EC504 Project Report: Cache Sync Project

Austin Schiller, Larry Sun, Everett Carson

Data Structure Analysis

To organize and compress the dictionary of search terms, we decided to build a trie. A trie is a tree like data structure whose keys are letters. Navigating through the trie from the root to a leaf will result in a complete term, with each leaf representing a different term. The trie combines words with the same beginnings, allowing a large number of words to be stored efficiently. A visualization of how our trie is built can be seen below.

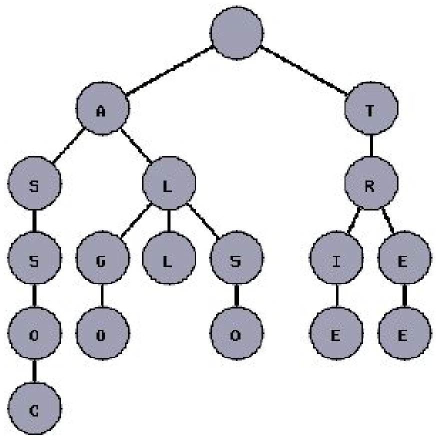

figure 1. A simple Trie

Each individual node is also an object we created. Firstly, the node object has a character attribute called letter. It also has a linked list of node objects that it fills with its children. The application which we created these objects for also requires us to have some nonstandard attributes. Each node has a integer attribute called frequency which corresponds to the popularity of the search term that the node belongs to. A node only has a nonzero frequency if it is the last letter of its corresponding term. This frequency characteristic is also what we use to determine if an in-line words being present in the trie, such as in the case of 'test' and 'testing.'

Traversing the trie was somewhat confusing to implement, but ultimately made easy with recursion. For example, predicting the user's query based on his or her input required us to compare the frequencies of the all the children of a given node. In order to do this, we recursively travel through the children of that node. If any of its children, grandchildren, etc. have a nonzero frequency, we compare it to a list of suggested queries, which contains the highest frequency terms for the given term. We add a term to the list if its frequency is greater that the list's minimum, and remove the minimum.

<u>Run Times of Trie</u>
Create trie: Θ (m) -- m = entries in text file (assuming m >>> largest input size)
Insert (addPattern): Θ (s) -- s = size of string to insert
Search (searchPattern): Θ (s) -- s = size of string to insert
predictQuery : Θ (m) -- m = entries in text file (Worst case go through every entry)

To compare and "sync" the tries from one instance of our application to another, we decided to use Bloom Filters. Each Bloom Filter is represented by an array of bits to allow for maximum compression. We hashed the trie of each instance of the application using 3 hash functions and used the resulting values to set the corresponding bits of the Bloom Filter. We then wrote the Bloom Filter to a text file and programmed the application to send the text file over the client/server sockets every time the sync button is pressed.

$$fp = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k$$

figure 2. The formula for finding the false positive rate of a bloom filter

Bloom Filters, unfortunately, have a small false positive rate. Figure 2 above shows the relationship between k (the number of hash functions), m (the size of the bloom filter), n (the number of input keys), and fp (the false positive rate) of the bloom filter. The first sample text file given in class had approximately 140,000 entries in it. Solving the equation using 140,000 for n, 3 for k, and 2240000 (16 * 140,000) as m, we get only 0.005 for fp, the false positive rate. We deemed this acceptable for the purposes of our project.

<u>Run Times of Bloom Filter</u>
hashTrie: Θ (m) -- m = 3*entries in text file
Compress to Text File : Θ (m) -- m = entries in text file (16*m bits)
findMissing: Θ (m)

Responsibilities:

Austin - Gui, file handling, and combining all elements
Everett - Trie creation and auto completion algorithm
Larry - Trie creation, Bloom filter, multithreading

Link to Video:

https://www.youtube.com/watch?v=1OLreM9S7fs&feature=youtube_gdata

References:

Hash functions:
http://stackoverflow.com/questions/8567238/hash-function-in-c-for-string-to-int

http://stackoverflow.com/questions/25280106/hash-algorithm-for-string-of-characters-using-xor-and-bit-shift

Bloom Filter Equation:

http://www.maxburstein.com/blog/creating-a-simple-bloom-filter/