CS 1632 - DELIVERABLE 4: Performance Testing

https://github.com/shixiongjing/D4

Shixiong Jing(shixiongjing)

Xiaomeng Niu (XiaomenggNiu)

**Summary**

The most challenging part of this deliverable is coming up with an optimal way to traverse the text file and store the longest word. Starting from every node and using permutation to find out all possible words are not an optimal choice. We designed our algorithm to speed up from 2 parts.

Part 1: By observing the identity of acyclic graphs and noticing the requirement that all words must only stop only at the end node, we decided to build the tree from the end nodes (nodes that does not have edges to point to other nodes) and build the DLB trie bottom up. This can save time by avoiding repeated routes. For example, if using the naïve approach, after visiting node A and go from A->B->K->X->N…->W; but when we visit node B, we have to go through B->K->X->N… ->W all the way again. Instead of this naïve approach, we used the end nodes as roots and build a tree structure bottom up to save some repeated routes.

Part 2: The easiest way to store and search from dictionaries is DLB trie. In this project, we used an external library called Trie, which basically support a built trie data structure (For more information, please go to <https://github.com/tyler/trie>). Since the order of the letters does not matter, in order to reduce the size of the trie and make the DLB more efficient, we decided to store words in alphabetically sorted fashion with extended ‘.’s for different words with same sorted result (edge cases for words like ‘tab’ and ‘bat’ that both end with same sorted form, in this case ‘abt’). When searching for words, we will also sort the ‘proposed words’ before handing it into the DLB trie. We will also check if there are other variable forms stored with extended ‘.’s each time a temporary longest word is found.

Part 3: We used new\_list.rb to pre-build the DLB trie ahead of time and store it in newlist.ba and newlist.tail, which can be read by word\_finder.rb to create the trie object.

Part 4: To make the sorting of words more efficient, we built a special version of insertion sort. Insertion sort is one of the best sorting algorithms for small arrays (many quick sort implementations use insertion sort for small arrays). It was further optimized for our use case because the sorted words were grown from roots, thus each time it would only insert one letter into an already sorted string. So, we only need to do one round of ‘insertion’ of the insertion sort.

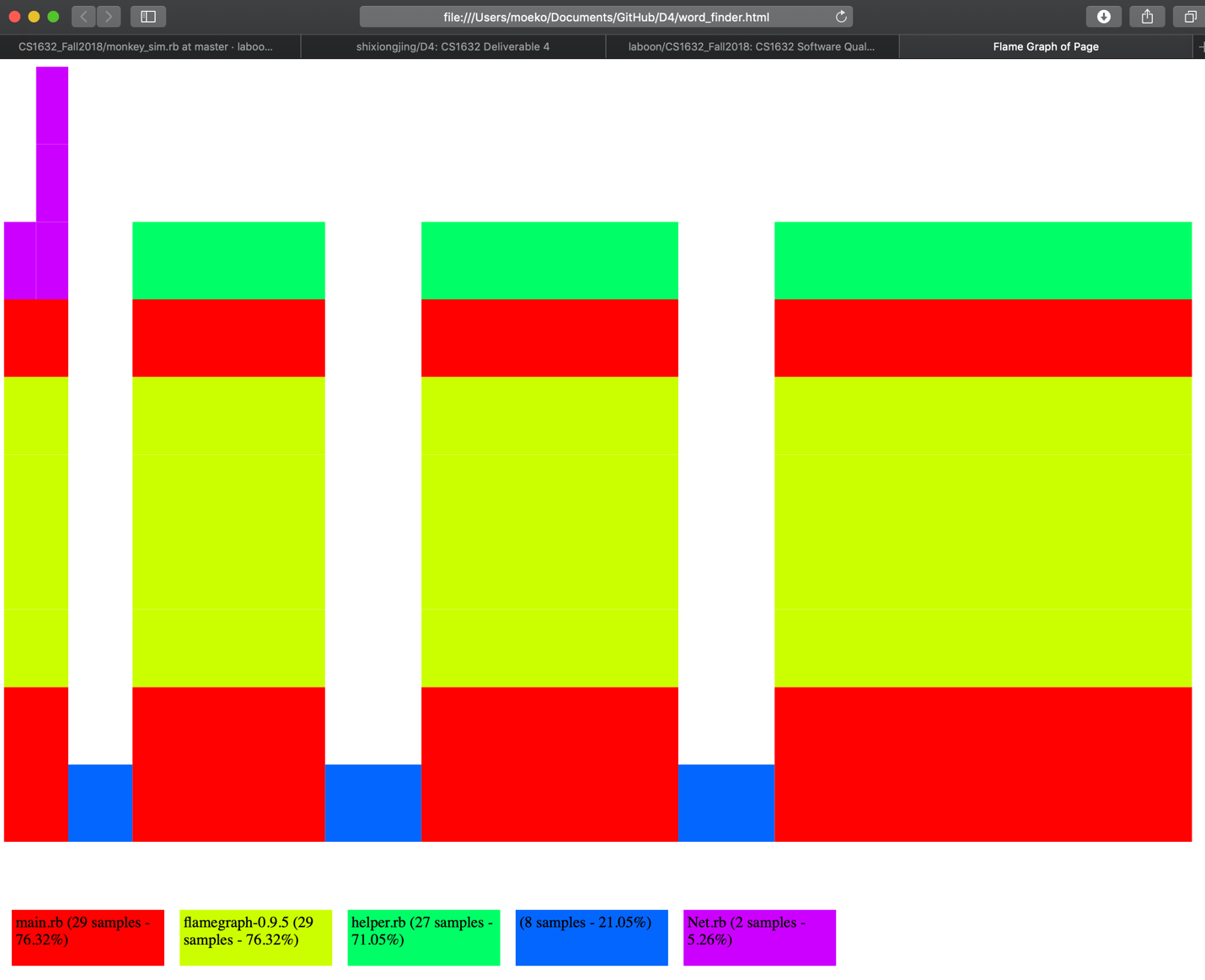
For testing, we also tested edge cases on different inputs of the program (no input argument, one invalid argument, more than one input arguments).

The result of our flame graph showed that the methods which took up the most CPU time were used to generate the flame graph itself. Therefore, the first improvement we made is simply remove the flamegraph code. Second largest portion of the CPU time was spent on helper methods, which is likely to be the insert\_sort or the read\_dictionary method. However, there is not much to improve on the sort method because even if we know where to insert the new letter ahead of time, making space for the letter in the array will have the same runtime as one round of ‘insertion’ in insertion sort. Read\_dictionary also does not have much to improve because it simply calls the File.read() function from the standard library.

The initial real runtime for initial commit is 0.142 seconds on average for ultra large graph, and 0.140 seconds on average for revised commit. Improvement of 0.02 seconds is significant recalling that even the runtime for small graphs are above 0.130 seconds in both version.

**Screenshots**

Flame Graph

****