Lecture Notes, CSE 232, Fall 2014 Semester

Dr. Brett Olsen

Week 4 - Strings

OK, today we're going to be talking about strings, string processing, string data structures, and some important string algorithms that you'll run into a lot.

String Processing So there's a number of basic string processing tasks that I more or less assume you can handle as a part of basic fluency in your programming language of choice: reading strings in from a file, splitting strings into tokens by whitespace, translation of characters in a string from one mapping to another (e.g., upper to lower case), finding a short substring in a longer string, and so on. For most of these basic tasks, there are library functions that can handle them, and so it's not necessary to go into details.

Frequency Counting A common subtask you'll see for a number of string algorithms is the need to keep count of how often a particular character or word shows up in our input. We talked about a good data structure for this task last week: a dictionary where the keys are the character or word and the element is the frequency count for each key.

String Data Structures Alright, last week we talked about general data structures. There are a couple of related data structures that are optimized for manipulations and processing of strings and can be used to make important algorithms much faster.

Trie The first is called a "trie", pronounced just like "tree" (and yes, this is confusing - the name comes from re-"trie"-val). Here's what it looks like:

Suppose we have a list of words that we need to store for later use - let's suppose we're doing the word split calculation I talked about last week, where we have a long string that needs to be split into words from the list. Let's use this example list:

```
pea
peace
peas
not
note
nut
```

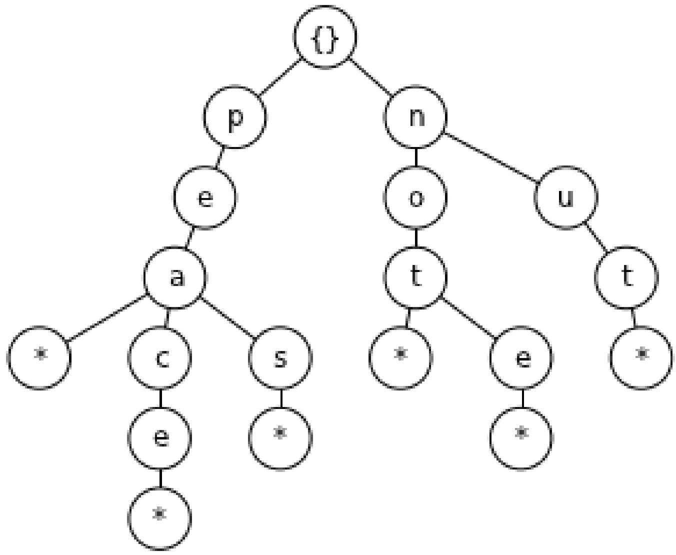
And our target string will be `peanut`. So last week we talked about a way to solve this problem using a dictionary. We add our words to a dictionary and then loop through all possible splits of the target string until we find one where both of the split words are in our dictionary:

```
p eanut
pe anut
pea nut
pean ut
peanu t
```

There are five different splits possible and two words to check for each split, so we'll need to do 10 dictionary lookups.

A trie will let us do this faster. A trie is a type of tree where each string in our list is represented by a path of nodes from the root to a terminator:

In [89]:
```
import networkx as nx

words = ['pea', 'peace', 'peas', 'not', 'note', 'nut']

G = nx.DiGraph()

labels = {}

G.add_node('ROOT'); labels['ROOT'] = '{}'

def counter():
    n = 0
    while True:
        yield n
        n += 1

def add_word(root, word, counter, suffix='*'):
    idx = counter.next()
    if word == '':
        G.add_node(idx)
        G.add_edge(root, idx)
        labels[idx] = suffix
    else:
        keys = G[root].keys()
        children = [labels[key] for key in keys]
        if not word[0] in children:
            G.add_node(idx)
            G.add_edge(root, idx)
            labels[idx] = word[0]
        key = [key for key in G[root].keys() if labels[key] == word[0]]
        add_word(key[0], word[1:], counter)

C = counter()
for word in words:
    add_word('ROOT', word, C)

nx.draw_graphviz(G, prog='dot', node_color='white', labels=labels, node_size=700, arrows=False
```



