Handout 5 – Web Indexer Part 2

[…]

So, what did our index look like in the previous part? Well, index was a dictionary that mapped words to a list of sites that had that word:

"mit" --> ["web.mit.edu ", "csail.mit.edu", ...]
"google" --> ["web.mit.edu", "google.com", ...]
...
word --> [*site1*, *site2*, *site3*, ...]

And let's get straight what our types were:

index - dictionary, keys are words, values are lists of sites
index[word] - list of sites

So when we searched for a word, we got back something that looked like this:

[*site1*, *site2*, *site3*, ...]

Now, we want to retrieve not just a list of sites, but also how many times the word appeared on each of those sites (the frequencies). We want something that looks like this:

[(*site1*, *freq1*), (*site2*, *freq2*), (*site3*, *freq3*), ...]

So what we can do to get that instead of what we had before?

One thing we can do is change our index so that the values are lists of (site, freq) tuples. So now index will look like:

"mit" --> [("web.mit.edu", 3), ("csail.mit.edu", 2), ...]
"google" --> [(" web.mit.edu", 1), ("google.com", 3), ...]
...
word --> [(*site1*, *freq1*), (*site2*, *freq2*), (*site3* , *freq3*), ...]

Let's stick with this idea for now, and we'll see soon why we can't do this.

How do we implement the index_site function? Remember, we're given a site (like " web.mit.edu") and the text on that site (one giant string). As usual, we're going to split it into a list of words. Then, we're going to look at each word, and update the index for that word.

So, last time, if we hadn't seen the word (if word was not in the index), then we had to create a new list, with this site inside. So this time, we'll create a new list with a (site, freq) tuple inside. What's the frequency? Well, if this is the first time we're seeing the word, it's 1. So the tuple will be (site, 1).

If we HAVE seen the word, then last time, we also checked if we had already included the site (if site was in the list index[word]). But what do we do this time? Because now index[word] is no longer a list of sites, it's a list of (site, freq) tuples.

We could check if (site, 1) is in the list. But what if the word's frequency in the site is more than 1? We don't want to check for a specific frequency, we want to check for only the site. And we have no way of doing that without looping through the list index[word].

So this is why we want to use a dictionary -- so that it's easier, and we don't have to loop through a list to find the (site, freq) tuple we need to use.

Okay, so let's back up. So what will each word map to in the index? Another dictionary. What will the keys of this dictionary be? The sites. And the values? The frequencies. So now the index will look something like this:

"mit" --> { "web.mit.edu": 3, "csail.mit.edu": 2, ... }
"google" --> { "web.mit.edu": 1, "google.com": 3, ... }
...
word --> { *site1 : freq1*, *site2* : *freq2*, *site3 : freq3*, ... }

And let's get straight what our types are:

index - dictionary, keys are words, values are more dictionaries
index[word] - dictionary, keys are sites, values are frequencies
index[word][site] - int , the frequency of that word in that site

So let's get back to index_site. We're going to go through each word, and what should we do? If we haven't seen that word before, then instead of making a new list like we did last time, we'll make a new dictionary. What should that dictionary contain? This site-freq entry. And again, the frequency initially is 1, since we're seeing this word for the first time.

If we HAVE seen the word, then we know that the dictionary index[word] exists. Last time, this was a list, and we checked if site was already in the list. In this case, we're going to check if the key site is already in the dictionary.

Last time, if it WAS already in the list, we ignored it. This time, if it's already in the dictionary, should we ignore it? No, we should increment its frequency.

Last time, if it WASN'T already in the list, we appended it. This time, if it's not already in the dictionary, do we append it? No, there's no such thing as append for dictionaries. We'll add a new entry for it instead. The site is obvious, but what's the frequency? Again, if this is the first time we're seeing this word on THIS site, then the frequency of the word is 1 for this site. So we'll make a new entry for this site mapping to 1.

So we have three cases:

Case 1 -- we haven't seen the word on any site, i.e. it's not in the index. In this case, we make a new entry in index for word. And its value is a dictionary containing the single mapping site to 1.

Case 2 -- we've seen the word on another site, but no on this site yet, i.e. index[word] exists, but this site is not in index[word]. In this case, we want to make a new entry to index[word] for this site. Its value is 1.

Case 3 -- neither of the above, i.e. we've seen the word on this already, so index[word][site] exists. In this case, we want to increment the frequency of the word on this site, so we increment index[word][site].

Great, so we've taken care of the indexing. So now, search_single_word. This should again be trivial like last time, but there's just one catch. What is index[word]? It's a dictionary of site-freq entries. What do we need to return? A list of (site, freq) tuples. Perfect for the items function! So index[word].items() will return what we need. And remember to return the empty list again if the word isn't part of the index.

Finally, search_multiple_words. This is a little tricky. So last time, what we did was we made a new list, initially an empty list, then we went through each word and called search_single_word on it. That function returned a list of sites, so we went through each site, and if it wasn't already included in our new list, we appended it.

This time, we're going to apply the same big idea, but with a few changes. We're still going to go word by word and call search_single_word on each one. But now, what does search_single_word return? It returns a list of (site, freq) tuples. And we need to return a similar list of (site, freq) tuples.

The only difference is, if we see a site more than once, we don't want to ignore it this time. Instead, we want to return the sum of all its frequencies. (e.g. if we search for "mit people", the word "mit" may appear 3 times on " web.mit.edu", and "people" 1 time on the same site, so we'd return ("web.mit.edu", 4) because there were 4 hits).

So, for similar reasoning above, we don't want to maintain a LIST like last time (of (site, freq) tuples like last time). instead, we want to maintain a dictionary. So we'll set a variable to an empty dictionary initially. The keys of this dictionary will be each site that we've seen, and the values will be the total frequency of each site.

So now, we're going to go through each word, like we said, search_single_word on it. This returns a list of (site, freq) tuples. Now we'll see if the current site is in our dictionary. If it is, we're going to increase the site's frequency by freq, and if it's not, we're going to make a new entry in the dictionary, where site maps to freq.

Finally, we need to return a list of (site, freq) tuples again, so remember to call items() on the dictionary like last time.

Oh, one last thing -- your search engine will work now, but you'll see that the results aren't sorted. You'll see a site with 5 hits, then 1 hit, then 3 hits, etc.

To sort it, remember to follow the instructions in the lab: before you return the dictionary.items() list in both your search functions, assign it to a variable, let's say L, then call L.sort(...) like it shows in the lab. Then, return L. Note that sort doesn't return a new list, it modifies the list.

Great, you should have a working search engine now. Try using "mitsites20.txt" or "mitsites50.txt" to see some cool results, and compare against Google. =) Just know that 20 will take a little more time to build the index, and 50 will take a decent amount of time.

Also, it's best to not include words like "and" and "the" in your search, because almost every site will probably have those words.

[…]