Introduction

Sparse suffix trees are a variation of the suffix tree data structure that are much faster to construct and traverse due to the decreased number of nodes in the tree. Based on the 2017 paper by Gawrychowski and Kociumaka, my sparse suffix tree algorithm employs optimal construction methods while maintaining reasonable cognitive breakdowns for readability.

Gawrychowski and Kociumaka's algorithm takes advantage of Karp-Rabin fingerprints as a centerpiece to speedier suffix tree construction. Breaking the text into "supercharacters" (fixed-size blocks of the text), the algorithm saves the fingerprint of each supercharacter which makes comparisons faster than brute-force checking individual characters. In doing so, the tree need not maintain lexicographical order during construction time, which saves on sorting costs. Once the general structure of the (unsorted) trie is formed, the algorithm switches the fingerprints back to the original text and sorts each level of the trie lexicographically to form the sparse suffix tree.

File Breakdown

main.cpp

This file contains general utility functions that manage the suffix tree object construction. The user must specify a b value, which will determine the length of each supercharacter (ceiling function of [n/b], with n = the number of characters in the text). The user must also specify a c value, which is a certainty variable. The chosen c value will guarantee that the suffix tree algorithm produces the correct result with probability $1 - n^{-c}$.

phis.h

The algorithm heavily relies on Karp-Rabin fingerprints to construct the initial trie that will be used to build the final sparse suffix tree. These fingerprints are intended to be easier to process, sort, and compare than substrings. However, these fingerprints must be calculated in an efficient way. My phis class stores the fingerprints of each suffix in the text, which allows for much quicker computation of the fingerprints of substrings later on. Fingerprints can be retrieved by a simple getter, called get_fingerprint.

node.h

The trie will be constructed using nodes. Each node object has a print (the fingerprint associated with that node in the trie), children (a map of fingerprints that lead to respective descendant nodes), a location (where the associated substring is located in the text to be used for LCE references), and a length (how long the associated substring of this node is). The variables within each node can be accessed or edited through various getters and setters. They are not static

because the lengths, fingerprints, and descendants of each node may change as the trie is being constructed.

trie.h

The trie class uses the node objects from node.h and the fingerprints from the phis.h class. The most complex portion of the algorithm exists within the trie class. Initializing the trie begins by establishing a root node and parsing through the text starting at index 0. Depending on the division of rounds as specified by the user-selected b value, the construction will insert supercharacters into the trie via the insert() function.

There are two main cases that may occur when a node is inserted into the trie:

1. There are no existing children that have a shared supercharacter prefix as the inserted suffix. In this case, a new node is simply created for the current suffix and added into the trie as a child of the root node.
2. There is at least one matching supercharacter prefix with an already existing node. In this case we follow this node down the trie. We must check if the label we matched with this node represents the entire node (there are some cases where a label only represents the prefix of a node, so the entire node may not match with our own suffix)
    a. If the label did not represent the entire node, we must use an LCE calculation to determine exactly how much text represented from this node matches our own suffix we wish to insert. Once that is determined, the matching text is separated from the mismatched text such that two nodes are formed. The text that matched takes the place of the original node, and the mismatched text becomes a new node that is a child of the matched text node. Then, the mismatched text on our own suffix becomes another child of the matched text node. Any children the old node had become children of the mismatched text node; now they are grandchildren to the matched text node. (This is by far the most complicated part of the algorithm)
    b. If the label actually represents the whole node, we can then repeat the process of checking the children of this node for more matches (start again from #1 with the child node).

This entire process requires frequent calculation of fingerprints in order to find text matches, and also creates numerous nodes depending on how many matches occur in the text.

Snode.h

The snode object is a node specially designed for the sparse suffix tree construction. The difference between the snode and node objects are that the children of the nodes are stored in unordered maps, while the children of the snodes are stored in ordered maps. In fact, this distinction between sorted and unsorted children is the only distinction between the compacted tries from trie.h and the actual sparse suffix tree.

SuffixTree.h

The final piece of the construction is the sparse suffix tree itself. The constructor of the sparse suffix tree class calls the constructor of the trie class, then bases its own construction off of the structure of the trie. The sparse suffix tree essentially rebuilds itself, mimicking the branches and node placements of the trie with the distinction that now the children are all sorted rather than unsorted. The divisions between nodes remain identical. Thus, the SuffixTree class is relatively simple to understand, since all of the difficult work already occurred in the trie class.

