Lab Report for Project 2

Dominic DiPofi

11-6-21

Introduction:

In this project, different .txt files are read in a program, word by word. Each word is added to a node that stores a String containing the word as well as an int count of how many times the word has appeared. There are four different add methods for each of four different lists: unsorted, sorted (alphabetically), self-adjusting (to the front), and self-adjusting (up by one). The time it takes to read and add each word in a .txt file, the number of nodes/distinct vocabulary words, the total number of words/sum of the int count, number of key comparisons (each time the current word being read in the file is compared to a word in one of the nodes in the list), and the number of reference changes (either when a node’s link or the list pointer is changed) are all recorded for each of the four lists, and their outputs were put into an excel sheet with various graphs used to analyze the data.

Data:

The data was recorded in a .xlsx file, but it is also shown below to more easily describe in the analysis section.

Figure 1 (Data for Hamlet, Bleak House, and War and Peace .txt files)



Figure 2 (Run time for the four lists for Hamlet, Bleak House, and War and Peace .txt files)

Figure 3 (Run time vs vocabulary for List 1 for the aforementioned .txt files)

Figure 4 (Run time vs key comparisons for the Hamlet .txt file)

Analysis:

Figure 1 shows all the data collected for three .txt files (Hamlet, Bleak House, and War and Peace). The letter after the number (H, B, or W) represents Hamlet, Bleak House, and War and Peace, respectively. As seen in this figure, the reference changes do not seem to have much of an impact on run time, as Lists 3 and 4 both have similarly high reference change counts, but list 3 is the fastest list for all the .txt files while list 4 is the slowest. This seems to highlight that moving nodes takes time, but so does searching through each node, so putting the most recent word’s node at the front, like in list 3, makes search times quicker which makes up for the runtime costs due to switching nodes and links. This is not the case so much for list 4, as the node is only being moved one space at a time which leads to an increased number of key comparisons as compared to list 3, which is shown in more detail in Figure 4.

Figure 2 shows the run time for the four lists for Hamlet, Bleak House, and War and Peace. The run times are much shorter for Hamlet, as it has many less words than either Bleak House or War and Peace and is to be expected. Bleak House has less words than War and Peace, and it’s run times are shorter as well.

Figure 3 shows the run time versus the vocabulary of the .txt file for List 1, but a similar-trending graph could be drawn from any of the other three lists. As the number of distinct words making up the vocabulary increases, the run time increases. This makes sense because more words means that more comparisons need to be made, which increases the runtime.

Figure 4 shows the run time versus the number of key comparisons for the Hamlet.txt file. It can be logically concluded that as key comparisons increase, run time increases. This is why list 3 has such a low run time compared to the other lists; it has a substantially lower number of key comparisons compared to the other lists.

Conclusion:

As seen in the data and analysis sections, lower runtimes are a result of lower word count, less reference changes, and fewer key comparisons. A high reference count can be rectified if it results in a substantial drop in key comparisons, as evidenced by list 3.

Overall, this project helped me to construct and use lists in a meaningful way, as well as create various methods to utilize these lists. The lab report was useful to help me draw conclusions from the program data that I may not have otherwise seen. I look forward to continuing to add to this project with more lists in the future.