EECS 2500- Linear Data Structures: Lab 3

Balihaar Gill

Date: 11-27-2018

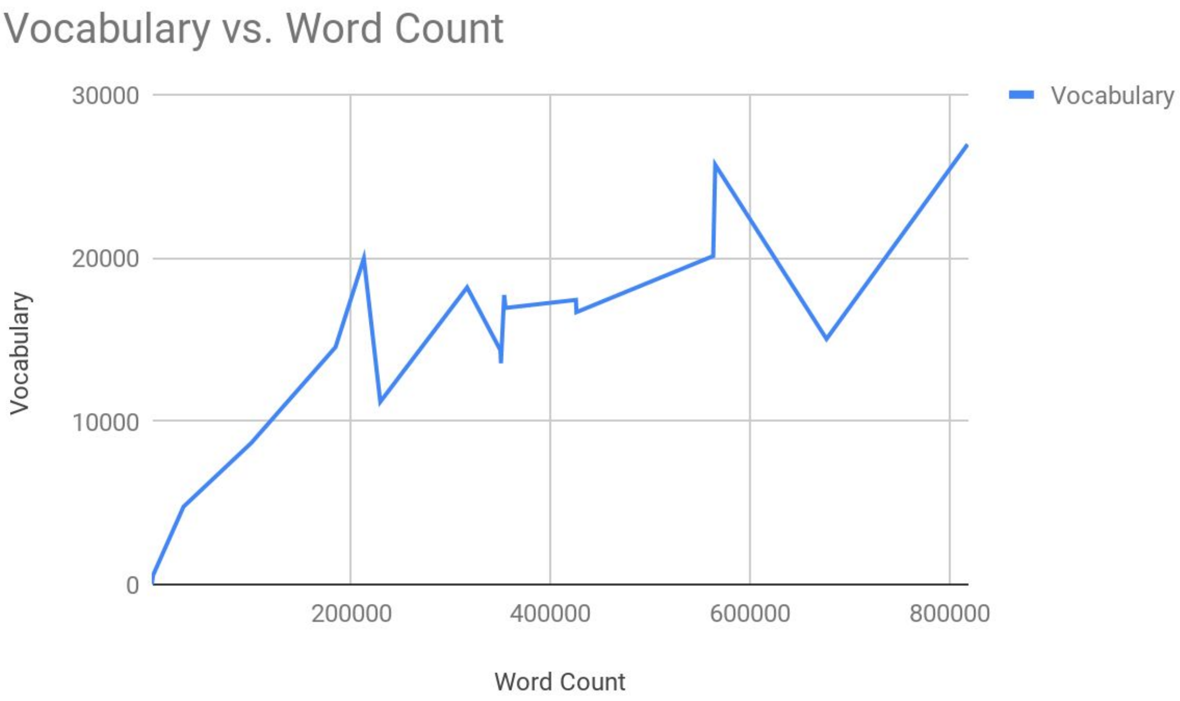
Lab Objective:
To code multiple lists that parse large text files and gather multiple data metrics from them.