Modifications

The most major modification I made to the original theoretical algorithm was the choice to rebuild the tree as a suffix tree rather than simply editing the existing trie to turn into a suffix tree. The reason for this is based on the fact that I used an unordered map to store the child nodes of the trie due to the readability and efficiency of using hashing to retrieve information rather than a binary search. However, once it came time to actually sort the child nodes, I rebuilt the tree using ordered maps to maintain the lexicographical order of the suffixes. I found it to be easier to understand to simply rebuild the tree in the sorted way rather than editing the old trie to include sorting. Since the number of nodes in the tree will never exceed n (the number of characters in the text), this maintains a linear time construction.

Due to time constraints, I did not implement the optimization algorithm suggested in the paper for b values greater than the square root of n. In addition, I did not implement the optimization for LCE calls; I simply used a brute force method.

Flaws in this implementation involve the inefficiency of the big integer class I used, called InfInt. The original algorithm assumes constant time for big integer calculations, however in my implementation, I did not optimize the multiplications for big integers to constant time. In addition, relying on built-in C++ data structures caused some slowdowns, particularly with vector functions.

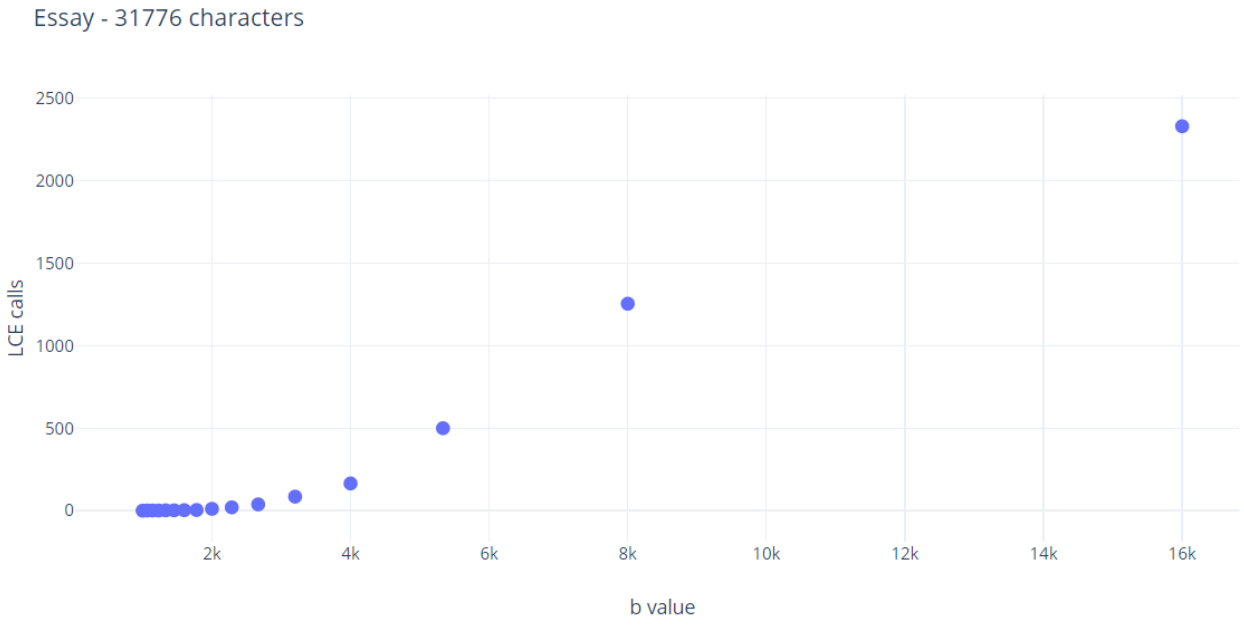
Results

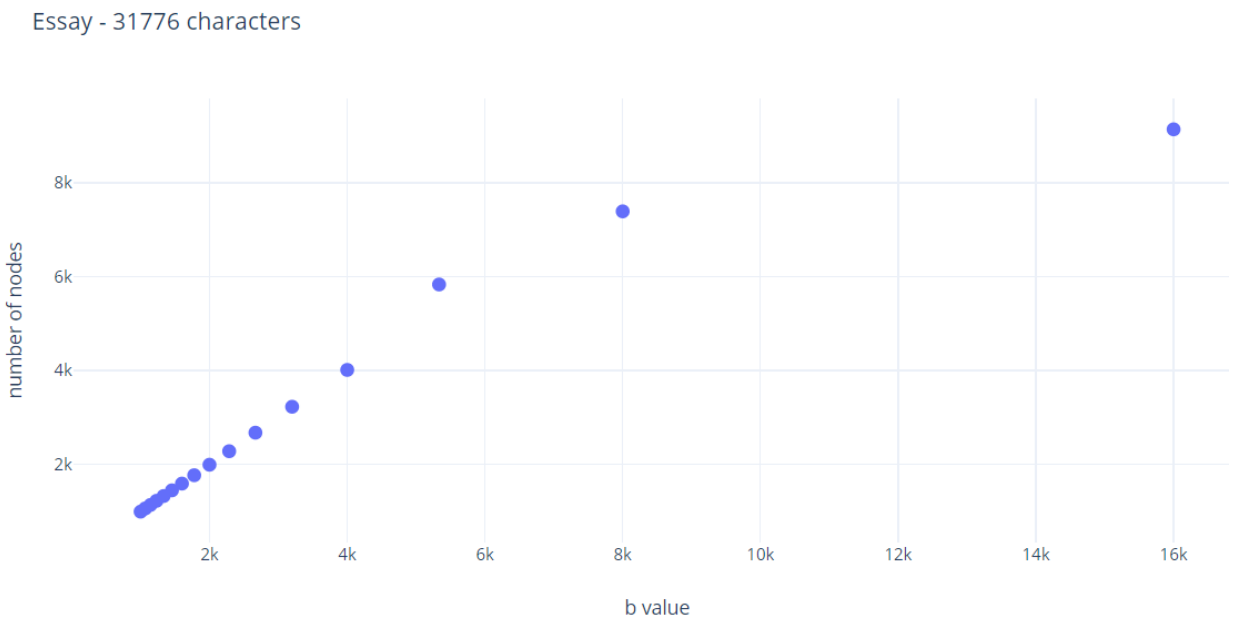
Running this program on randomized text and comparing it to real-world text reveals interesting behavior. I recorded the number of LCE calls, which represent the complexity of the sparse suffix tree's structure. I also recorded the number of nodes in the final sparse suffix tree on various texts.

Essay – 128 ASCII alphabet

I ran the sparse suffix tree algorithm on a 15-page paper I wrote for my women's and gender studies class with the intention of representing narrative English text and how it will affect the structure of the sparse suffix tree. I selected various b values that represented different supercharacter sizes (since the size of the supercharacters is ceiling(n/b)).

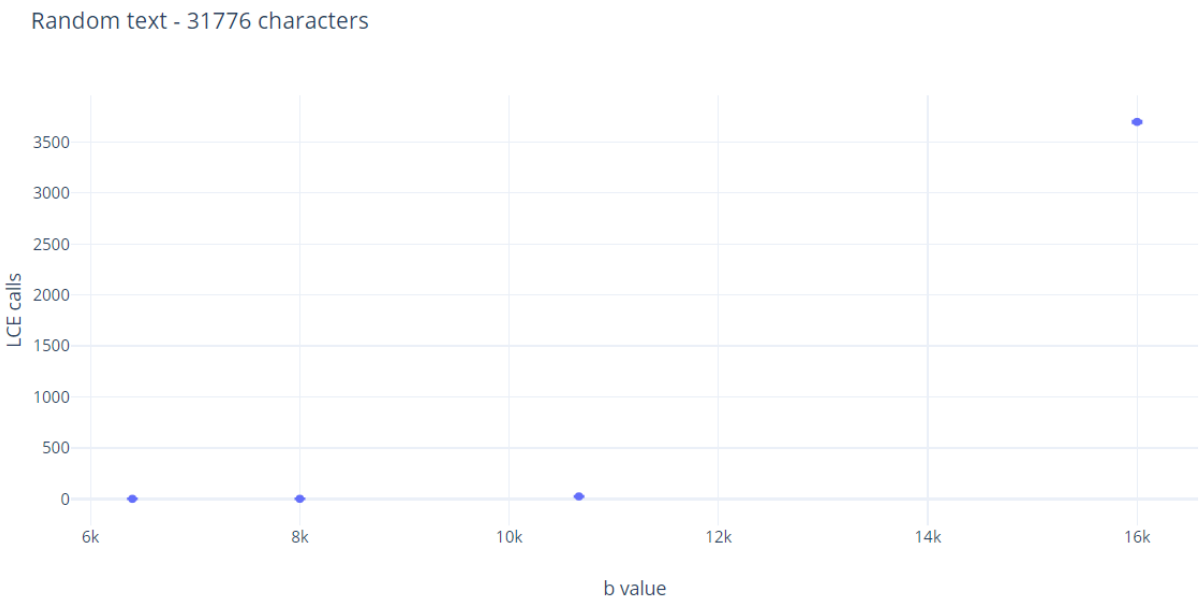
b value	supercharacter length	LCE calls	nodes
16000	2	2331	9142
8000	4	1255	7394
5334	6	500	5835
4000	8	165	4014
3200	10	85	3229
2667	12	38	2676
2286	14	20	2281
2000	16	11	1993
1778	18	4	1769
1600	20	3	1591
1455	22	2	1447
1334	24	2	1326
1231	26	1	1224
1143	28	1	1136
1067	30	1	1061
1000	32	0	993

