Modeling Challenge I: Word Ladders

Eckel, TJHSST AI1, Fall 2020

Background & Explanation

In Artificial Intelligence, understanding the algorithms and actually being able to use them in novel situations are equally important. As such, throughout our units, we will periodically have assignments called "Modeling Challenges". These are assignments where I'll give you a new context for an algorithm we're learning, but I won't tell you how to use it. These assignments won't teach you new algorithms, but they will build your skills in applying algorithms in new contexts. So, here is a new problem you can solve using BFS, and you get to decide how it's going to work.

The challenge is to generate word ladders from a list of words. A word ladder is a sequence of words that each differ from the previous word by exactly one letter. For example, this is a word ladder:

```
HEAD
HEAL
TEAL
TELL
TALL
TAIL
```

Your goal in this lab is to read in a list of six-letter words, store in some kind of backing data structure which words connect to which other words by changing a single letter, and then use this data structure to find word ladders. You'll be given a file of puzzles – pairs of words – and you must either generate a minimal length ladder from one word to the other or print that such a ladder does not exist in that dictionary.

You know all the theory you need to accomplish this task (BFS search for shortest path) but implementing it is another matter. As a starting point, your BFS code should look *almost exactly identical* to the Sliding Puzzles code. It's the reading in the files and storing the words and figuring out how to get children that'll be different here.

Files Provided

Several files are provided for you.

- For the standard specification you want to look at these three files. If you don't do an OW, this is all you need:
  - **`words_06_letters.txt`** is your dictionary
  - **`puzzles_normal.txt`** is your puzzle file
  - **`word_ladder_sample_run.txt`** is a sample run to show you what the output should look like
- For the optimization Outstanding Work specification you want to look at these two files:
  - **`words_06_longer.txt`** is your dictionary
  - **`puzzles_longer.txt`** is your puzzle file
  - no sample run is provided; this should do the same thing as the normal spec, just faster, so you already know if your code works, it's just a question of whether it's fast enough
- For the all-lengths Outstanding Work specification you want to look at these three files:
  - **`words_all.txt`** is your dictionary
  - **`puzzles_all.txt`** is your puzzle file
  - **`word_ladder_all_sr.txt`** is your sample run

Important note: **your word ladders may be different from the ones shown in the sample runs**, but they should be **exactly the same length.** If you find a shorter ladder, your code is incorrect because it's making impossible moves; if you find a longer one, your code is incorrect because it's missing the shortest path somehow.

A Quick Intro to Python File I/O

As I'm sure you expect at this point, file I/O in python is much simpler than Java. This code, for example, will read in every line of a file and store each line in a list of strings.

```
with open("some_file.txt") as f:
    line_list = [line.strip() for line in f]
```

The `.strip()` command is essential here. It removes any leading or trailing whitespace on a string. In this case, it will remove the "\n" from each line that has one, but if the last line of the file doesn't end in "\n" it will still work without removing a random final character from the last word. You'll also want **`.split()`** for the file of puzzles.

If you do some google searching, you can find other ways of doing file I/O and you should feel free to use them if they feel more right to you. But the two-line solution above is one quick way of reading in a file for later processing, and it'll do just fine for this program.

Either way, one **absolutely necessary** guiding principle: when using any file, **you only want to read from the file ONCE.** Open it once, read it, and store the information in a data structure, then search or manipulate the data structure. Continually searching a file for information is **extremely slow** compared to searching data structures stored in RAM.

Implementation

From here, the choices are yours! I won't give any advice about how to store the words or generate children. You might even be able to learn a lot, in this assignment, by trying it a couple different ways and seeing what is fastest.

Be sure to read all the bullet points in the specification carefully, and to double check the sample run, before submitting. This is an assignment where students often get small details wrong; check your work **carefully** before turning it in.

A Few Word Ladder Brainteasers

Once your code is generating word ladders successfully, you're ready to submit it. I'm only going to test word ladder generation with my grading scripts.

But: **before you submit**, I'd like you to save a separate copy of your code and use it to answer a few puzzles.