Our dictionary method required us to do 10 dictionary lookups to check for the correct split. How can we do the same operation with our trie? Well, let's start with our input string `peanut`. We can follow a path in our trie along the characters in the input string until we either run out of options or hit a terminator vertex, which will mark a valid prefix of our input string that is in our word list. So the first character is `p`, which is a child of the root node, so we move to that child. That node doesn't have a terminator, so it isn't in the dictionary and we can't split here. So we take the next character `e`. It's a child of `p`, so we move there. Again, there's no terminator, so `pe` isn't a valid prefix. The next character `a` is a child, so we move to that node. We do find a terminator as a child of `a`, indicating that `pea` is in our dictionary and a valid prefix of our input string. The next character `n` isn't a child of `a`, so there are no other possible valid prefixes. Then we check the remainder of the string `nut` to see if it's in our dictionary and we find the path `nut*` in our trie, so we've found a valid splitting. This gives us a valid split of the input string in fewer dictionary checks than we had to do when we were explicitly testing splits that couldn't have worked, while also using less memory to store the word list.

However, this structure particularly shines with more complex problems. Suppose that instead of finding a valid split we wanted to check if a valid split exists at all. Given an input string where there is no good split, say `pesnut`, we must necessarily check all possible splits before confirming that none of them are valid. However, using the trie, we can find this much faster. First we walk down the first two characters `p` and `e`. We haven't yet hit a terminator, but the next character `s` isn't a child of `e`, so there are no valid prefixes for this string. That gives us our answer much faster than we could possibly get in using a simple dictionary.

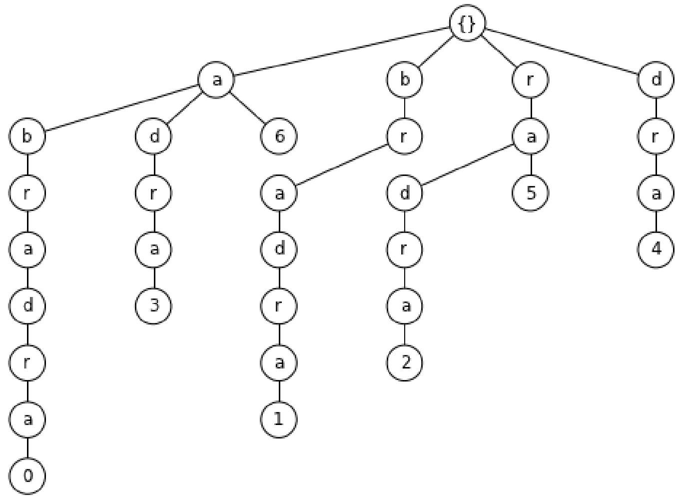
Let's look at an even more complicated problem. Suppose that we have a string S and we want to find out, not if it can be split into *two* words in our list, but if it can be split into an *arbitrary* number of words in our list. How would we do this? Well, with only two possible words, there are $N-1$ possible splits, which is feasible to calculate with a complete search. But with an arbitrary number of words, there are 2^{N-1} possible splits, which rapidly becomes far too large for any kind of complete search. We can use a combination of our trie structure and dynamic programming to solve this.

Suppose our target string is `peasnutpea`. We'll start with the same strategy as before, by walking down our trie from the root following the characters in the target string. We'll get to a prefix of `pea` before we hit a terminator, indicating that this is a valid prefix. Continuing down, we can also find another valid prefix of `peas` with a terminator, after which we can't continue. Because these are the only two possible prefixes of valid word splits, then the target string is splittable if and only if at least one of the suffixes remaining (`snutpea` and `nutpea`) are also splittable. But these are smaller instances of our original problem, showing that we have optimal substructure. The different subproblems we can find are all of the suffixes of S, and the correct solution for each suffix depends on the solutions to the smaller suffixes. Then we can construct a table marking whether the suffix beginning at a particular character is splittable using our trie and calculate the result based on this table in $O(N)$.

```
p e a s n u t p e a
T F F F T F F T F F
```