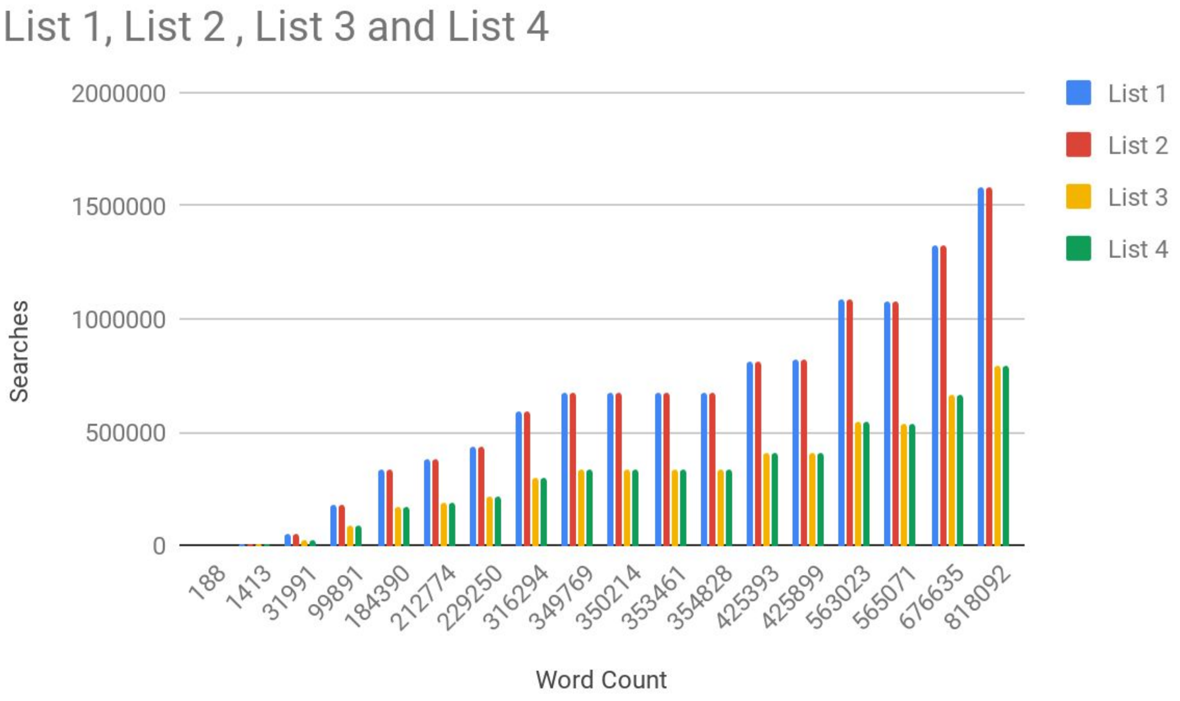
Introduction:
After learning about list implementations in lecture, this lab is to provide evidence of the efficiency of different types of lists and their organization methods based on metrics such as word occurrences, original words, reference changes, and comparisons.

Procedure:
This lab utilized:
Object-Oriented Data Structures Using Java: 3rd Edition
Eclipse IDE 2018‑09
JDK 11

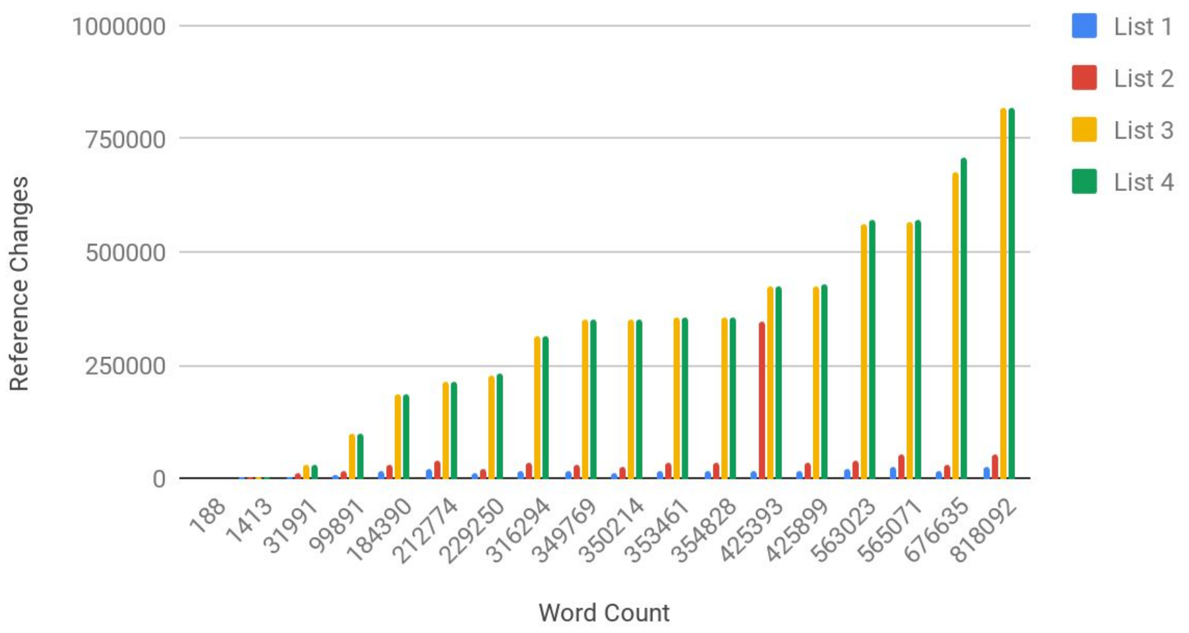
- Parse through a selected text file and convert every token to lowercase and remove all punctuation. Parsing is done by delimiting spaces.
- Add the words to multiple singly-linked lists in the following order:
    - An unsorted linked list, in which additions always occur at the beginning
    - A alphabetically sorted linked list.
    - A self-adjusting list in which, when a word is found to be already in the list, the node containing that word is moved to become the first node of the list. For words NOT already in the list, they become the list's first word.
    - A self-adjusting list in which, when a word is found to be already in the list, the word's position moves back up towards the start of the list by one node. Words NOT already in the list become the list's first word.
    - Adjust the sorted list to check the previous word before sorting to find a better starting pointer.
    - A skip list!
- Provide the following performance metrics:
    - 1) The total number of words in the list (sum counts in each node).
    - 2) The total number of distinct words in the list – the file's vocabulary – also to be computed at the end, rather than on-the-fly.
    - 3) The total number of times a comparison was made between the word just read and a node in the list.
    - 4) The total number of reference changes made.
    - 5) Elapsed time (in decimal seconds with three places - measure time in milliseconds, and divide by 1000, so your output will look like "12.345 seconds")

