CS 655: Analyzing Sequences
Homework 6

Part 1: Implementing a simple trie

Building a trie:
In order to build a trie, first a list of words is read from a file. Each word is split into its constitutive letters. We begin by checking if there is an arc from state zero, with the first letter of the word as its input symbol. If there is no arc, a new arc is added to the trie, else we travel down the existing arcs till we find similar letters and branch out when we encounter a different letter. The state reached when a word is complete is set as a final state.

Prefix Searching:
Given a prefix, the trie is traversed till we reach a state that ends the prefix. All the outgoing arcs from this state are considered, and each arc traversed in a depth first search pattern, emitting a word whenever a final state is encountered.

Implementation:
The file containing the list of words is given as an argument while calling the program. A trie is built for these words. User is prompted to give an input prefix. The trie is searched and a list of matching words is returned. A sample terminal output is shown is figure 1 for a random set of words. I ran the program using words from `/usr/share/dict/words`, but the program took about a day to build a trie using these words. I'm not sure if it is suppose to take so long or there is something wrong in the way I coded.

For efficiently handling a series of prefix searches, we can store the state at which the previous prefix ended in a global variable, and search for the next letter entered in the arcs coming from this state. This will reduce the time taken to traverse to the current state in the trie, for every iteration. We can have a pointer pointing to the end of matching words from a particular arc. For example, if state 1 has 3 arcs- a, b and c, after searching for all words along arc a, and beginning searches in arc b, a pointer can point to this position in the list of words. Next time the user enters a letter; say we

have to go down arc b, the words between the pointer to that arc can be printed and when the user types the next character, we can go down by two states and give the list of matching words and noting the pointers again. This can reduce the number of times the trie is traversed to get matching words.

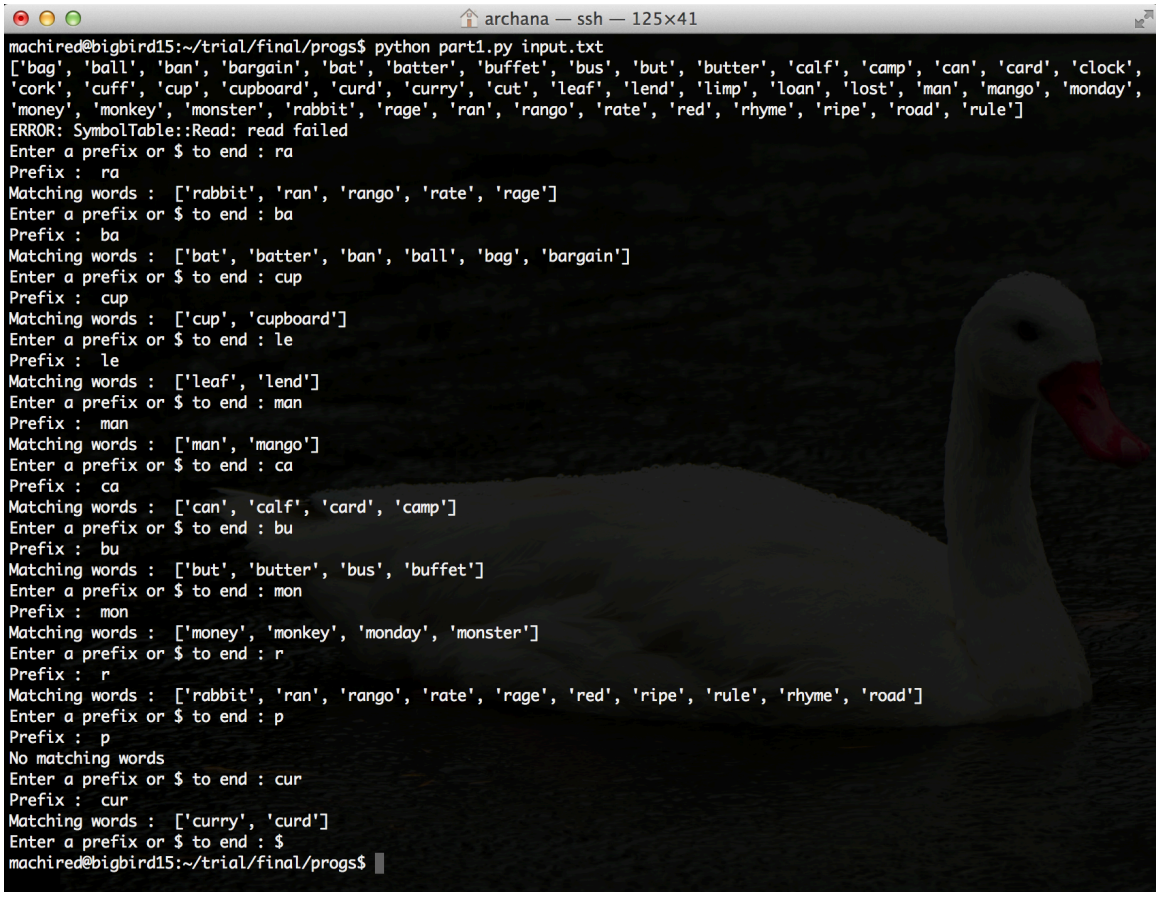

Figure 1. Sample terminal output for prefix searching

Part 2: Levenshtein Automata

Building the Levenshtein Automaton:

