Autocomplete

Submission to website: Monday, October 24, 10pm

Checkoff by LA/TA: Thursday, October 27, 10pm

This lab assumes you have Python 2.7 installed on your machine. Please use the Chrome web browser.

Introduction

Type "aren't you" into Google search and you'll get a handful of search suggestions, ranging from "aren't you clever?" to "aren't you a little short for a stormtrooper?". If you've ever done a Google search, you've probably seen an autocompletion - a handy list of words that pops up under your search, guessing at what you were about to type.

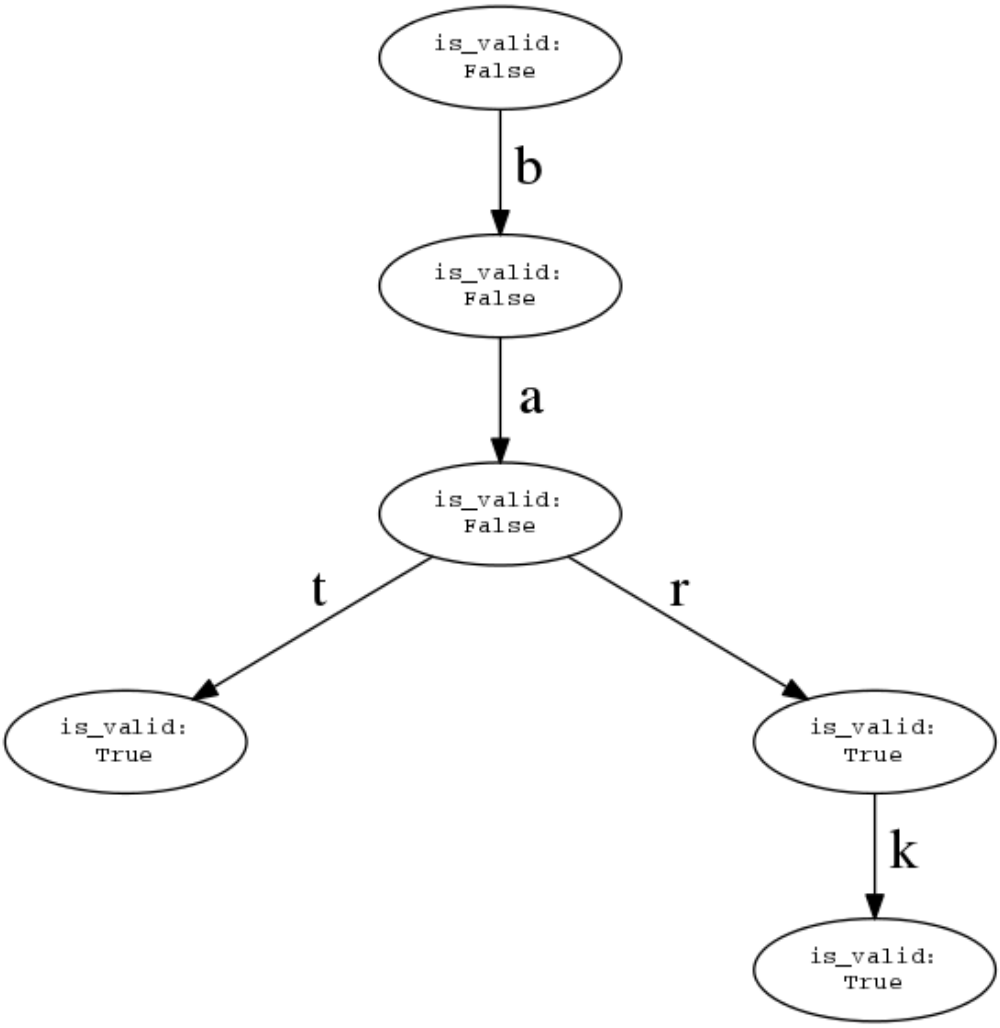
Search engines aren't the only place you'll find this mechanism. Powerful code editors, like Eclipse and Visual Studio, use autocomplete to make the process of coding more efficient by offering suggestions for completing long function or variable names.

In Lab 6, we are going to implement our own version of an autocomplete engine using a tree structure called a "trie," as described in this document. The staff have already found a nice *corpus* (list of words) for you to use - the full text of Jules Verne's "In the Year 2889." The lab will ask you first to generate the trie data structure using the list of words provided. You will then use the trie to write your own autocomplete, which selects the top few words that a user is likely to be typing.

