Homework 3 - A-6.9 Solution

A-6.9

Word Cluster diagram is a figure that represents the words with a size proportional to its frequency in a speech or document. You can see examples of such diagrams if you search for "word cluster diagram" and there are applications online that creates the cluster for a given document. This involves calculating the frequency of each word in a speech/document as an important part of the preprocessing for the visualization tool.

One brute force approach is to scan the document and for each word calculate the frequency of each word inserted in a table or a (balanced) binary search tree whose entries/nodes consists of the word w (the key) and the frequency f of the word. We can represent each word by an array of size m. This would take an average of $O(nm \log n)$.

The next step would be to assign a font number to each word, where higher frequencies get larger fonts and less frequencies get smaller fonts. So each word has to be visited and a font number be assigned based on the frequency. This operation takes O(n). Over all this approach takes $O(nm \log n)$ time on average. Let see if we can improve the timing.

First let's understand the data and its usage. The Word Cluster Diagram problem does not require an absolute accurate frequency count. In fact, a few missed count for a word does not impact the diagram and its visual use in anyway.

Let's see how we use can use a hash table, where the key will represent the word itself and its value will represent the frequency of that word. The basic approach is to use the characters in the string to compute an integer, and then take the integer and use a Universal Hash Function and it to a table of size N. So we first need to convert the string keys to integers and then map them into the range of [0, N-1] for some fixed N.

Option 1: Use a hash function to use the first 10 integer values of the characters in the word. Note that the average length of an English word is 5.1 characters (just googled it!). So the keys are string characters of 10 ASCII uppercase letters; if we solve it for uppercase letters, we can generalize it to both lowercase and uppercase letters). Recall that a hash function takes a key and return an index into a Hash Table.

There are 26^{10} possible keys and an array of this size will hardly fit in any laptop memory. We may be better off to sum up these ten integers. The ASCII codes for these characters are in the range of 65-95, and so the sum of 10 characters would be between 650 to 950. So we may choose N=300. (Note that 650-950 = 300). To map each value to range 0 to 299, we subtract 650 from each value. Here 299 is not a prime number, but it is a 2-prime number, being the product of 13x23.This seems like a feasible option from memory point of view and collision avoidance. We could pick a prime number larger than 300, but as we said above we don't need to be that precise. This idea for this option comes from Section 6.2.1, Summing Components, and there are more alternatives there.

Option 2: Use a Polynomial- Evaluation Function (Section 6.2.2), where the key is the 10 first characters of the word. We can choose the value a=31, which is prime and in case we include both uppercase and lower case we choose a=53.

Option 3: Use the last two 16-bit of the word and form a 32-bit integer, similar to Option 1.

With any of these options, we can then use a modulus operator (sections 6.24, 6.25, or 6.5 Universal Hashing Function) by selecting a prime number (less number of collisions) greater than *N* as the hash function to map our keys to an index in an array.

Now we scan through the sequence of *n* character strings and perform a lookup in the hash Table for the current word. If we find an entry for the current word key, then we update its *frequency* to a new value '$frequency + 1$'. If we don't find an entry, we insert a new entry with $frequency = 1$.

The next issue is how we address collision. We can use a collision handling scheme like separate chaining, linear probing or cuckoo hashing for our hash map. Any one of them works very well here. Since we are interested in the first 200 to 300 words for visualization, we can even skip any collision handling and just increment the frequency of the existing word by 1, in particular if the parser provided meaningful words only; and has eliminated the propositions, etc. Cuckoo Hashing will provide the most accurate though. In all these approaches we get an amortized $O(1)$ to perform a single insertion and updating the frequency.

Since we are going through a sequence of *n* words once, the running time of this algorithm will be $O(n)$.