CS-1501 Algorithm Implementations – Writing Section

Summer 2013

Project 1 – Writeup

**The de la Briandais Tree: A Runtime Analysis & Comparison to Sorted Arrays via Implementation in a String Search Algorithm**

**By Tyler Raborn**

The need for efficient searching of a list of ASCII strings transcends the many subdivisions and genres of modern-day programs. From HTML internet browsers to text editors, from spreadsheets to binary files, search algorithms need to check sets of strings against pre-determined “master” sets, with the most obvious (and that being used in this particular experiment) example implementation being that used with the words in a dictionary.

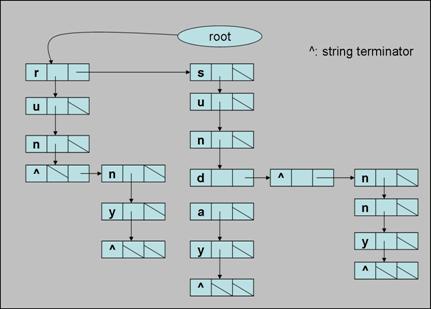
Among the most performance-critical aspects of this implementation is the manner in which the dictionary’s words are both stored and iterated over. Despite the considerable size of an English dictionary in terms of entries, systems have evolved to possess memory pools of such capacity to make this once imposing size a trivial detail for most modern software implementations. As such, any dictionary search algorithm simply needs to utilize a cost-effective and relatively efficient manner of iterating through a memory-based data structure in search of the target string.

In this particular implementation, the algorithm’s primary goal is to analyze a set of given input strings, and to calculate every possible anagram using the letters in those strings such that the anagram contains the exact number of letters in the original string. For clarification, an anagram can be defined programmatically as follows: some string is an anagram of some other string if, and only if, the set of characters contained within string isidenticalto the set of characters contained within, the number of occurrences of each character in each string are identical, and ifis a valid word in the dictionary**.** The algorithm tests for these stipulations in a classic brute force manner: by comparing the various permutations of each letter in each word listed via the input file to the pre-established valid words in the *dictionary* text file. The working hypothesis is that the data structure used to store the words from the dictionary file is an asymptotically significant factor in terms of the algorithm’s worst-case runtime.

The algorithm uses a variant of the brute-force approach of checking *every possible* permutation of characters in the string. This yields a runtime of,where is equal to, or the length of the permuted string. The method of *pruning* is utilized in this variant of the algorithm, which helps to detract from the base version’s imposing factorial runtime. Pruning refers to the process of checking to see if a particular character permutation, or element, has already been processed, and if it has, halting any and all operations, recursive calls, and calculations on that element by removing it’s visibility to the algorithm, thus ensuring that subsequent recursive iterations stemming from this element are not performed. Pruning as implemented in the anagram program is as follows: as the outer for-loop iterates over the length of the input string, a separate parameterized array of each character in the input string holds the remaining valid letters. As each element of this char array is tested, it is replaced by an ampersand character which ensures that any future iterations of the algorithm do not modify that index. This is achieved via a simple if-statement at the beginning of each for-loop iteration that checks for the presence of the ampersand. Should this character be found, the algorithm skips any further calculations or recursion involving that particular index, and continues on to the next index. This code serves to ensure the elimination of redundant answers while improving the algorithm’s overall runtime, although little to no effect is observed on the aggregate asymptotic measurement.

Provided that the string calculated is not a redundant element, the algorithm proceeds to perform a check on the validity of the string as a word in the dictionary. Specifically, it checks to see if the string satisfies any of the following states: that the string is a prefix to some valid word, that the string is a valid word but not a prefix, and that the string is a prefix to some valid entry in the dictionary but not actually a word. Should the string be a valid dictionary entry but not a prefix, pruning is enacted in that the algorithm halts any further recursion on the word. This is done via an if-statement that compares the string to the contents of an array allocated for storing already-calculated anagrams. The other two states both result in further recursive calls being enabled on their respective character permutations; the only difference in how the algorithm handles these is that should the string be a word in addition to being a prefix, that string is added to the data structure responsible for holding all calculated valid anagrams, which, in turn, is checked for the presence of the current target permutation.

