CS 571: Suffix Tree Implementation Report

Jesse Waite

Spring 2016

1 System Specs

All testing in this report was performed under the following system specs

- Intel i5 2.2 Ghz
- 8 GB RAM
- 64bit Windows OS
- Microsoft vs c++ compiler, no optimizations

2 Tree Construction: Empirical Time Complexity Analysis

The following results were obtained building suffix trees for the brca2 gene, Tomato chromoplast, and the yeast gene.

Input	Build Time (ms)	Space (bytes)	Space/Time (bytes/ms)
brca2	348	11383	32
tomato chromoplast	4849	155462	32
yeast gene	30908	959366	31

The above numbers are for Windows 10, whereas under Ubuntu/g++ with -O2 optimizations the build time of the largest Yeast.txt input was 0.6s compared with 30s under Windows/vsc++. So for whatever reason, my code

was vastly faster under Ubuntu/g++, even without compiler optimizations.

But given these figures, the processing rate for the tree average around 32 bytes/ms with respect to the input size. These figures are clearly linear with respect to the input size of each string, as quadratic time would have required far more time as input size was increased. This was consistent with the expected linear run-time of McCreight's algorithm, since it leverages suffix links to amortize the benefits of previous suffix insertions. However, my implementaton included a somewhat inefficient use of vectors to store lexicographically-ordered out-edges from each node, which was inefficient from both a systems and an algorithmic perspective. So even the numbers above, though good, could be improved through code optimizations in terms of better class design, and better node/edge insertion methods.

3 Space Complexity

As reported by my output, the class space overhead was as follows:

- SuffixTree: 48 bytes
- TreeNode: 32
- Edge: 12

For each input, the total reported suffix tree size was as follows:

Input	Size (bytes)	Tree Size	Space Constant (tree size / input size)
BRCA2	11383	822088	72
Tomato chromoplast	155462	11195176	72
Yeast chromosome	959366	69565268	72

Again, this was consistent with what was expected. The required TreeNode class was a bit bloated and likely could be slimmed down, but for our purposes the node class was easy to understand and implement. However, when monitoring actual process memory usage, the memory usage was much

higher than reported by mutiplying and summing class overhead by the number of instances of each class. I'm fairly certain this extra overhead was because of the internal memory allocation strategy of certain data structures. Foremost, I use vectors to store the out-edges for each node, which was a poor choice of data structure, and most vector implementations also allocate an internal buffer capacity much larger than their curent size, usually maintaining an internal size up to the next power of two with respect to the current size.
As a result, the tomato chromosome, despite being a string of only 1MB, consumed about 250 MB on my machine, despite the raw class overhead statistics. Again, this is almost certainly due to internal overhead for the vectors within each TreeNode, which was the only dynamic datastructure I used other than the tree itself.

4 Burrows Wheeler Transform

The BWT for each input is attached in separate files, under the appropriate names per the test input file names (BRCA2_BWT.txt, Tomato_BWT.txt, and Yeast_BWT.txt).

5 Longest repeated substring

The longest repeated substring is reported as the longest substring that occurs at last twice in the input. The correct way to find such string would be to use the Bender-Farach method, but logically the process entails searching for the lowest internal node in the suffix tree. The concatenation of all path labels from the root to this node is necessarily the longest repeated substring in the suffix tree.
We know that this node represents the longest repeated substing for the following reasons. Assuming there was any lower internal node, this would imply that there existed some internal node representing a longer path, and therefore a longer repeated substring. Clearly this is a contradiction, so the node must be the deepest internal node. Further, this node's children are all leaves, for the same reason. And since they are all leaves, the point at which these two suffixes differ occurs at the end of the longest repeated substring.

Accordingly, find the deepest internal node in brute force fashion can be done either by tracking the deepest node during the build procedure (preferred), or by searching the tree using DFS or BFS to find the deepest internal node. In my implementation, I chose to keep the search procedure separate from the build procedure, and used BFS to find the deepest internal node. I then used info in this node and one of its children to determine the longest repeating substring and one set of indices representing this string in the input.

The longest repeating substring for each test input was as follows:

- Test1 (BANANA): (len 3) ANA
- Test2 (MISSISSIPPI): (len 4) ISSI
- BRCA2: (len 14) AAGAGATACAGAAT
- Tomato: (len 48) AATCAATGCAATTTAGGAGGAATCAATGCAATT-TAGGAGGAATCAATG
- Yeast: (len 8375) CTCATGTTTGCCGC[...middle portion omitted for brevity...]CTCAGGTGCTGCA. The full yeast longest repeated substring can be found in the tail of Yeast_Test.txt.

6 Conclusion

In this lab, we built a suffix tree and analyzed the space and time complexity of our output. As shown previously, my constuction time and space complexity were both linear with respect to the input size, although an optimized implementation could likely improve on the coefficient of this linear time implementation.