Carl M. Kobel og Mikkel Langhave
Genome Scale Algorithms, BiRC, spring 2019.

Project 3: Construction suffix trees from LCP and suffix arrays

Status of work

We have implemented the gen-lcp that makes the LCP and suffix arrays using the suffix tree that we previously implemented.
We have a search-st2 that uses the LCP array and the suffix array to make a new suffix tree.
We do not support the outputting the LCP and suffix arrays as files and we do not support making the suffix tree suffix from the suffix array as and LCP array as files.

Description of implementation

Each suffix is added in regard to the lcp values of the neighbouring suffixes. When the lcp of a suffix is zero, the suffix is added to the root, when the lcp increases, the suffix is added to the latest node, when the lcp is unchanged the suffix is added to the latest parent, and lastly when the lcp decreases, the program jumps up through old parents via a parent stack, which is only saved since the last visit to the root.

Insight during implementation

Using indexes is difficult, which is why the algorithm is implemented so that each node contains the string it represents.
Taking care of all edge cases while construction the suffix tree from the lcp tree is difficult and need to good test cases.

How to run the program

The program can be run exactly like the last hand-ins. The files used in the following example will be present in the hand-in directory.

```
machine:3_lcp_suffix_tree student$ python3 search-st2.py data/seqs.fasta data/reads.fastq
iss     0       one     2       0       3M      *       0       0       iss     ~~~
iss     0       one     5       0       3M      *       0       0       iss     ~~~
mis     0       one     1       0       3M      *       0       0       mis     ~~~
ssi     0       one     3       0       3M      *       0       0       ssi     ~~~
ssi     0       one     6       0       3M      *       0       0       ssi     ~~~
.
  .
    .
      <etc.>
```

Description of the machine

The running time analysis was done on a laptop running mac os 10.12, with a i5-4258U, 2.40GHz CPU and 8 GB 1600 MHz DDR3 RAM.

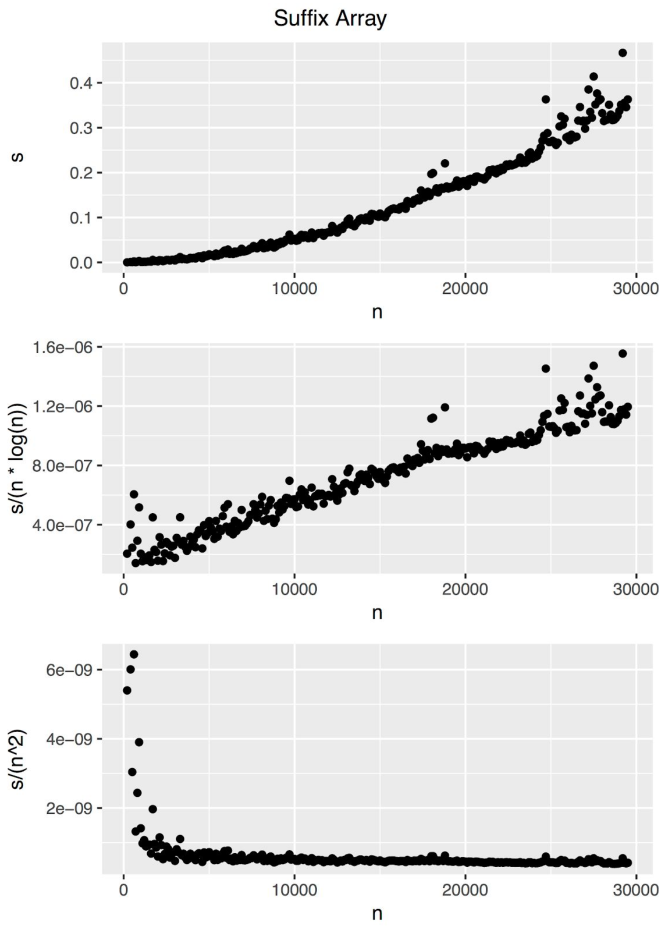
Correctness of program

We have tested that the program works with “mississippi” and “sassasass” and other “small” cases and compared “manually” with the results from the previous handling.