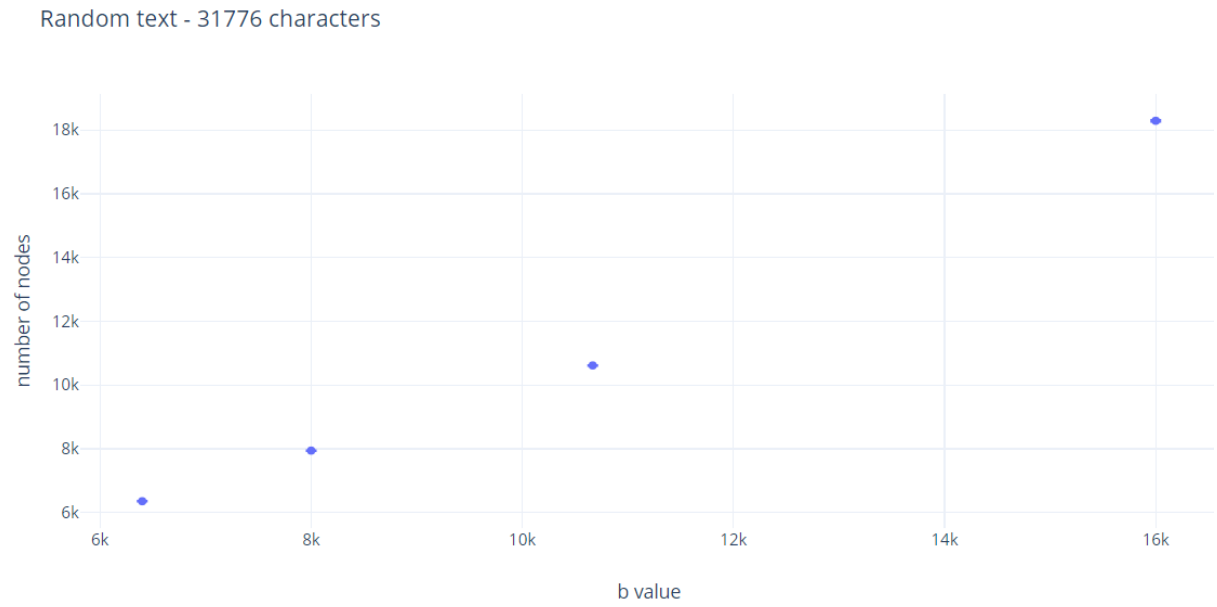




I then ran the algorithm on randomly generated text on the ASCII alphabet to compare its behavior to the English text from the essay.

b value	**supercharacter length**	**LCE calls**	**nodes**			*LCE calls*	*nodes*
16000	2	3683	18305		*average*	3697.8	18288.4
		3679	18247		*std error*	7.330757123	14.91174034
		3703	18269				
		3718	18334				
		3706	18287				
10667	3	25	10617		*average*	23.6	10614.8
		23	10615		*std error*	1.029563014	1.280624847
		24	10613				
		20	10611				
		26	10618				
8000	4	0	7944		*average*	0.2	7944.2
		0	7944		*std error*	0.2	0.2
		0	7944				
		0	7944				
		1	7945				
6400	5	0	6356		*average*	0	6356
		0	6356		*std error*	0	0
		0	6356				
		0	6356				
		0	6356				