While pruning certainly helps to reduce redundancy and the resulting repetition in finding the anagrams *in* a dictionary, an even greater performance benefit should come in the form of *how* the dictionary is stored. The choice of data structure will noticeably impact the runtime virtually every single time the algorithm compares a permutation of characters to entries in the dictionary, and as such, this choice is absolutely critical in terms of reducing the program’s runtime.

To analyze and illustrate the aforementioned criticality in the choice of data structures for use by the anagram algorithm, two radically differing constructs are compared. One represents the *naïve* approach of a sorted array, while the other demonstrates the inherent usefulness for handling strings in the family of specialized trees called *tries*, which are described in detail later on.

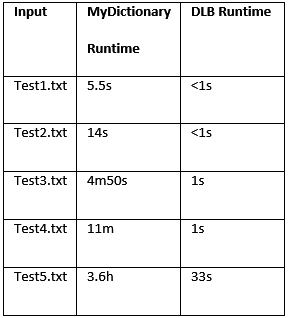
**Figure 1.1** – a de la Briandais tree (<http://people.cs.pitt.edu/~kirk/cs1501/quiz1sol_files/image004.jpg> )

The first data structure takes the form of a sorted Java ***ArrayList*** located in a wrapper class called ***MyDictionary.*** While both simple to visualize and to implement, the runtime for searching a sorted array is; recall that in a sorted array of elements, the worst-case scenario for a linear search is that which requires iteration over all indices to find the last element. This runtime is a bottleneck in the performance of the Anagram-finding algorithm, as a faster data structure for string search speeds will directly impact the program’s performance.

MyDictionary contains the necessary methods and fields to allow for prefix-searching; that is, method ***searchPrefix()*** accepts a string and returns one of three different integer values based on whether that string is a prefix, a word, both or neither in the underlying array. ***Add()*** simply appends the parameterized String object to the end of the array and subsequently performs any necessary shifting and sorting to ensure that alphabetical sorting is maintained. As these methods all depend on searching the sorted array, the overall performance of the program is bottlenecked by this naive data structure’s linear runtime.

A far greater structural alternative in terms of string object access speeds is the de la Briandais tree (see **fig. 1.1**), a type of trie data structure that is nearly perfectly optimized and designed for holding large sets of words in memory. As with any tree data structure, the DLB is a series of interconnected *nodes*, and any performance gains come at the price of high memory usage*.* However, much of the similarity ends there, as each node is not just a container for a single value or datum, but rather an entire Linked List data structure, whose individual nodes each contain a single character in a word in the dictionary. The maximum, or *worst-case*, size of one of these Linked Lists is limited to just twenty-six, the length of the English alphabet. As with all tree data structures, the DLB features a *root* node, from which all searches originate. In this case, the root node is a Linked List whose nodes contain the first letter of each word stored in the tree, with no repetition. This lack of doubles holds true for every Linked List in a DLB; it eliminates redundancy by sharing all *initially common* characters should they be shared by multiple strings within the DLB, with each list’s size never exceeding the length of the alphabet. To clarify this concept: assuming two strings andstored in an otherwise empty de la Briandais Tree, a single common prefix of length is shared between the two if, and only if, the first characters of both and are identical in size, composition, and ordering.

The DLB implementation utilized in the experiment features a wrapper **DLB** class that contains an internal, privatized **LinkedList** class and as well as a private **Node** class. The Linked List class, while not entirely necessary from a technical standpoint, helped with the visualization of the DLB’s structure during the coding process. It features a ***Head*** field, for keeping track of the first element of the list, as well as a special constructor that attaches a pointer to the new list’s head node, which in turn points back to the list that it is contained within. This feature was very useful during iteration, as it allowed the programmer to derive a node’s linked list by calling that node’s ***thisList*** field, as opposed to having a constantly incrementing localized Linked List pointer.

