22C:16 Homework 8

Due via ICON on Wednesday, April 25th, 4:59 pm

What to submit: Your submission for this homework will consist of 4 files: (i) `degreeDistribution.py` for Problem 1, (ii) `printTriangles.py` for Problem 2, (iii) `playGameBetter.py` for Problem 3, and (iv) `playGameFaster.py` for Problem 4. Each of these Python files should start also with your name, section number and student ID appearing at the top of the file as Python comments. Also make sure that your Python code is appropriately commented. You will get no credit for this homework if your files are named differently, have a different format, or if your files are missing your information.
All four problems pertain to the program `playGame3.py` discussed in class.

1. Consider the program `playGame3.py` in which we built a "word network" in order to have our program play the *word ladder* game. Let us define the *degree* of a 5-letter word as the number of neighbors it has in the word network. Write a program that computes and outputs the *degree distribution* of the word network. More specifically, your program should output the number of words with degree 0, the number of words with degree 1, the number of words with degree 2, etc. If D is the largest degree of a word in the word network, then your program should output $D+1$ lines with Line 1 showing the number of words with degree 0, Line 2 showing the number of words with degree 1, and so on.

   You are required to compute the degree distribution using a dictionary. The keys in this dictionary will be integers in the range 0 through D and eventually the value associated with a key d will be the number of words with degree d. You should start with the program `playGame3.py` and modify it in order to obtain the solution to the current problem. You should delete from `playGame3.py` any code that is not relevant to the current problem. Your program will be evaluated not just on the basis of correctness, but also with regards to whether you defined appropriate functions or not.

   Save your program as `degreeDistribution.py` and submit it as part of your solution to the homework.

2. Again consider the "word network" that was built in `playGame3.py`. A *triangle* is three 5-letter words u, v, and w such that u and v are neighbors, v and w are neighbors, and u and w are neighbors. Write a program that outputs all word triangles to a file called `triangle.txt`. Each word triangle should appear in a separate line with one or more blanks separating consecutive words. A word triangle should appear only once in `triangle.txt`. For example, be careful to ensure that a word triangle with words u, v, and w does not appear once as `u v w` and another time as `v u w`.

   If you try generate all possible triples u, v, and w and then determine if pairs of words in this triple are neighbors, you would be doing too much work. To make your program efficient, you should start with a word u and then ask about which pairs of neighbors of u are also neighbors.

   You should start with the program `playGame3.py` and modify it in order to obtain the solution to the current problem. You should delete from `playGame3.py` any code that is not relevant to the current problem. Save your program as `printTriangles.py` and submit it as part of your solution to the homework.

3. The program `playGame3.py` does not attempt to produce a *shortest* valid chain of words from the given source word to the given target word. One way to produce a shortest chain is to find the "oldest" word in the `reached` set and process it before any of the others. By "oldest" word I mean the word that entered the `reached` set earliest, among those

that are currently in the reached set. (It may not be clear to you that this algorithm does indeed produce shortest word chains and if you wish, you should convince yourself of this by executing this algorithm by hand on small networks.) Note that in the current implementation of `playGame3.py` we pick an *arbitrary* word from the `reached` set and therefore the current implementation is not guaranteed to produce a shortest word chain. Your task for this problem is to modify `playGame3.py` to make the above described change to the algorithm so that shortest word chains are produced.

We used a dictionary to implement the `reached` set in `playGame3.py`. A dictionary is not particularly set up to be able to tell us who the oldest element in it is. So you will have to think a little bit to figure out whether you should still use a dictionary for the `reached` set, albeit modified in some manner to allow you to find the oldest word in it. Alternately, you could use a list implementation of `reached` rather than a dictionary implementation. I leave this decision to you.

Save your program as `playGameBetter.py` and submit as part of your solution to the homework.

4. On my machine the part of `playGame3.py` that reads words from `words.dat` and builds the word network as a dictionary takes about half a minute. This is because `words.dat` contains 5,757 words and `playGame3.py` evaluates all pairs of words and there are 16,568,646 pairs (i.e., more than 16.5 million pairs).

   This problem asks you to to employ a more sophisticated algorithm to significantly speedup `playGame3.py`. The main idea of the algorithm is simple: two words are neighbors if (i) they have the same first letter, but they differ in exactly one letter in the remaining 4 letters or (ii) they have different first letters but are identical in the remaining 4 letters. The word pair (`cares`, `cakes`) is of type (i) and the word pair (`bakes`, `cakes`) is of type (ii).

   To find word pairs of type (i), you should read the words into a 2-dimensional data structure, let us call this `wordMatrix`. This is a list of length 26, whose first element is the list of all words that start with the letter "a", whose second element is the list of all words that start with the letter "b" and so on. To find word pairs of type (i), we simply have to consider all pairs of words that start with the same letter and this is easy to do since these are all sitting together in one list in `wordMatrix`. Since there are far fewer pairs of words that start with the same letter, this will allow us to consider far fewer pairs than 16.5 million.

   To find word pairs of type (ii), you should read the words into a 2-dimensional data structure, let us call this `suffixes`. This is a list of length 5,757 in which each element is a length-2 list. Each word `w` in `words.dat` should appear in `suffixes` as a list `[w[1:], w[0]]`. Thus the word `house` will appear as `["ouse", "h"]`. Once the structure `suffixes` is built it can be sorted and at this point all words that are identical in their last 4 letter will appear bunched together in `suffixes`. Then to find word pairs of type (ii) you only have to compare an element in `suffixes` with other "nearby" elements. This will allow your program to consider far fewer pairs than the 16.5 million pairs considered in the original program.

   Implement this idea to build the dictionary `D` much more efficiently than in `playGame3.py`. Save your program as `playGameFaster.py` and submit as part of your solution.