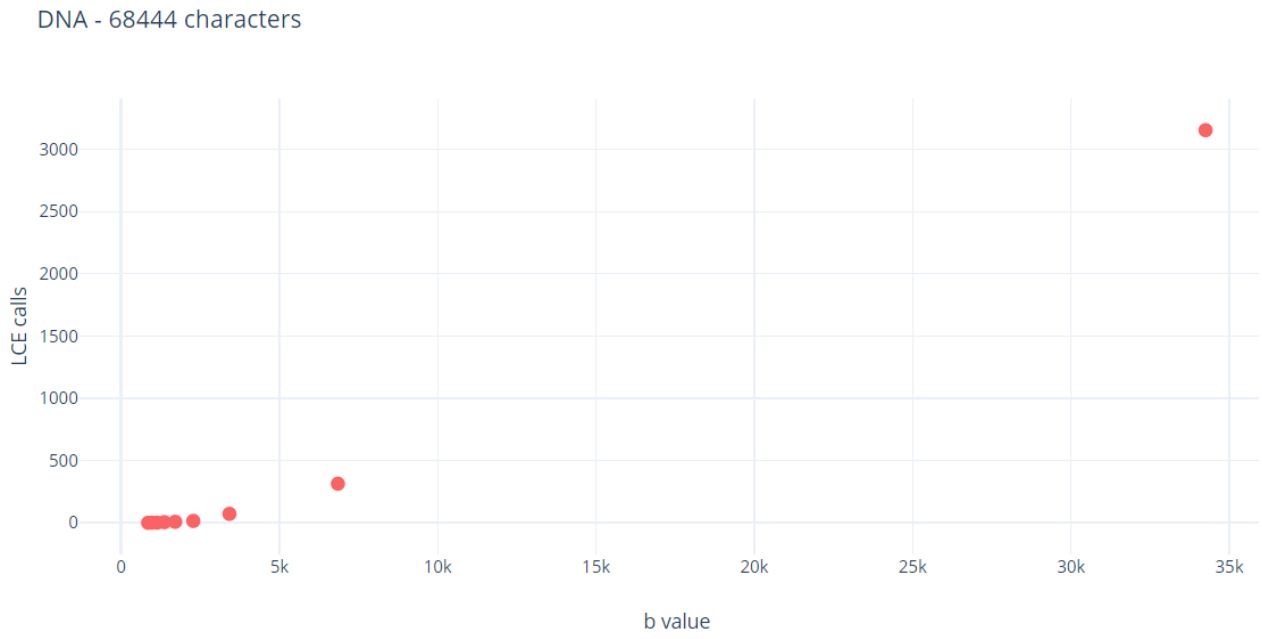


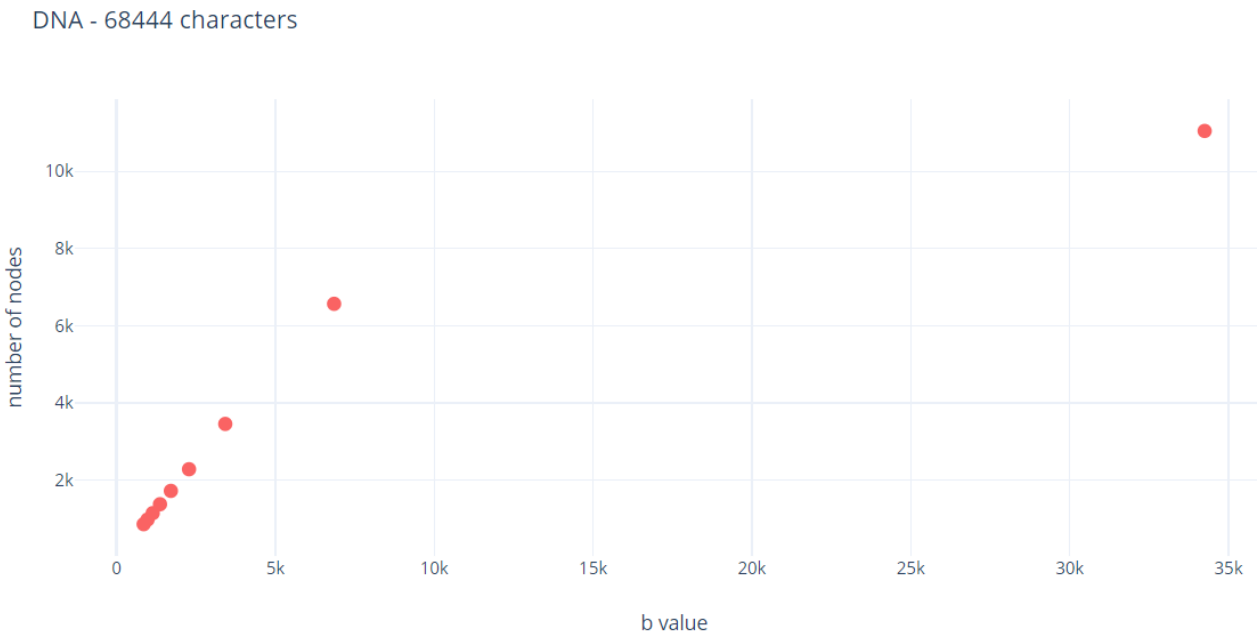


Compared to the English text, the user-selected b value produces almost no LCE calls in the construction algorithm on random text. Anything less than a b value of 10,000 produces a simple suffix tree structure. On the English text of comparable size, however, matches are found in any b value above 2,000. This suggests that English text is much more repetitive than random text, so there are more chances for supercharacters to match up and create a more complex suffix tree structure.