****The ***Add()*** method of the DLB is split into two parts: if the DLB is empty upon calling the method, then ***Add()*** simply creates a chain of Linked Lists, with each list’s head node holding the ensuing character in the string. If the DLB is not empty, a for-loop iterates over the length of the word until it either finds the string in its entirety followed by a terminating sentinel character, resulting in a false return Boolean, or until the characters no longer match, in which case it appends the missing characters to the bottom of the DLB one ***LinkedList*** object at a time. It then returns true after attaching a sentinel node to the end of the string. The sentinel itself in this implementation is a Node object that contains the asterisk ASCII character. It is important to note that in every situation where ***Add()*** returns true, a Sentinel character is appended. To further demonstrate this concept, refer back to ***Figure 1.1***. Prior to the string “sunny” being inserted into the DLB, words sharing a prefix with “sunny” include both “sunday” and “sun”. In this illustration, the sentinel node is that which contains the ‘^’ character. Thus, sharing a linked list with the character ‘d’ in “sunday” is a sentinel node, which indicates that the previous three characters of “sunday” are also simultaneously representing the stored word “sun”. Continuing with this concept, inserting “sunny”, which also shares its initial three characters with “sunday”, results in the remainder of the word, that is, the string containing all characters (inclusive) after the first char mismatch between “sunday” and “sunny”, being stored in its own unique pointer chain, with the first letter of this non-prefix string sharing a linked list with both ‘d’, from “sunday” and ‘^’, the sentinel termination character for the word “sun”.

**Figure 1.2 –** Runtime Comparison of the Backtracking Anagram-Finding Algorithm when utilizing a de la Briandais Tree and a Sorted Array

Both of the ***isPrefix()*** methods operate in the same manner, with the parameter-less version featuring a for-loop that iterates over every character in S until it finds a mismatch. If ***isPrefix()*** finds a mismatch, it returns 0. If it does not encounter a character mismatch, but also does not find a sentinel node appended to the end of the string, it returns 1. If the method encounters only a sentinel node, ***isPrefix*** returns 2. And finally, if it encounters both a sentinel node *and* other characters in the final linked list, it returns 3, which indicates that S is both a word *and* a prefix. ***isPrefix(StringBuilder S, int start, int end)*** works the same way, except the for-loop only iterates between the indices provided by the additional two integer parameters instead of over the entirety of S.

The difference in runtime when utilizing each data structure is asymptotically relevant, and immediately noticeable to the user. The test machine featured Windows 8 64-bit installed on an Intel Core i5 2500k processor overclocked to 4.3GHz, with 16GB of dual-channel 1600MHz DDR3 memory and all Java & Windows OS components installed onto a Crucial M4 SATA III solid-state disk. In one particularly illustrative example, the algorithm ran for three hours and ten minutes on this test machine while utilizing the MyDictionary data structure on the provided ‘test5.txt’ input list. On the same machine, utilizing a DLB instead resulted in a runtime of only 33 seconds. A direct comparison of the input files using both data structures on the test computer is illustrated in ***Figure 1.2*.**

As previously mentioned, both the worst-case andaverage runtime for searching a sorted array is, with the additional overhead of iterating over the partial lengths of words that share a prefix with the target. The worst-case runtime for a DLB, as with a binary search tree, is completely dependent on the height of the data structure. As such, this would lead to a Big O runtime of for a string lookup featuring characters with an alphabet of letters. However, as with many worst-case runtimes, this scenario is highly unlikely, as it would require each sought-after character to be the final letter in a given series of linked lists within the DLB, with each linked list containing every character of the utilized alphabet and thus achieving the maximum size possible. As such, the average runtime for the DLB approaches**,** based primarily on the length of the target string. This yields the following comparison: the sorted array’s average search time is based on the length of the entire dictionary array in conjunction with the length of the target string, while the DLB’s average search time is based solely on the length of the word. For example, if test input file ‘test4.txt’ is used, which contains the strings “backtracking” and “donaldknuth” possessing lengths of 12 and 11 letters long, respectively, each 12 and 11-letter-long valid anagram would average with a DLB, and with a sorted array, where is the length of the sought-after string, and is the length of the sorted dictionary array. With a length of 5,455,8 words, that is, from a complexity standpoint, in the same class as five-thousand times longer than the 12 characters of “donaldknuth”, the limiting factor for the DLB’s runtime. As such, the DLB provides a very noticeable and helpful boost to the performance of the search algorithm.