you want.

If you write any helper methods (which you really should!), we encourage you to write code to verify that they work before using them. You can do this by adding function calls to your `main`, e.g. `System.out.println(getFreqMapThatMatchesPattern(pattern))` or `System.out.println(keepOnlyWordsThatMatchPattern(pattern)`.

You're also welcome to add automated tests for your helper methods to `PatternAwareLetterFreqGuesser`, though we won't be officially teaching you how those work until Lab 2.

Task: Implement the methods in `PatternAwareLetterFreqGuesser`, including any helper methods.

You'll know you're done when you pass all the tests in `PatternAwareLetterFreqGuesserTest`.

Note: If you're failing the `getGuess` tests in `PatternAwareLetterFreqGuesserTest` and don't know where to start, we recommend checking that you add print statements to help debug. In Lab 2, we'll also see how to use the debugger. If you'd like you are welcome to look ahead and learn how it works.

Note: In the full dataset, not every word is of the same length! Thus, the words and `pattern` may not be the same length. For example, suppose the pattern is `---l`. In that case, the counts of the word `liminal` should not be included in the frequency map, because the pattern and word lengths differ, and thus the secret word obviously couldn't be `liminal`.

Note: Consider the pattern `-l--`. Technically, this only matches `flew`. However, the `PatternAwareLetterFreqGuesser` should *also* say that it matches `ally`. This guesser should only consider the pattern letter-by-letter, and should not consider the case where a letter is revealed in one spot and not revealed in another spot.

PAGALetterFreqGuesser

The `PatternAwareLetterFreqGuesser` is an improvement over the naive approach, but it can still be significantly improved. For example, suppose that the first guess was `e`, and that the word had no `e` in it. In this case, you'd get back the pattern `----`, which seems to provide no useful information.

However, the fact that we don't see an `e` in the pattern means the word has no `e`. For the example above, that means that we know that the only possible words are `[ally, cool, good]`, because the other words all have an `e` in them. If we were to count up the letters in these words, we'd get `{a=1, c=1, d=1, g=1, l=3, o=4, y=1}`.

The final guesser you'll write is `PAGALetterFreqGuesser`, where PAGA stands for "Pattern And Guesses Aware". This guesser should take into account the pattern (like your previous guesser), and should also take into account guesses which do not appear in the pattern, e.g. the fact that `----` contains no `e`.

Task: Implement the methods in `PAGALetterFreqGuesser`, including any helper methods. Do not add any additonal instance variables apart from the ones that we have provided; this will cause some integration tests to fail.

You'll know you're done when you pass all the tests in `PAGALetterFreqGuesserTest` and `PAGALetterFreqGuesserIntegrationTest`.

Warning: Another implication of taking guesses into account can be seen in the pattern `be-`. This pattern **does not** match `bee`, because all the `e`'s have been revealed.

Quick Note on Code Reuse

One undesirable result of the way we structured this assignment is that you probably have a large amount of nearly redundant code between your three guessers. You can avoid such redundancy by putting common helper methods into a single helper class with a suitable name, e.g. `LFGHelper`. You're welcome to do so for this project. Note that in this case, all such methods should be static (since the `LFGHelper` class is never instantiated and carries no data of its own).

Choosers

Now that you've built some guessers, it's time to build a word chooser. You'll first implement `RandomChooser`, which randomly chooses a word and provides the appropriate feedback. For example, the RandomChooser might select the word `cool` in its constructor. If the guesser picks `o` using the `makeGuess('o')` function, then it will create the pattern `-oo-`, which will be returned on the next call to `getPattern`.

RandomChooser

`RandomChooser` Constructors and Methods

```
public RandomChooser(int wordLength, String dictionaryFile)
```

The constructor takes two arguments:

`int wordLength`
: The desired word length

`String dictionaryFile`
: A filename containing the dictionary to select a word from, with one word on each line.

You should throw an `IllegalArgumentException` if `wordLength` is less than one. You should throw an `IllegalStateException` if there are no words found of `wordLength`.

You should randomly select a word from the provided dictionary that is `wordLength` characters long to be the secret word. You will likely want to store the chosen word as a field. You **must** use StdRandom from algs4! This is so that we can test your code by controlling the randomness.

Your code to select a random word should look more or less the same as this. Note that you'll need to assign the result of the `words.get` call to a variable, i.e. `String word = words.get(...`.

```
int numWords = words.size();
int randomlyChosenWordNumber = StdRandom.uniform(numWords);
words.get(randomlyChosenWordNumber);
```

Here, `words` should be a list of words of the desired length in sorted order, i.e. the output of calling `readWordsOfLength` from `FileUtils.java` in `utils`.