I also ran the algorithm on a segment of influenza DNA. As opposed to English text, the DNA is restricted to an alphabet of size 4 = {A, T, G, C}.

b value	supercharacter length	LCE calls	nodes
34250	2	3153	11046
6850	10	314	6566
3425	20	72	3454
2284	30	15	2280
1713	40	8	1719
1370	50	5	1373
1142	60	1	1142
979	70	1	976
857	80	0	856

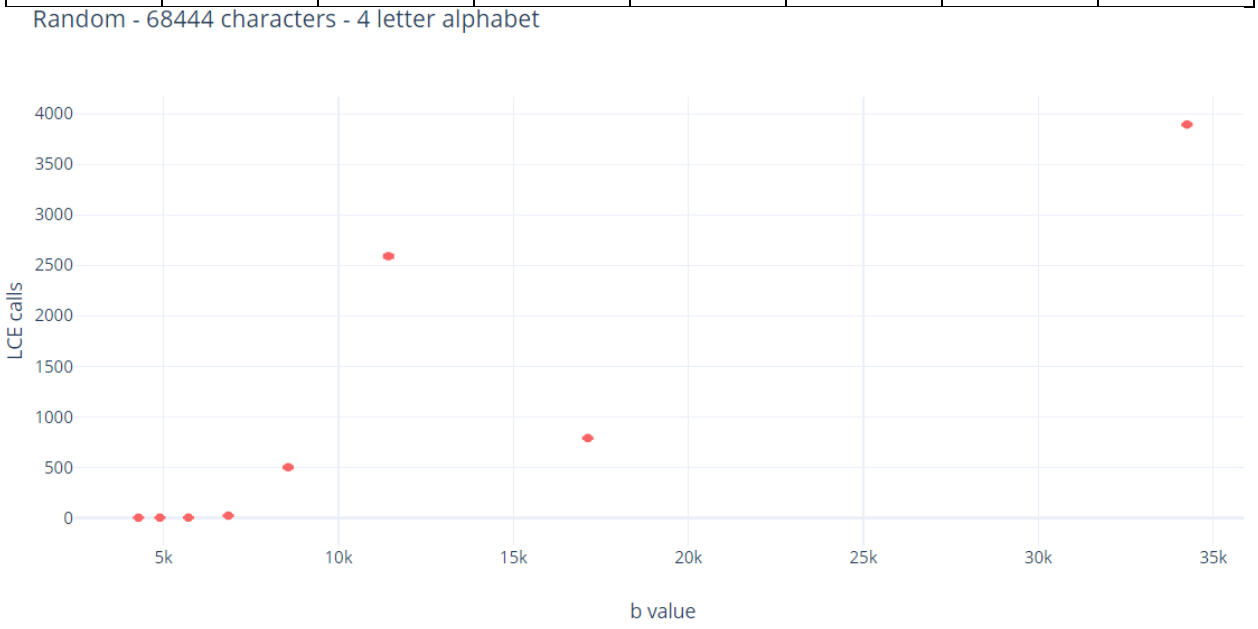


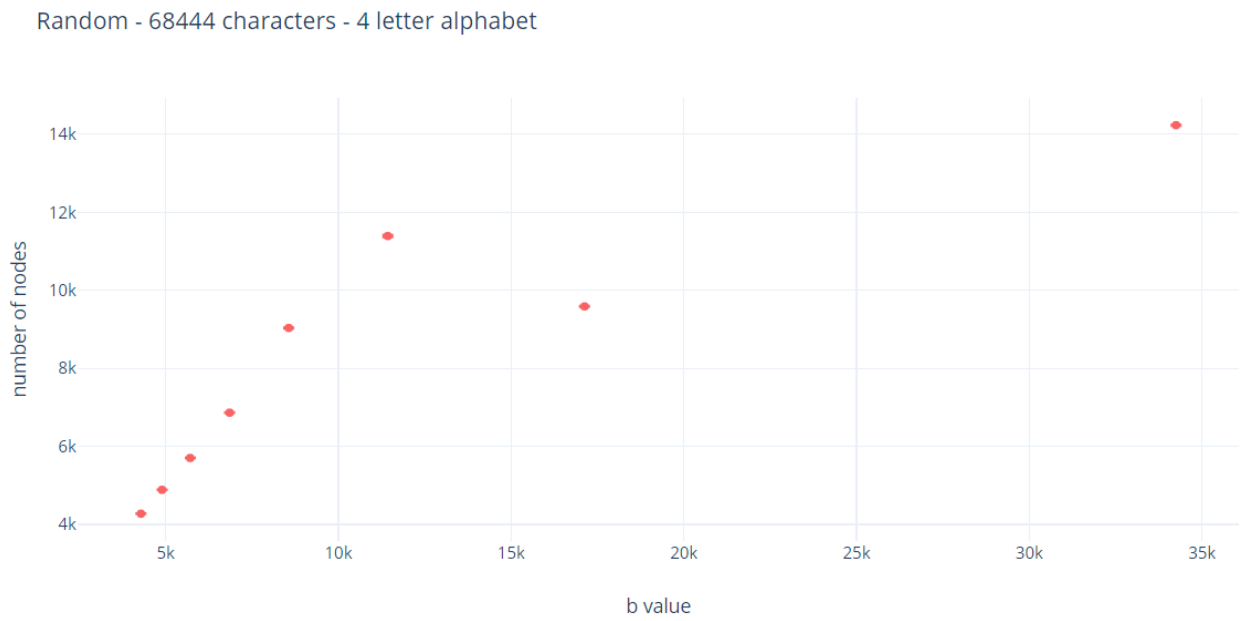


Again, I chose to compare the results of the DNA sample to randomized text on a 4-letter alphabet:

b value	**supercharacter length**	**LCE calls**	**nodes**			*LCE calls*	*nodes*
34250	2	3897	14241		*average*	3894	14231
		3903	14231		*std error*	6.276941931	8.179242019
		3910	14256				
		3885	14213				
		3875	14214				
17125	4	779	9579		*average*	788.4	9587.6
		806	9607		*std error*	4.905099387	5.635601121
		792	9593				
		783	9583				
		782	9576				
11427	6	2575	11385		*average*	2589.8	11391.8
		2635	11463		*std error*	13.01307035	22.41740395
		2568	11346				
		2568	11346				
		2603	11419				
8563	8	498	9033		*average*	500	9035.8
		521	9059		*std error*	7.328028384	7.519308479
		512	9046				

		486	9018				
		483	9023				
6850	10	25	6870		*average*	20.2	6865.2
		22	6867		*std error*	2.853068524	2.853068524
		14	6859				
		27	6872				
		13	6858				
5709	12	2	5706		*average*	1.2	5705.2
		1	5705		*std error*	0.374165739	0.374165739
		1	5705				
		2	5706				
		0	5704				
4893	14	0	4889		*average*	0.4	4889.4
		0	4889		*std error*	0.244948974	0.244948974
		0	4889				
		1	4890				
		1	4890				
4282	16	0	4278		*average*	0	4278
		0	4278		*std error*	0	0
		0	4278				
		0	4278				
		0	4278				

