COMP3506 Report – Angus Trusler 43614114

Introduction

The purpose of this document is to provide justification for the algorithms and data structures used in implementing the solutions to the problem specification.

Problem Background and Thought Process

The goal of this assignment is to efficiently search a large textual document in different ways. The textual document (referred from now as the "text") needs to be accessible in a way that enables efficient searching. As a .txt file on disk, this isn't possible in its current form, so pre-processing is required. Therefore, consideration needs to be given to the nature of the text itself and what appropriate data structure is most appropriate. For instance, is the text composed of words with little variation and equal frequency, like DNA genomes; or is the text a standard use of the English language, with a high variance of words, each with unpredictable frequency? Analysis of the text itself is vital to appropriate data structure choice.

The specification identifies "general textual searches" as the type of text to be searched and provides *The Complete Works of William Shakespeare* as an example, so passages of English text were assumed to be the text type for this application.

An important characteristic of language texts that was considered when developing the submitted solution to this assignment, is the observed frequency and variance of words within a given text. English text follows two relevant laws related to word frequency and variance: Zipf's Law, and Heap's Law.

Heap's Law states that the degree of variance of words within a given text is proportional to the text's length. In other words, as a given text gets longer, the less likely a new word is to be encountered. For the purposes of this assignment, the variety of words in a text is O(log n) where n is the text's length.

Zipf's Law states that the frequency of any given word with a text is inversely proportional to the word's rank on the text's word-frequency table. In other words, the second most common word will appear in the text approximately half as often as the most frequent; the third most common words, one thirds as often, and so on. As this describes a power law, for the purposes of this assignment, the frequency of a word is O(log (n*l)) where n is the variety of words within a given text, and l is the length of the text.

These two laws play important roles in determining the runtime complexity of all methods in any solution to this assignment and will be referred to later in this document.

Chosen Data Structure Justification

The primary data structure chosen for this assignment takes the form of a Compressed Retrieval Tree (referred from now as “Compressed Trie” or “the trie”). The trie is made of a hierarchy of linked nodes, each representing an ordered series of characters. Typically, a word from the text would be identified within the hierarchy with a special child node of a character. Textual words are retrieved from the trie by beginning at the empty root node, and ‘spelling’ the desired word by following subsequent child nodes.

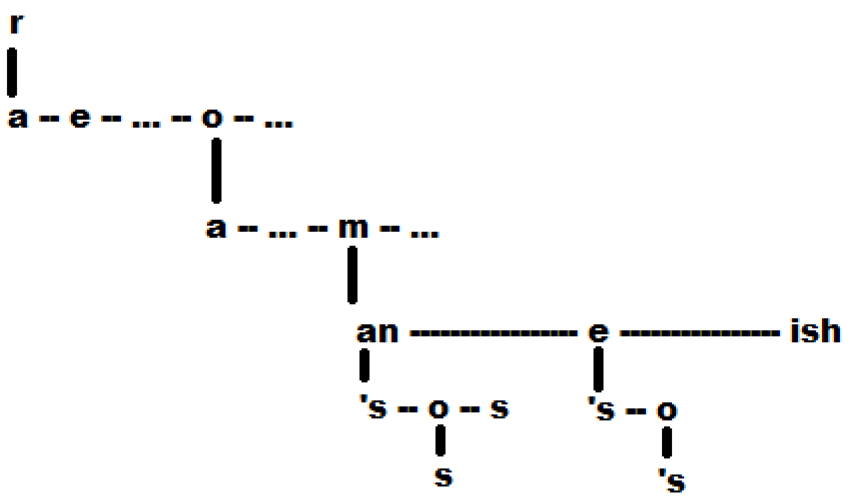
In the submitted implementation, the CompressedTrie class follows this outline as closely as possible. There are two subclasses:

- trieNode – a class representing the nodes that make up the linked hierarchy.
- textCoordinates. – a class representing a location of a textual word within the textual document.