The trie data structure

A trie, also known as a *prefix tree*, is a type of search tree that stores words organized by their *prefixes* (their first characters), with longer prefixes given by successive levels of the tree. Each node is a Boolean (`true` / `false`) value stating whether this node's prefix is a word.

For example, consider the words `'bat'`, `'bar'`, and `'bark'`. A trie over these words would look like the following:

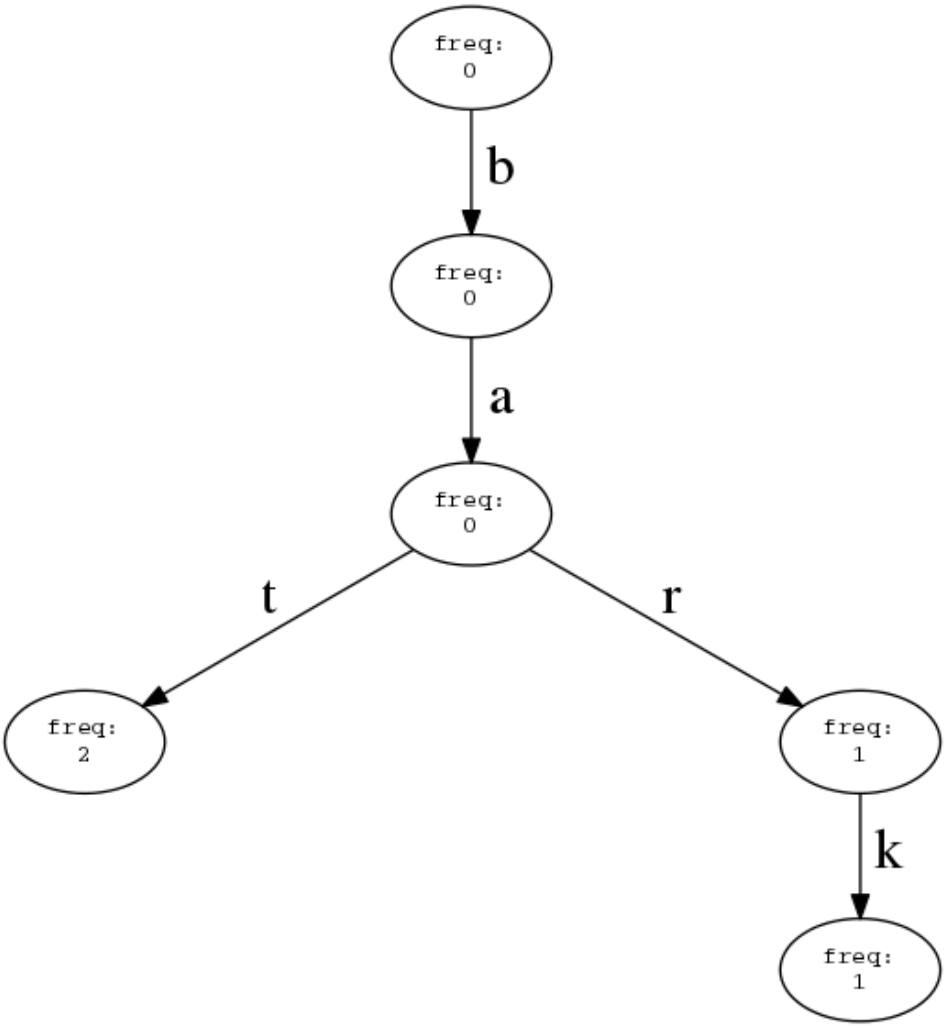


To list all words beginning with `'ba'`, begin at the root and follow the `'b'` and `'a'` edges to reach the node representing a `'ba'` prefix. This node is itself a trie and contains all words prefixed by `'ba'`. Enumerating all paths leading to `true` nodes (in this case, `'t'`, `'r'`, `'rk'`) produces the list of `'ba'` words: `'bat'`, `'bar'`, and `'bark'`.

Note that we also check the `'ba'` node itself, though in this case the node is `false`, meaning `'ba'` is not known to be a word. Consider the words beginning with the string `'bar'`. Just as before, follow the `'b'`, `'a'`, and `'r'` edges to the `'bar'` node, then enumerate all paths to `true` nodes (`''` and `'k'`) to find the valid words: `'bar'` and `'bark'`.

This trie structure on its own, however, is not very useful. If we type only a few characters, for example `'b'`, the long list of words `b` generates is of little help to the user, who is only interested in the most likely candidates. To this end, we **augment** the nodes of our trie with a *frequency* metric, describing how often each word appears in our corpus. Assume that the more often a word appears in the corpus, the more likely it is to be typed by our user. When using the trie to enumerate likely words, suggest only a few likely matches instead of the entire list.