Suffix Trie A suffix trie is a trie constructed from words that are the suffixes of a single longer string S where we've then modified the terminators to mark which suffix each terminator ends. Let's look at a suffix trie constructed on an example string `abradra`:

In [90]:
```
import networkx as nx
import matplotlib.pyplot as plt

S = "abradra"

G = nx.DiGraph()

labels = {}
G.add_node('ROOT'); labels['ROOT'] = '{}'

def counter():
    n = 0
    while True:
        yield n
        n += 1

def add_word(root, word, counter, suffix='*'):
    idx = counter.next()
    if word == '':
        G.add_node(idx)
        G.add_edge(root, idx)
        labels[idx] = suffix
    else:
        keys = G[root].keys()
        children = [labels[key] for key in keys]
        if not word[0] in children:
            G.add_node(idx)
            G.add_edge(root, idx)
            labels[idx] = word[0]
        key = [key for key in G[root].keys() if labels[key] == word[0]]
        add_word(key[0], word[1:], counter, suffix=suffix)

C = counter()
for i in range(len(S)):
    add_word('ROOT', S[i:], C, suffix=i)

plt.figure(figsize=(10,6))
nx.draw_graphviz(G, prog='dot', node_color='white', labels=labels, node_size=600, arrows=False
```



Uses of Tries OK, what can we do with tries and suffix tries?

String Matching String matching - trying to find all locations of a (small) substring P of size m in a (larger) string S of size n - is a classic problem, which comes up very often in a wide variety of contexts. A naive brute-force implementation would try to match each character in P to one in S by starting from each character in S, giving an $O(nm)$ algorithm. An better algorithm (called the Knuth-Morris-Pratt algorithm, if you'd like to look it up in more detail) can provide $O(n+m)$ performance, which is obviously much better for longer search strings. But let's think for a minute about the case where we have a fixes S but a large number of searches k with different substrings P. This is the case for say, a number of bioinformatics problems, where the large string is a section of the genome that we want to search for lots of different, say, sequencing reads. These naive algorithms will then have to run independently for each new search, taking at best $O(kn+km)$ time. But suppose we can take some time ahead of time to construct a suffix trie for S (which will take from $O(n^2)$ to $O(n)$ depending on the method) that we can then store for later searches. Let's look at how to find all substrings `ra` in `abradra` using this trie. Well, every substring of S is a prefix of some suffix of S and we know that tries can be used to rapidly find prefixes of words in our trie. That means that every occurence of P in S can be found by simply walking down the path dictated by its characters in the suffix trie of S. In this case, we'll walk down to `ra` and the leaf nodes that are children of that final `a` node tell us the starting position of each instance of the substring: 2 and 5.

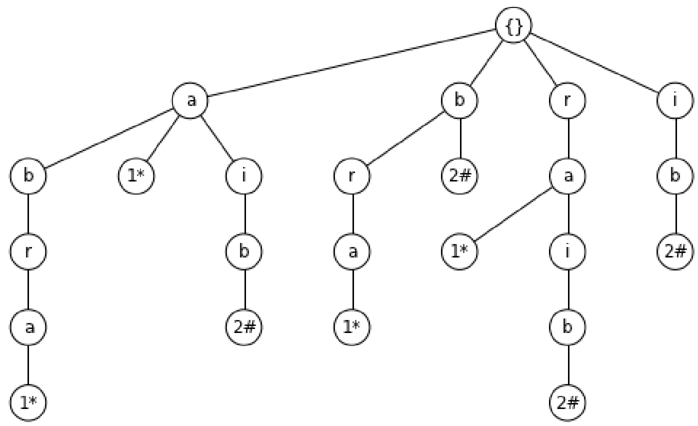
This search method takes only $O(m+occ)$ time once the trie is already constructed - we have to walk through each character in P and then count how many times it occurs, and the search time does not vary with the size of S.