A standard compressed trie has one parent node with many children nodes. As the number of children nodes can vary according to alphabet size and to what extent the trie is compressed, the parent/child relationship had to be dynamic. To this end, instead of having a parent node point to an unknown number of children, the parent points to one child: the lexicographically first child.

To maintain the appropriate relationship between parent and children, accessing the children involves iterating over all children until the appropriate node is identified. This is achieved by ordering the children in a linked fashion via ‘siblings’. Each sibling refers to it’s ordered neighbour, and all refer to the same parent.

Fig 1: a layout of child / sibling relationship amongst trieNodes


r
a -- e -- ... -- o -- ...
a -- ... -- m -- ...
an ---------------- e ---------------- ish
's -- o -- s
s
's -- o
's

Each node has a reference to the last found position of the word it represents (a textCoordinates object). To track the location of *every* position for a given word, the textCoordinates objects link to each other, forming a linked list. To find a specific occurrence of a given word, one must find the word in the trie, then navigate the linked list to the specific position.

Acccess Time for a Word

Access time to any given node, like a standard compressed trie remains O(a * l) where a is the alphabet, and l is the length of the word.

To get the access time of *all* occurrences of a given word from the perspective of the whole document though, I refer to Zipf's Law and Heap's Law. Knowing that the most frequently occurring word in any given text occurs logarithmically proportional to the entire text's length, the amortised runtime complexity of scenarios of word access becomes O(log n) where n is the original text's length. For example, to return all locations of "the" within a text can conceivably run in O(log n) time where n is the text document's length.

Access time for a *specific* occurrence of a word is linear (O(n) where n is the word's frequency), as to find the specific textCoordinates object, one must navigate the ordered linked list of all word occurrences. It is worth noting that the length of this list is O(log n) where n is document size, per Zipf's Law.

Adding words into the trie requires navigating to the closest appropriate node, so adding a word is the same runtime complexity as accessing *all* locations of a given word (O(log n) where n is document size).

Memory usage for Document

To store a contiguous set of strings in a file, compressed text (like in Huffman Encoding) is very memory efficient. However, access to specific points within a text relies on pattern matching algorithms, which are never better than O(n) where n is the document size.

Because this assignment is focused on accessing specific points in the document, rather than storing the document efficiently, the choice was made to prioritise access time at the expense of memory usage.

However, while more memory is used than the simple text file, memory usage is still proportional to the total number of words in the document (O(n) where n is the text's length.). The difference is in the space requirement for a given word. In text file, it's a simple string (or less). In my CDT, the memory usage of a word is: TC + TN/WF where:

TC is the memory of a textCoordinates object, TN is the memory of a trieNode object, and WF is the word's frequency.

Justification for not separately tracking stop words

In phraseOccurence, pattern matching algorithms like brute force or Boyer-Moore may see additional benefit from only considering less frequently occurring words (as the text size lowers considerably, and word variety increases). However, to utilise phraseOccurence, identifying where stop words exist is necessary to correctly locate most phrases (as stop words are invariably the most common words, phrases are likely to contain them. Therefore: not tracking stop words is only beneficial when attempting to locate phrases that don't contain them.

So to determine if the submitted data structure is at least as good as pattern matching, the worst case scenario will be considered: A phrase entirely consisting of stop words.

Because any word (including stop words) is accessible in O(log n) time, all locations of the phrase's first word are identified in O(log n) time. Additionally, knowing that word frequency is O(log n), the number of candidate lines are O(log n).Where orthodox pattern matching must consider every line O(n), my algorithm checks in (non-pathological use) worst-case scenario a log-n number of lines (where n is text document length). When one considers that the words following the first word still follow a Zipfian distribution, the number of passing candidate lines remains a log-n number (where n is now the number of candidate lines). Therefore, to locate a phrase (even if only stop words), the runtime complexity of the algorithm is proportional to the O(log n) where n is the document's length.