`public int makeGuess(char letter)`

This method should return the number of occurrences of the guessed letter in the secret word.

You may assume all guesses passed are valid, i.e. they are lowercase letters and have not been guessed before.

You should also update your `pattern`.

`public String getPattern()`

This method should return the current pattern to be displayed for the game using the guesses that have been made. Letters that have not yet been guessed should be displayed as a dash (`-`). There should be no leading or trailing spaces.

`public String getWord()`

This method should return the secret word being considered by the `RandomChooser`.

Task: Implement the methods in `RandomChooser` as described above. Also, be sure to take a look at the code in `utils/FileUtils.java` for some helper methods that might help you!

Testing RandomChooser

You can test your RandomChooser by running the tests in `RandomChooserTest` and `RandomChooserIntegrationTest`.

Playing a Game

Once you've passed all the tests in `RandomChooser`, it's time to try out a game. Open the `AoAGame.java` file and run the game. Note that you'll have to pick a word length of 4 because all of the words in `example.txt` are of length 4. Try looking at the words in `example.txt` to help you win.

Now change the `DICTIONARY_FILE` to be `"data/sorted_scrabble.txt"` and try again. You'll see you can pick other word lengths.

You can also see how your AI players fare. To do this, open the `main` method in `AoAGame` and change

```
Guesser guesser = new ConsoleGuesser(console);
// Guesser guesser = new PAGALetterFreqGuesser(DICTIONARY_FILE);
```

So that it reads:

```
// Guesser guesser = new ConsoleGuesser(console);
Guesser guesser = new PAGALetterFreqGuesser(DICTIONARY_FILE);
```

In other words, the guesser will no longer be whoever is using the console (i.e. you), and will instead be the `PAGALetterFreqGuesser`.

(Optional) `EvilChooser`

The final and most interesting part of this project is the implementation of the `EvilChooser`. Unlike the unintelligent `RandomChooser`, the `EvilChooser` is a true servant of Azathoth.

This diabolical chooser will be harder to beat than a `RandomChooser`. It's interesting to think about how this might be possible? The most obvious way would be for `EvilChooser` to select from words that humans are likely to fail to find, e.g. `jazz`, but it's actually even worse than that.

How is it Evil?

Your `EvilChooser` is shifty and devious. It should pretend to pick a word, while delaying its actual selection until it is forced through the power of logic to actually do so.

As a result, the computer is always considering a set of words that could be the answer. In order to fool the user into thinking it is playing fairly, the computer only considers words with the *same letter pattern*.

For example, suppose that the computer knows the words in `data/example.txt`:

ally	beta	cool	deal	else	flew	good	hope	ibex

Instead of beginning by choosing a word, the `EvilChooser` narrows down its set of possible answers *as* the user makes guesses. When the user guesses `e`, the computer must reveal where the letter `e` appears, if at all. Since it hasn't chosen a word yet, its options fall into *five* "families":

Pattern	Family
`----`	`[ally, cool, good]`
`-e--`	`[beta, deal]`
`--e-`	`[flew, ibex]`
`---e`	`[hope]`
`e--e`	`[else]`

The guess forces the `EvilChooser` to choose a *family* of words, but not a particular *word* in that family. The computer could use several different strategies for picking the family to display. Your implementation should **always choose the family with the largest number of words.** This is a reasonable strategy that leaves the chooser's options open!

For the example described above (guessed `e`), the computer would pick the family with the pattern `----`. This reduces the possible answers it can consider to:

`ally`	`cool`	`good`

Since no letters were revealed, this counts as a wrong guess and the player loses a guess.

The user then guesses the letter `o`. The computer now has *two* word families to consider:

Pattern	Family
`-oo-`	`[cool, good]`
`----`	`[ally]`

It picks the biggest family and reveals the letter `o` in two places. Since the computer revealed some letters, this was a correct guess and the number of guesses left does not change. The computer now has only two possible answers to choose from:

`cool`	`good`

If the user guesses a letter that doesn't appear anywhere in the set of words, such as `t`, the family you previously chose will still match, as it's the only family, and no letters were revealed. Since no letters were revealed, you'd count `t` as a wrong answer.