Results:

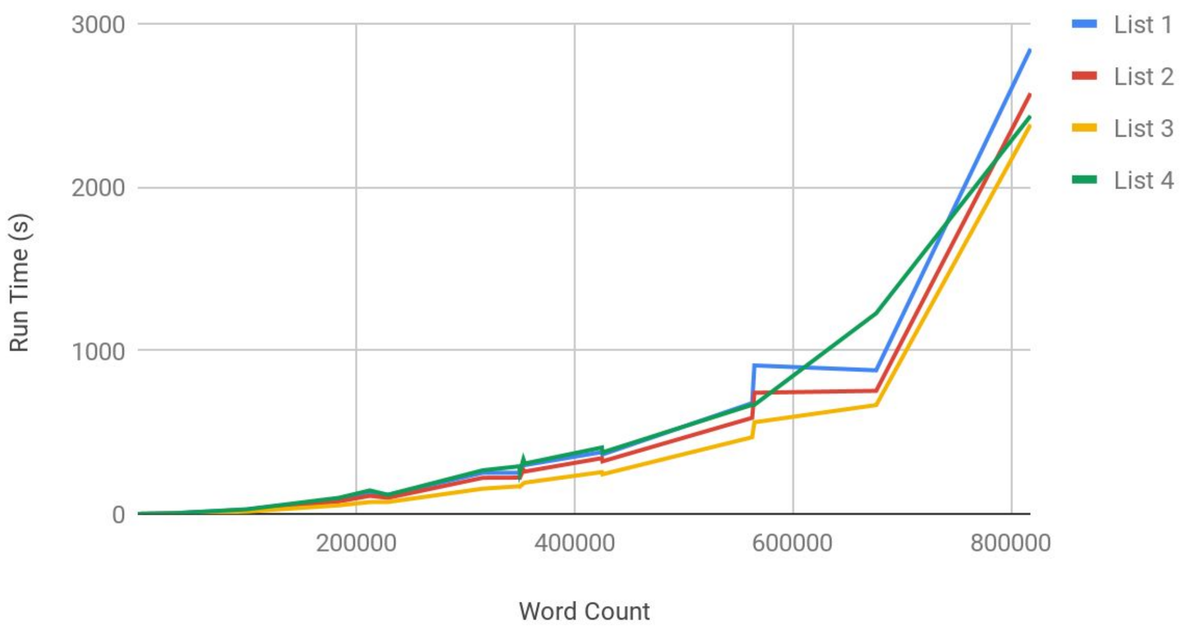

Vocabulary vs. Word Count
Vocabulary
30000
20000
10000
0
200000
400000
600000
800000
Word Count