Description of tests

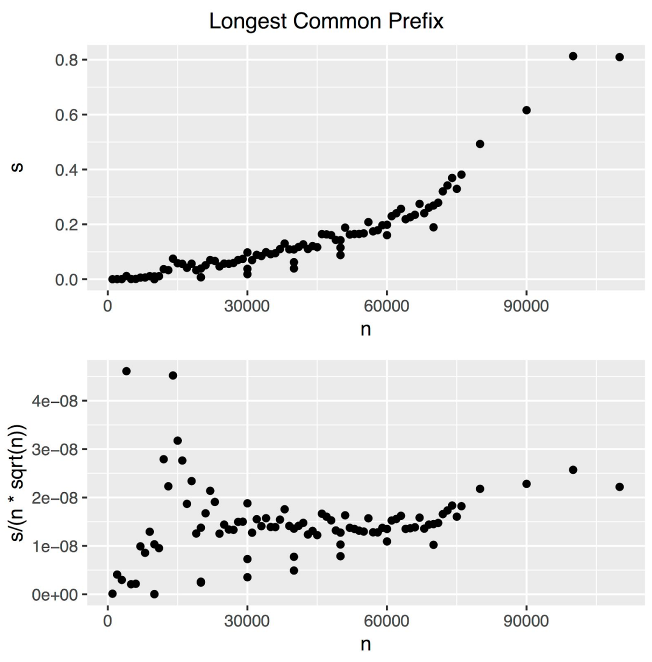
We have made time tests of the construction of lcp arrays , suffix arrays and making of the suffix tree separately.

1 - Suffix array



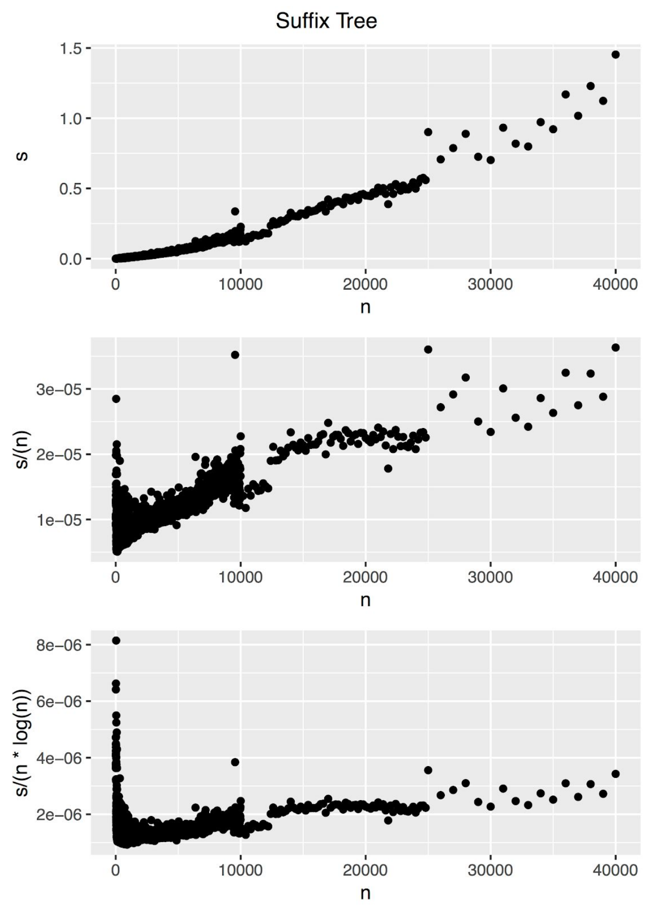
The suffix array test was done with a random genome, and not a worst case test.
We expect it not to be $O(n\ log\ n)$ because of the comparisons when sorting.

2 - LCP arrays



The LCP array construction was run on the case where alfabet size was one.
Because of the long time to make comparisons we should theoretically have $O(n^2 \log n)$ running time. $O(n \log n)$ comparisons in the sorting part of the algorithm and $O(n)$ for each comparison. But when we test that case the running time seems to be around $O(n\sqrt{n})$.

3 - Suffix tree



This run is on a random genome, the worst case should be making where the LCP starts high and falls but we had difficulties making such a string.
The construction should be $O(n)$ if indexes are used and everything is perfect, but because every node stores a part of the string and not the indexes this implementation is slower.
As the graph shows it is not $O(n\ log\ n)$ but $o(n^2)$

Conclusion

Because the suffix tree is constructed with the string in each node, and not with indexes, the running time of the algorithm is not exactly linear, but closer to O(n * log(n)).