1) Several words in **`words_06_letters.txt`** actually don't connect to any other words at all (ie, have no valid children). How many words are singletons like this?
2) On the other hand, a whole lot of the dictionary lives in one big clump. What is the number of words in the largest connected subcomponent of this graph? In other words, what is the size of the largest group of words that can all be reached from each other by a word ladder of some length? (Answering this should be very similar to finding the total number of solvable 3x3 states!)
3) How many clumps (or connected subcomponents) are there, total? Excluding the singletons in #1 but including the huge clump in #2, how many different, separate clumps are there with at least two words in them?
4) BFS always finds **ideal paths**, ie the shortest path between two nodes. What is the **longest ideal path** between two words in this dictionary? (This question is a lot like "what is the hardest 3x3 puzzle" from the last assignment!) Give at least one pair of words that produce an ideal word ladder of the longest possible length, as well as the complete solution from one to the other and the length of that solution. (As a hint, you can logically assume the longest path is inside the largest connected subcomponent found in question 2.)

Send your answers to this part to me in a DM on Mattermost. Then, submit only the word ladder generating code to the Dropbox link. See the specification below for details.

Specification

Submit a single Python script to the link on the course website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- You have sent me the answers to questions 1-4 on Mattermost before submitting.
- Your code does all of the following, in order (**please read carefully**):
    - Accept **two** command line arguments. The first argument is the name of a txt file containing six-letter words. The second argument is the name of a txt file containing word ladder puzzles.
    - Read in the words from the dictionary file, building your backing data structure.
    - Display the time taken to generate your data structure.
    - Read puzzles from the second txt file and solve them. Each puzzle is two words on the same line. Find the shortest word ladder from each word to the other and output the size of the word ladder followed by each word in it, in order. If the ladder is impossible, say so. (See the sample run on website.)
    - Display the total time taken to solve all of the puzzles.
- Total combined runtime is less than two minutes. (The dictionary used will be smaller than the example provided, so should execute more quickly.)

For **resubmission**:

- Complete the specification correctly.

Specification for Outstanding Work: Word Ladder Optimization

There are two ways to extend this assignment and get outstanding work credit. One way is to optimize this assignment beautifully. The submission link for this extension is on the course website.

This extension **receives outstanding work credit** if:

- Your code meets the complete specification on the front of the paper.
- Total runtime (generating data structure and solving puzzles combined) on the standard specification's example files is less than a tenth of a second. (As you can see in my sample run, it's possible to clear this with room to spare, and I promise I haven't used any data structure or algorithm I haven't taught you. This is standard BFS.)
- Total runtime on the bigger files "words_06_longer.txt" and "puzzles_longer.txt" is less than 1.5 seconds.

It seems to be the case that that the laptop I grade on is as fast or faster than most student laptops. If you clear that time on your computer, you should on mine as well. (Certainly applies to FCPSon computers.)

Specification for Outstanding Work: Any-Length Word Ladders

The other way to get outstanding work credit for extending this assignment is to modify it to work with words of any length. You can do both of these if you like, getting separate credit for each one. The submission link for this extension is on the course website.

- Modify your code so that, in addition to one letter being *changed*, it is also a valid move to *eliminate* or *add* a single letter (keeping the remaining letters in the same order). Of course, after doing so, you must still have a valid word in the dictionary. Now, word ladders can be constructed from any word length to any other.
- Download the additional files "words_all.txt" and "puzzles_all.txt" to test your code. These contain words of all lengths.

This extension **receives outstanding work credit** if:

- Your code outputs everything specified in the standard specification on the previous page of this sheet (using, of course, the new modifications).
- **Your code also generates and prints answers to questions 1, 2, 3, and 4 from earlier in this assignment, and prints their answers below all of the puzzle solutions it has generated.**
    - Note that I'll be using a different dictionary from the one provided to you, so hardcoding any part of this is a bad idea!
    - Generating these answers is included in the runtime total below; this means you will need to find an efficient way of finding the longest path. The same hint applies here, about logically assuming the longest path is inside the largest connected subcomponent found in question 2.
- **Total** runtime is less than two minutes. (For what it's worth, mine runs in about 10 seconds on the given files.)