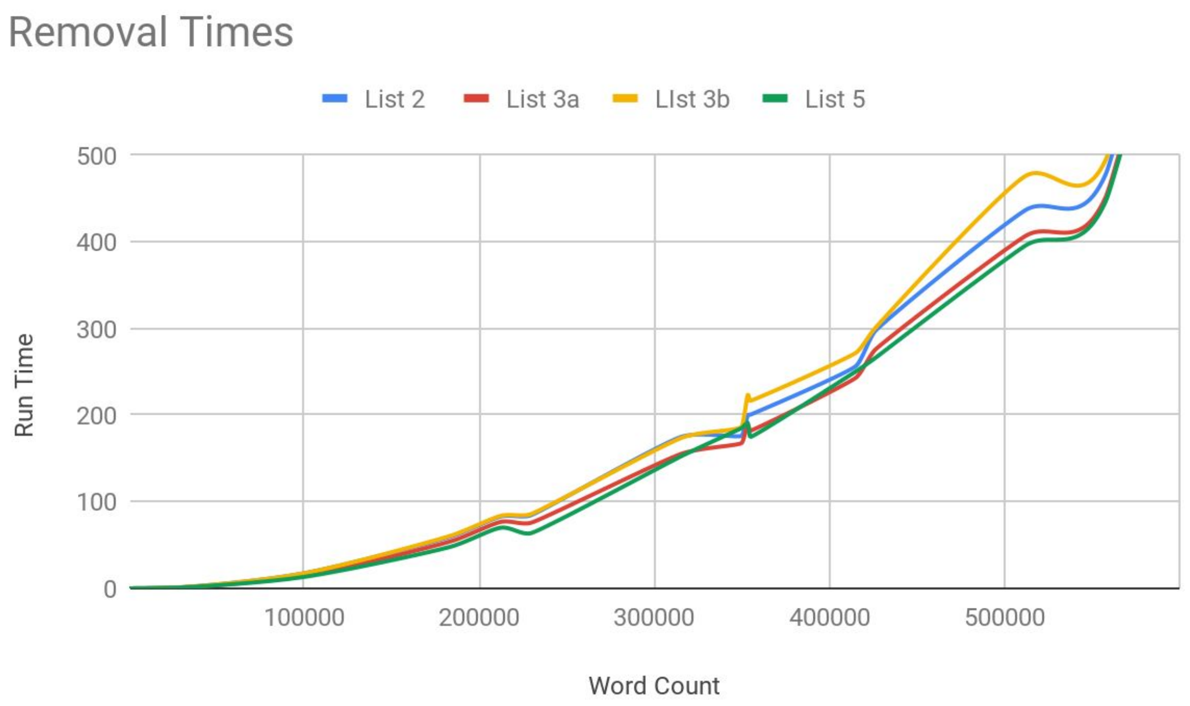

List 1, List 2 , List 3 and List 4
List 1
List 2
List 3
List 4
Searches
2000000
1500000
1000000
500000
0
188
1413
31991
99891
184390
212774
229250
316294
349769
350214
353461
354828
425393
425899
563023
565071
676635
818092
Word Count