A word and an edit distance are given as input. For every character in the word, an arc is added for a correct match. Additionally arcs for deletion, insertion and substitution are added. The edit distance gives number of insertions, deletions or substitutions allowed. Case changes are considered as edits.

Searching:

Words are searched by manually walking down the Levenshtein automaton and trie. For a given state, the arcs common in both Levenshtein automaton and trie are considered. Then the path down these arcs is traversed again looking for similar arcs at each state. If both Levenshtein automaton and trie reach the final state, then that word is emitted as a similar word.

Implementation:

The file containing the words to build the trie is given as an argument while calling the program. The user is then prompted to enter a word and an edit distance. Words from the input file within edit distance from the given word are displayed.

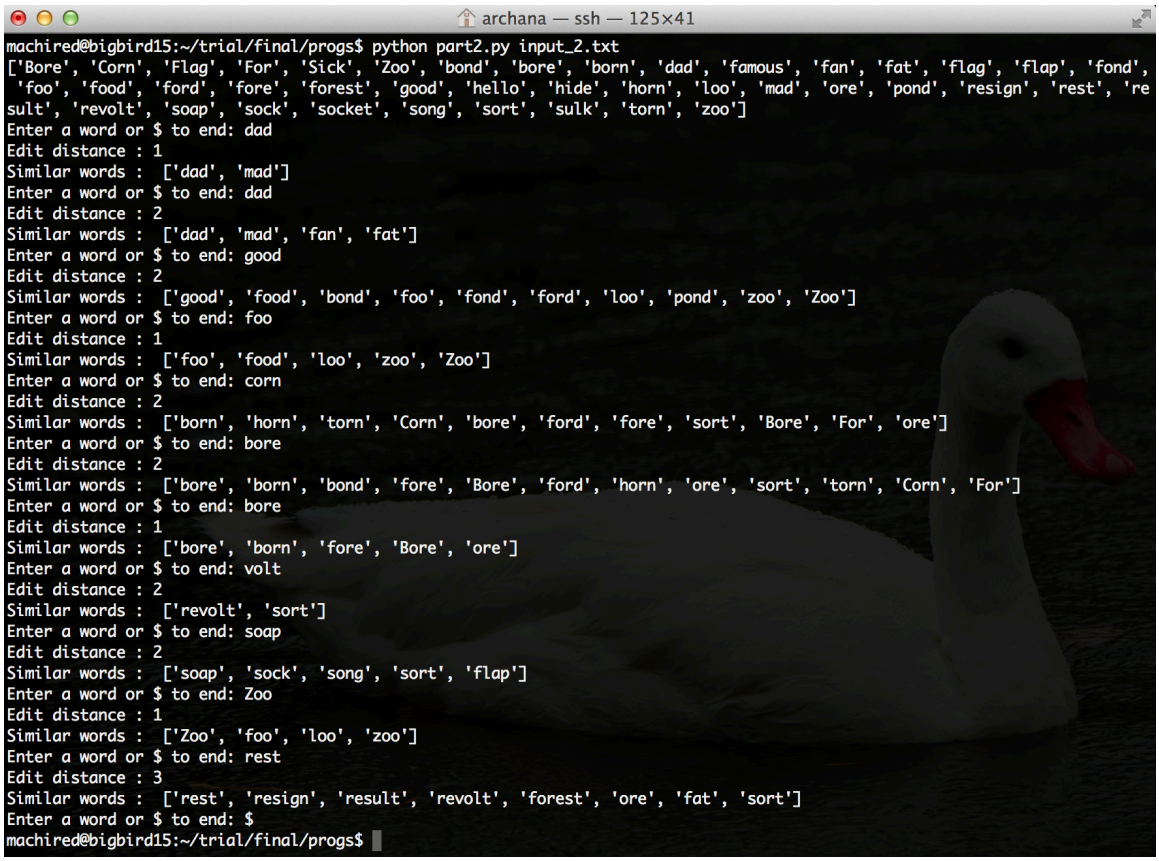

Figure 2. Sample terminal output for part 2

Part 3: Extending Part2

In order to sort the list of matching words, the error arcs of the automaton are given weights. A regular arc is given a weight of 10. Additional arcs with lesser weights are added for particular situations. The different weighting schemes considered are:

