BFS Application: Word Ladders

Eckel, TJHSST AI1, Fall 2023

Background & Explanation

In Artificial Intelligence, understanding the algorithms and actually being able to use them in novel situations are equally important. In this case, we're going to use BFS just like we did with sliding puzzles, but the situation here will be so different that you'll get practice with a couple of important skills:

- Recognizing when a familiar tool applies to wildly different scenario
- Making different choices in the modeling process to be efficient in different situations

Word Ladders

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.

- **`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

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.

File I/O reminders

As mentioned on the Sliding Puzzles assignment, 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.

Once again, here's good example code:

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

Don't forget: the `.strip()` command is essential here. You'll also want **`.split()`** for the file of puzzles.

Implementation

Take a moment and think about applying BFS to this situation. What would `goal_test()` be? What would `get_children()` be?

Really ponder for a moment, before reading on.

The crucial difference here – and the reason we're doing this assignment – is that getting children is completely different in this situation. In sliding puzzles, the only information we needed to determine a node's children was just the node itself. That's not true here. Here, any word could have theoretically 150 different "children" but only a small number of them are *actually words*. On the other hand, the sliding puzzles have an enormous number of total nodes (181,440 for 3x3 alone), and this dictionary has less than 5,000 words. So where it made sense in that assignment to generate children on the fly, the relative efficiencies here are different. Now, it'll make sense for us to *pre-generate the entire graph* so that we can avoid repeating work every time we need to find a list of children.

So – that's your job. Read in the file, and find an efficient way to store every word paired with all of its potential children in advance, before you start solving any puzzles. Then, any call to `get_children()` is just a lookup in the data structure you've already made.

Creating Word Ladders

Once you've figured out your backing data structure, take a look at the sample puzzle file and the sample run output. It should be fairly self-explanatory what needs to happen here; make it happen!

You might notice at the bottom of the sample run that there are some other bits of information; indeed, just like the BFS Sliding Puzzles assignment, I have some brain teasers for you. This time, they need to be part of the code you submit.

Word Ladder Brainteasers

Answers to these brainteasers must be printed by the code you submit, unlike last time. Add code to produce these answers based on the dictionary it reads in and print these answers as well at the bottom of your output. See sample run to double check!

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!) The concept of a "clump" seems to confuse a lot of students each year – feel free to ask if you're not sure what this is asking for.
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? (To clarify: two words are part of the same clump if and only if a word ladder can be constructed between 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. (You may notice in the sample run that my code finds this very fast. Yours should too – much less than a second. If you're finding this difficult to achieve, please ask for a hint – there is a non-obvious insight here you may need!)

Specification

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

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- 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.)
  - Display the total time taken to solve all of the puzzles.
  - Display the answers to the 4 brainteasers, based on the word list given on the command line.
  - Display the total time taken to solve all of the brainteasers. (You don't need to show the time for #4 separately like I did if you don't want to.)
- Total combined runtime should be very fast here; less than five seconds.