This approach is great for competition problems where you have the search strings ahead of time, before the input data. Say there's a dictionary of words provided and you need to rapidly search them. Construct the trie before you submit the code, and then only the searches are included in your time limits.

Longest Repeated Substring Suppose we want to identify the longest substring of S that occurs more than once. We can do that with our suffix trie by looking for the *deepest internal* node, that is, the node furthest from the root which has more than one child. Looking at our trie, there are two internal nodes, marking the substrings `ra` and `a`, with the deepest node marking `ra`.

Longest Common Substring Suppose we have two strings $S_1 =$ `abra` and $S_2 =$ `raib`, and we want to find the longest substring that is shared between them. Let's build a shared suffix trie on both input strings, while changing the terminator node to mark which string each substring comes from:

In [92]:
```
import networkx as nx
import matplotlib.pyplot as plt

S1 = "abra"
S2 = "raib"

G = nx.DiGraph()

labels = {}
G.add_node('ROOT'); labels['ROOT'] = '{}'

def counter():
    n = 0
    while True:
        yield n
        n += 1

def add_word(root, word, counter, suffix='*'):
    idx = counter.next()
    if word == '':
        G.add_node(idx)
        G.add_edge(root, idx)
        labels[idx] = suffix
    else:
        keys = G[root].keys()
        children = [labels[key] for key in keys]
        if not word[0] in children:
            G.add_node(idx)
            G.add_edge(root, idx)
            labels[idx] = word[0]
        key = [key for key in G[root].keys() if labels[key] == word[0]]
        add_word(key[0], word[1:], counter, suffix=suffix)

C = counter()
for i in range(len(S1)):
    add_word('ROOT', S1[i:], C, suffix='1*')
for i in range(len(S2)):
    add_word('ROOT', S2[i:], C, suffix='2#')

plt.figure(figsize=(10,6))
nx.draw_graphviz(G, prog='dot', node_color='white', labels=labels, node_size=600, arrows=False
```



Now we can find shared substrings between the two strings: any node that has descendant terminators from both strings marks a substring of both $S1$ and $S2$. The deepest of these nodes is the longest shared substring, in this case `ra`.

Extensions of tries (Supplementary) These basic tries are sufficient to handle most string processing problems, but there are a couple of refinements or extensions that you can find in textbooks that have improved performance on some tasks, at the cost of being significantly more difficult to implement and understand. I won't cover them in detail, but so that you can look them up if you're interested:

Suffix *trees* are compressed versions of a suffix *tries* where paths that don't split (multiple vertices in a row with only a single child) are compressed down to a single vertex. These save space and CPU time in traversing and constructing the tree, at the cost of some bookkeeping, but are otherwise identical.

Suffix *arrays* are essentially a way of representing a suffix tree in a more condensed format as an array of suffixes rather than with explicit nodes and pointers. They're sufficiently more efficient that they're worth using for most applications, particularly if you write a library to handle them in advance of a contest, but are also sufficiently complicated to analyze that I'm going to leave them out of the lecture. They're discussed in detail in Competitive Programming 3 if you're interested.

Fast construction of suffix trees (Supplementary) A naive suffix tree construction algorithm takes $O(n^2)$ time to create the initial tree. There is a faster $O(n)$ construction algorithm that is significantly more complicated. For most cases in a programming contest, an $O(n^2)$ construction will be OK, but it's worth learning about the faster method if you can. Similarly, there's a good $O(nlgn)$ construction for suffix *arrays* that you can find in Competitive Programming 3.

See http://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english/9513423#9513423 and http://marknelson.us/1996/08/01/suffix-trees/

Lab Section The problem this week is the homework question, based on a question from one of the Google Code Jam rounds:

- GCJ 1B 2013 - Garbled Email