Consider the following corpus: `"bat bark bat bar"`. The trie this corpus would generate is:



Assume we are interested in only the top result after autocompleting the string `'ba'`. Now instead of giving us all of `'bat'`, `'bark'`, and `'bar'`, we should just get the highest-frequency word - `'bat'`.

Note that in the tree above, the `'b'` and `'ba'` nodes have frequencies of `0`, meaning they're not valid words.

lab.py

In `lab.py`, you are responsible for three functions, `generate_trie`, `autocomplete`, and `autocorrect`.

`generate_trie` will take as input `words`, a list of words representing a corpus, (e.g. `["bat", "bark", "bat", "bar"]`) and should return a trie representing that list of words. `words` is guaranteed to contain words consisting only of lowercase letters.

A trie is a tree where each node (dictionary) has:

- `frequency`: an integer frequency (number of times the word is seen in the corpus) of the word *ending at that node*.

- `children`: a dictionary that maps suffix characters (strings) to other tries (tries are a recursive data structure).

`generate_trie` should return the root trie, which should, via `children` links, cover all words in the corpus. Note that the root trie represents the empty string `""`, so it should always have a frequency of 0, as it is not a word.

Note that, to pass our test cases, the tries you generate should be in *exactly the format documented above, with no extra fields in dictionaries, etc.* See the class examples of linked lists and binary search trees for the basic sort of dictionary-based encoding that we require.

`autocomplete` takes as input:

- `trie`, the trie that you generated in `generate_trie`
- `prefix`, the prefix string you are trying to autocomplete
- `N`, how many words of autocomplete you want.

That is, given `trie`, `prefix`, and `N`, you should return a list of the most-frequently-occurring `N` words in `trie` that start with `prefix`. In the case of a tie, you may output any of the most-frequently-occurring words. If there are fewer than `N` valid words available starting with `prefix`, return only as many as there are. The returned list may be in any order.

Autocorrect

You may have noticed that for some words, our autocomplete implementation generates very few or no suggestions. In cases such as these, we may want to guess that the user mistyped something in the original word. We ask you to implement a more sophisticated code-editing tool: autocorrect.

The `autocorrect` method in `lab.py` takes in `trie`, `prefix`, and `N`, just like `autocomplete`. However, its behavior is slightly more interesting.

`autocorrect` should invoke `autocomplete`, but if fewer than `N` completions are made, suggest additional words by applying one *valid edit* to the prefix.

An *edit* for a word can be any one of the following:

- A single-character insertion (add any one character in the range `a-z` at any place in the word)
- A single-character deletion (remove any one character from the word)
- A single-character replacement (replace any one character in the word with a character in the range `a-z`)
- A two-character transpose (switch the positions of any two characters in the word)

A *valid edit* is an edit that **leads to a word in the corpus**. Editing `"te"` to `"the"` is *valid*, but `"te"` to `"tze"` is probably not, as "tze" isn't a word. Likewise, editing `"phe"` to `"the"` is *valid*, but `"phe"` to `"pho"` is not because "pho" is not a word in the corpus, although many words beginning with "pho" are.

In summary, given a prefix that produces C *completions*, where C < N, generate up to N-C additional words by considering *all valid single edits* of that prefix (i.e., corpus words that can be generated by 1 edit of the original prefix), and selecting the most-frequently-occurring words. Return a list of suggestions produced by including *all* C of the completions and up to N-C of the most-frequently-occuring valid edits of the prefix; the list may be in any order. Be careful not to repeat suggested words!

Hints

We include a file `hints.pdf` that you may find useful if you find yourself stuck on this lab.

Testing your lab

We've included a 6.009-autocomplete-powered search bar so you can see your code in action. Run `server.py` and open your browser to localhost:8000 and type into the search bar to see the top 5 results from your `autocomplete` function, using the corpus of words from Jules Verne's "In the Year 2889." As in the previous labs, we provide you with a `test.py` script to help you verify the correctness of your code. The script has tests for both `generate_trie` and `autocomplete`, but the `autocomplete` tests are dependent on a successful `generate_trie` implementation. Consider implementing `generate_trie` first and making sure its tests pass before moving on to `autocomplete`.

Does your lab work? Do all tests in `test.py` pass? You're done! Submit your `lab.py` on funprog and get your lab checked off by a friendly staff member.