1. Swapping of ‘e’ and ‘i’ are given a weight of 2 as they are swapped frequently
2. Swapping of vowel with a vowel is given a weight of 5, as swapping a vowel for a voxel is more common than swapping a vowel for a consonant. Some particular vowel swaps are more common, like ‘a’ for ‘e’, which are given a weight of 3. Swapping ‘i’ and ‘o’ is given a weight of 3, as these two letters are next to each other and may cause a typographical error very often.
3. Substitution for phonetically similar letters cost less. Substitution of ‘c’ by ‘k’ or ‘s’ is given a weight 3.
4. Swapping a consonant with a consonant is given a weight 8, as consonants are most commonly mistyped as another consonant rather than a vowel.
5. Insertion of ‘s’ at the end is given a weight 2 as it is more common than inserting any other letter.
6. Insertion of a vowel after a vowel is more probable, therefore is given a weight 6.
7. Missing a letter from a word with repeated letter is very common. For example, in words like embarrass, it is common to forget r or s twice. So insertion of a letter similar to a given letter is given a cost of 6.

While traversing through the Levenshtein automaton to find similar matches, the weight along each path is added as we go and is recorded along with the final word. The difference in weights for arcs in different situations, gives varying weights for the words. The words are sorted by weights, and displayed from the least expensive path to the most expensive path. The result gained by reordering the matching words is shown in figure 3.

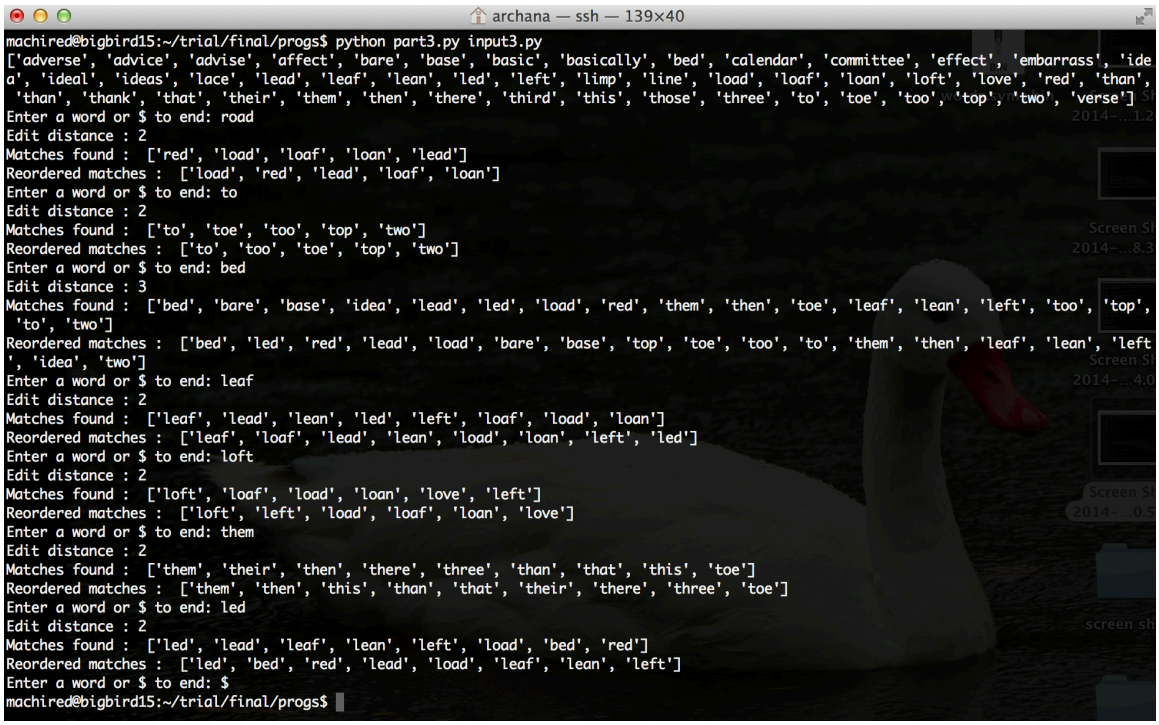

Fig. 3. Sample terminal output for part 3

From the examples above, it can be seen that the difference in weights can help improve the ordering of results. For example leaf is more likely to be confused with loaf (swapping of vowels) rather than other words in the list of matching words. In the same list loan and load are closer to leaf than led. For the input loft, left which is most likely to be a correct match is given at the end of the list in initial matches. By applying the weights, it is given as the first choice, as swapping of vowels is given lesser weight. Similarly, for the word ‘them’, then, than, this are better suggestions than their, there and three. As for word ‘led’, though the same distance away load is a better suggestion than leaf. This is a result of giving lesser weight to addition of vowel after vowel. Overall, in most cases, sorting by the weights of error arcs results in a better suggestion list.