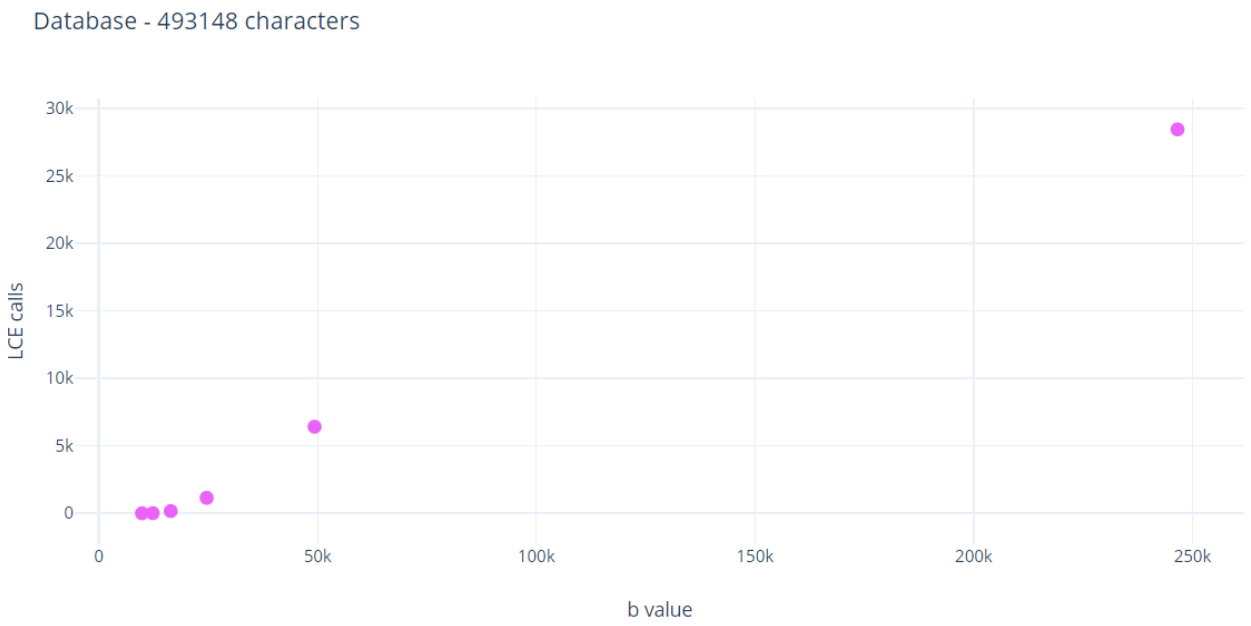


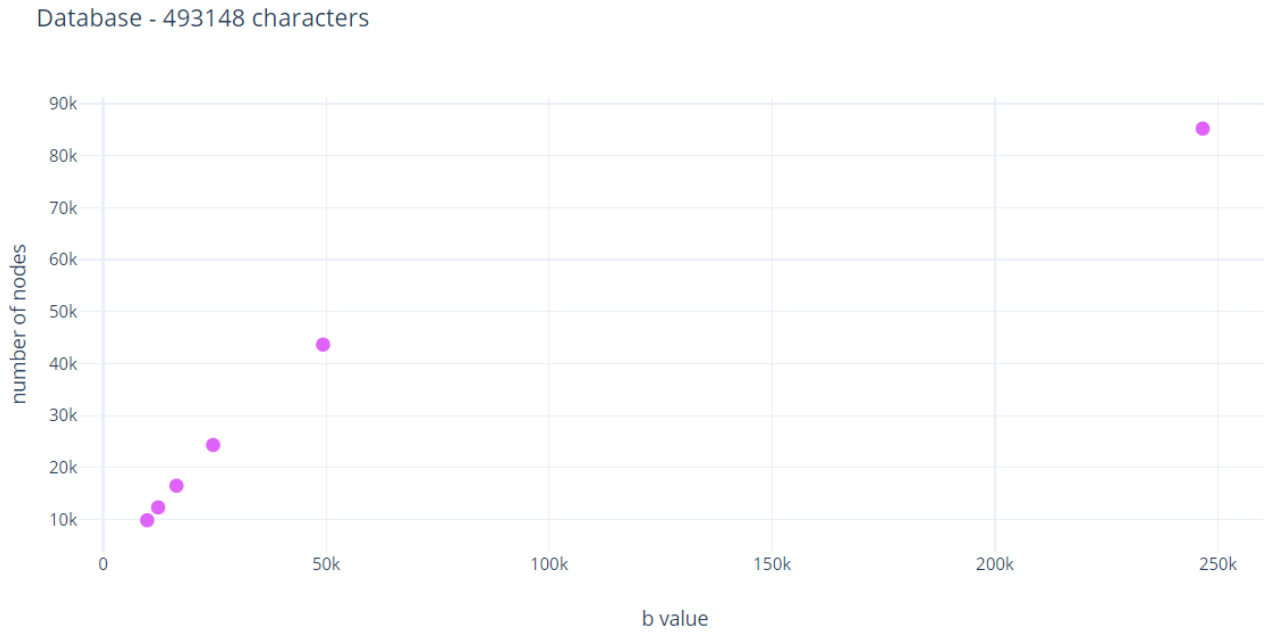


A notable difference between the random text and the DNA sample is the variability in the number of LCE calls on random text. The reason for the unexpected behavior is that the LCE calls represent the number of times a new node is created upon finding a supercharacter match. Once a node is created to represent that match, there is no need to call the LCE or make new nodes for that match again. In other words, the LCE calls represent the number of times a new match is found, not the number of times that same match is repeated within the text. Thus, supercharacters around 6 characters long will actually call the LCE more than supercharacters 4 characters long because with only 4 characters, multiple repeated matches are much more likely. With 6 characters, pairs can be found, but they are less likely to repeat multiple times. Thus each new match must call a new LCE.

Finally, I ran the algorithm on a large database of customers from Book Crossing.

b value	**LCE calls**	**nodes**
246600	28443	85219
49315	6412	43659
24660	1144	24340
16440	161	16510
12330	7	12335
9864	0	9863





The results from this larger database are similar to the results from the essay text!

Open Questions

- What functions can be reasonably optimized to improve overall runtime?
- How does the b value affect the efficiency of the query of a sparse suffix tree?
- What is the most efficient way to query a sparse suffix tree? What are the tradeoffs on accuracy vs runtime?
- Is there a way to incorporate the z algorithm in the construction of the sparse suffix tree?

- If the length of each supercharacter is less than the number of digits in the fingerprint, would it be more efficient to avoid using the fingerprints and just represent edges as supercharacters?
- How does the length of the substrings called by the LCE differ on the texts as opposed to simply measuring the number of function calls there are?