The user then guesses `d`, resulting in two families of the same size:

Pattern	Family
-oo-	[cool]
-ood	[good]

The computer again has several different strategies for picking the family to display. Among the families that are of largest size, your implementation should **always choose the family with the pattern that comes alphabetically earlier.** For computers, the character `-` comes alphabetically before all lowercase letters! Here, the computer would pick `-oo-`, locking itself into `cool` as the answer (and counting `d` as a wrong guess).

`EvilChooser` Constructors and Methods

You should implement the same methods as in `RandomChooser` with the additional restriction that they use the "evil" algorithm described above. Many of them will be identical, but the constructor and `makeGuess` will likely be significantly different.

`public EvilChooser(int wordLength, String dictionaryFile)`

The constructor takes the same arguments as `RandomChooser`'s, and should throw the same exceptions in the same cases. However, since this is *evil*, you shouldn't select a single word immediately!

`public int makeGuess(char letter)`

This method should find all the possible word families split on the guessed letter and pick the one with the most words. If there is a tie (two of the word families are of equal size), you should pick the one whose pattern is alphabetically earlier. The set of words representing the biggest family then becomes the dictionary for the next round.

NOTE: The easiest way to resolve ties is by using a `TreeMap` instead of a `HashMap`, since a `TreeMap` stores its keys in sorted order.

Once the new pattern is selected, return the number of occurrences of the guessed letter in the new pattern.

`public String getPattern()`

Just like `getPattern()` in `RandomChooser`, this method should return the current pattern to be displayed for the game using the guesses that have been made. Letters that have not yet been guessed should be displayed as a dash (`-`). There should be no leading or trailing spaces.

`public String getWord()`

This method returns any word from current `wordPool`. If there is only one word exists in `wordPool` return that word.

Testing EvilChooser

You can test your EvilChooser by running the tests in `EvilChooserTest` and `EvilChooserIntegrationTest`.

`EvilChooser` Tips

- The patterns will come from the words themselves. On any given turn, there is a current set of words that all have the same pattern. When the user guesses a new letter, go through each of the words that you have in the current dictionary and figure out what the correct new pattern would be for that particular word given the new guess.
- Use a `Map` to associate family patterns with the set of words that have each pattern.
    - Once again, the easiest way to resolve ties is by using a `TreeMap` instead of a `HashMap`, since a `TreeMap` stores its keys in sorted order.

- Once you have processed all the words, go through the different sets and find the one with the most words (resolving ties appropriately). That one becomes the new dictionary used by the `EvilChooser`!
- You probably don't need a `Map` as a field. Think about where it's used!

Task: Implement the methods in `EvilChooser` as described above.

Playing a Game II

Once again, it will be beneficial to actually play a game with your implementation. In `AoAGame`, change the constructor called for `chooser` to `EvilChooser`:

```
Chooser chooser = new EvilChooser(wordLength, DICTIONARY_FILE);
```

It should be noticeably more difficult to win against your evil implementation than the random one.

Extra: `EvenBetterGuesser` or `EvenBetterChooser`

Optionally, you might consider writing a second guesser or chooser that does something more intelligent in an attempt to regularly beat the evil chooser. If you do, let us know, we'd love to see what you come up with!

Grading

For this project, we are emphasizing the ability to **correctly implement a specification**. To evaluate this, we don't think that giving individual tests proportional weight makes sense. Some tests check essentially the same thing; and writing more tests means that we've given it more weight. Some tests don't check a behavior and instead check an implementation detail. Therefore, rather than weighting **individual tests** against each other, we have divided the project into **larger chunks**, each of which has its own thorough test suite.

Since it's the first project, **we have no hidden tests**. That is, the tests you can run locally are all the tests that the autograder will run.

Grading is divided into 5 parts, 3 of which are worth credit:

- **`NaiveLetterFreqGuesser` (0%)**: This class is only a stepping stone towards the `PAGALetterFreqGuesser` and will not be graded.
- **`PatterAwareLetterFreqGuesser` (0%)**: This class is only a stepping stone towards the `PAGALetterFreqGuesser` and will not be graded.
- **`PAGALetterFreqGuesser` (40%)**: To receive credit, you must implement `PAGALetterFreqGuesser` completely correctly.
- **`RandomChooser` (20%)**: To receive credit, you must implement `RandomChooser` completely correctly.
- **`EvilChooser` (40%)**: To receive credit, you must implement `EvilChooser` completely correctly. (Made optional, check Ed for more information)

Acknowledgements

`EvilChooser` is based on a Nifty Assignment by Keith Schwartz. All other parts are based on additions made by Adam Blank.

Last built: 2023-10-26 18:40 UTC