Foundations of AI
Instructor: Rasika Bhalerao
Assignment 3
Due October 7

Is there a series of single-character changes that you can make to get from the word "`cat`" to the word "`dog`" such that every word along the way is also a 3-letter word in the dictionary?

An example solution would be: `['cat', 'cot', 'cog', 'dog']`

The goal of this assignment is to write algorithms that can efficiently find a solution for us.

Learning goals:

- Practice implementing search algorithms
- Practice choosing a heuristic
- Practice designing a search algorithm

What to do:

1. Build a graph where the vertices are words of a certain length, and there are edges between words that are exactly one character apart.
    a. Get a list of the words in the English dictionary.
        i. You may find such a file online and read the words into a list.
    b. Ask the user for a start word and an end word. If the end word is not the same length as the start word, ask again until it is the same length.
    c. Limit the list of dictionary words to words that are of the same length as the start and end words.
    d. Make an adjacency list for the graph. (A dictionary where the keys are the vertices/words, and the values are the lists of all edges/words which are connected to that word.)
        i. For each word in the limited dictionary, find all words which are exactly one character different.
        ii. The `distance()` function in the `Levenshtein` package may be useful.
        iii. Minor optimizations will make this process run faster, such as using the argument `score_cutoff=1`.
    e. Create a class called `WordPathNode` which keeps track of a word and a parent `WordPathNode` (a linked list).
        i. Make sure to override the `__eq__()` and `__hash__()` methods.
        ii. We will define a node as the "root" if it has no parent (its `parent` is `None`). Write a method called `get_path_to_root()` which finds the path from the current node to the root, following the `parent` links. It should store the path in a list of strings and return the path.

2. Implement a function `find_edit_path_bfs(start_word: str, end_word: str)` which uses a breadth-first search to find a path from the `start_word` to the `end_word`, using only edges found in the adjacency list.
    a. It will be useful to keep track of the visited nodes.
    b. When adding a node to the frontier queue, make sure to take note of the parent in the `WordPathNode`.
    c. If it finds a solution, it should return the path as a list of words (by calling the `get_path_to_root()` method).
    d. If it is unable to find a solution, it should return None.
    e. Try it out!
3. Implement a function `find_edit_path_dfs(start_word: str, end_word: str)` which uses a **depth**-first search to find a path from the `start_word` to the `end_word`, using only edges found in the adjacency list.
    a. The same hints from the breadth-first search section also apply here.
    b. It will be a useful exercise (for interviews) to implement this recursively and also iteratively using a while loop. But, for full points on this homework grade, you only need one implementation.
    c. Try it out!
4. Implement a function `find_edit_path_iterative_deepening(start_word: str, end_word: str)` which uses **iterative deepening** to find a path from the `start_word` to the `end_word`, using only edges found in the adjacency list.
    a. This will be a series of depth-first searches. It may be useful to re-factor some of the code from the depth-first search algorithm so it can be used in both places.
    b. Try it out!
5. Implement a function `find_edit_path_A_star_search(start_word: str, end_word: str)` which uses **A* search** to find a path from the `start_word` to the `end_word`, using only edges found in the adjacency list.
    a. You will need to choose and implement a heuristic. There are several which may work, and it is fine to try multiple.
    b. Try it out!

What to turn in:
Please submit these via Gradescope:

- Your Python code for the search algorithms
- A text or pdf file with your answers to these questions:
    - For each search algorithm:
        - Is the first path that it finds guaranteed to be the shortest path?
        - How efficient (time and space) is this algorithm for finding paths between words that are fairly similar (>50% characters in common)?
        - How efficient (time and space) is this algorithm for finding paths between words that are not similar (<50% characters in common)?

        - How efficient (time and space) is this algorithm for determining that there is no path between the two words?
    - For the A* search, how did you choose an appropriate heuristic?
    - Suppose there is a set of words that you must include at some point in the path between the `start_word` and the `end_word`. The order of the words does not matter. How would you implement an algorithm that finds the shortest path from the `start_word` to the `end_word` which includes every word in the given set of words? You may describe the algorithm or provide pseudocode.
    - How long did this assignment take you? (1 sentence)
    - Whom did you work with, and how? (1 sentence each)
        - Discussing the assignment with others is encouraged, as long as you don't share the code.
    - Which resources did you use? (1 sentence each)
        - For each, please list the URL and a brief description of how it was useful.
    - A few sentences about:
        - What was the most difficult part of the assignment?
        - What was the most rewarding part of the assignment?
        - What did you learn doing the assignment?
        - Constructive and actionable suggestions for improving assignments, office hours, and class time are always welcome.