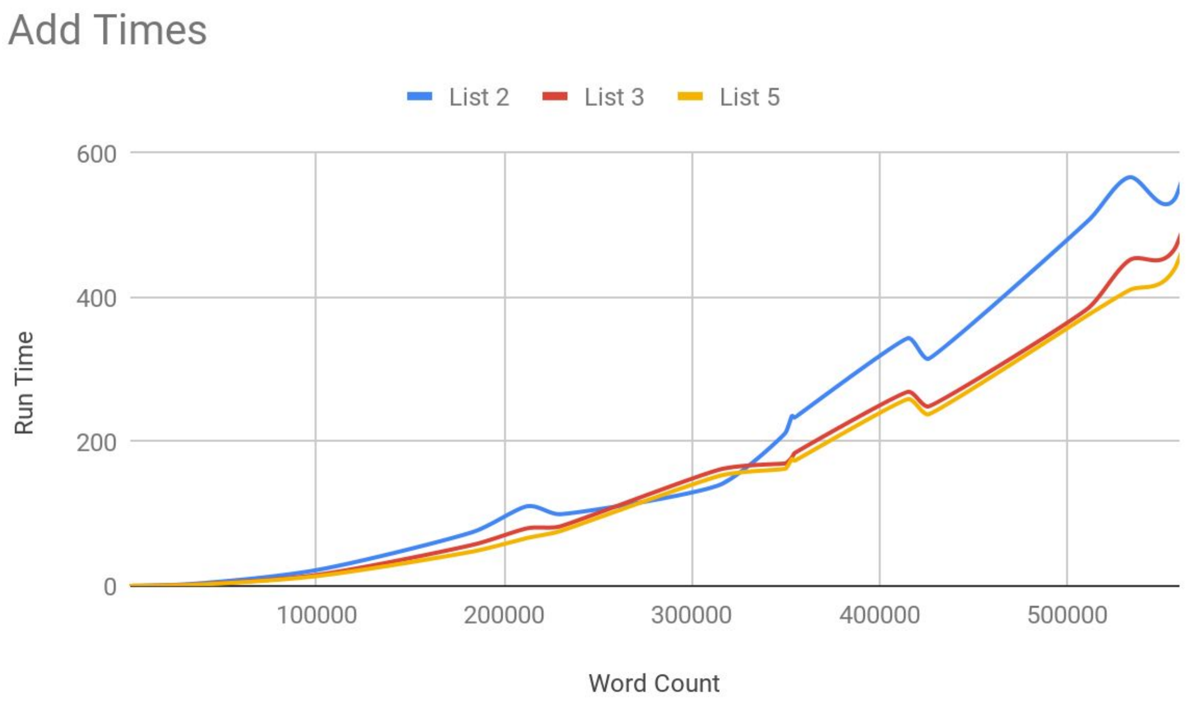
List 1, List 2, List 3 and List 4



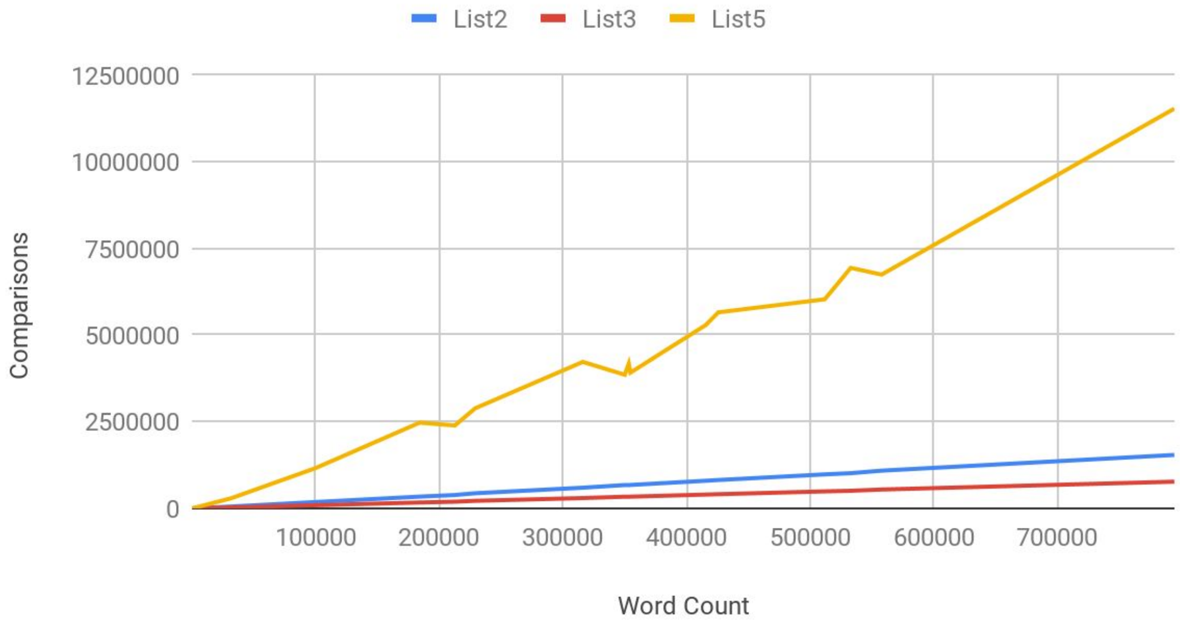
List 1, List 2, List 3 and LIst 4



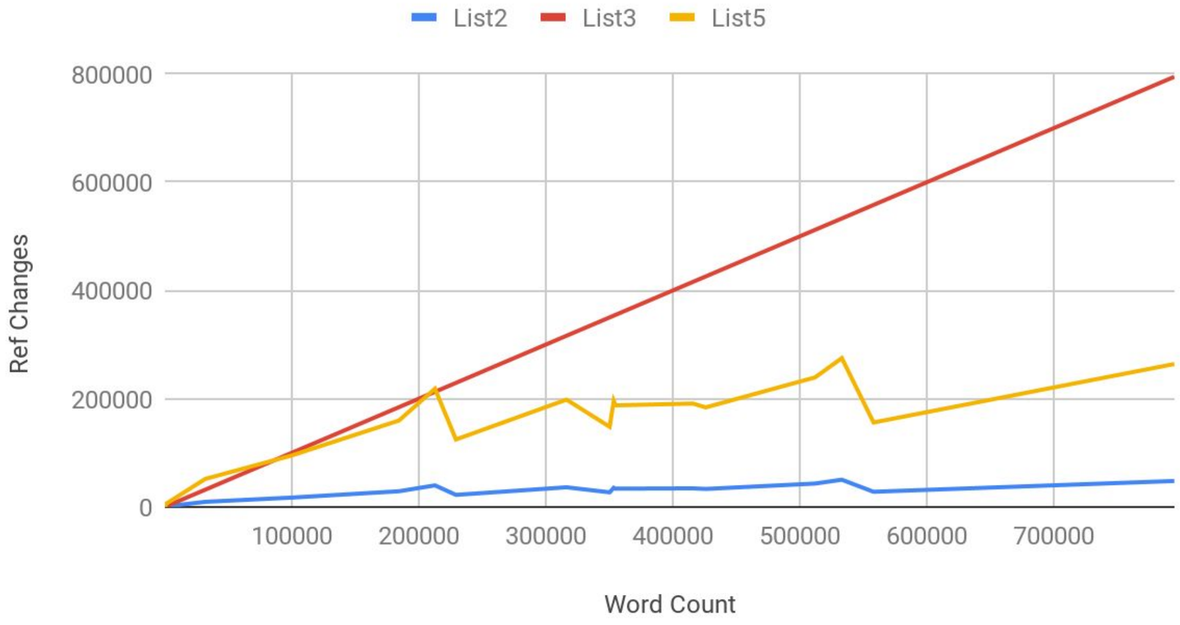

Removal Times


Add Times

Comparisons



Reference Changes



DIscussion of Results: Vocabulary and word occurrences stayed consistent with each list which ensured that each list was working correctly.

Runtime, which would generally be considered the factor of efficiency was best with the self-adjusting to front list (list 3). Followed very closely by node swapping list (list 4). The sorted list (list 2) worked surprisingly efficiently which could be because of search times getting better with each loop. The unsorted list (list 1) ran the slowest most likely because there was no form or organization to it. Runtimes may have varied the most due to different environments in which they ran, while different programs are running and other factors.

Searches were greater in lists 1 and 2 than in 3 and 4. This is likely due to 3 and 4 accounting for occurrences and moving frequent words closer in the list which reduce both search comparisons and search times.

Reference changes were greater in 3 and 4 than in 1 and 2 due to this. With more moving of nodes to account for occurrence frequency, reference changes would be at an increase. There is one odd case for list 2 which might just happen to be an outlier or a very oddly structured text.

With the newly implemented lists and changes, the skip list proved to be the fastest at inserting words. However, what is surprising is how neck-and-neck it is with the new list 3 that compares the inserted word with the previously inserted word as a starting pointer. At higher word counts, the skip list proved easily to be the winner. The sorted list easily showed to be the least efficient of the three.

The removal times of the new lists seemed about the same until about 200,000 words, when list 3b times were the longest, followed by the sorted list, 3a, and the quickest, as expected: skip lists.

Comparisons were unnaturally larger in the skip lists than in either list 2 or 3 which seemed to be an oddity. Whether this is natural or just a poor implementation of a comparison checker is unknown.

Reference changes were linearly increasing with list 3, probably due to the large amount of location pointer changes due to started halfway or at the beginning. Skip lists had the second most reference changes, likely due to more node to handle. List 2 had the least.

Conclusion: A list that organizes based on frequency of word occurrences will most likely be the best implementation with respect to efficiency. A sorted list is surprisingly faster than an unsorted list due to searches being made easier with every loop. Lots of factors can play into run time and a home computer with multiple processes running might not be the best option for experimentation. Run times increase significantly when done from a hard drive rather than a flash drive. Skip lists are easily the most efficient and useful of the lists created. Small changes in other lists such as looking at the previously inserted word also drastically improve run times. The changes with list 3’s second removal did not look to have a large impact relative to the other removal methods.