string-algorithms handin 1 - Suffix Trees

Marc Skodborg, 201206073
Stefan Niemann Madsen, 201206043
Simon Fischer, 201206049

May 8, 2016

1 Implemented algorithm

We have only implemented the naive suffix tree construction algorithm with running time $O(n^2)$ for now. We have not had the time to improve on the implementation beyond the basics either. For instance, the edges still contain the full string rather than indexes into the reference string on which the tree is built, which could have saved memory.

2 Insights

We quickly discovered the value of having a way of visualizing what the program, at any given moment during the implementation, believes the suffix tree is supposed to look like, to pinpoint mistakes and verify correct behaviour more easily.

Therefore, we spent some time integrating `GraphViz`, a visualization tool used to visualize the current tree, which shortened implementation time significantly.

3 Problems encountered

None worthy of mentioning.

4 Memory specifics

Bytes used per suffix node (and per input character, on average, in the strings you have experimented with),

The overhead on the individual elements of our tree data structure, the `Node`-objects we instantiate and connect, has a size of `56 bytes` each, as reported by Python's `sys.getsizeof()` function. We will have a maximum of $2n-1$ nodes in the tree, i.e. $(2n-1) \cdot 56\texttt{bytes} \approx n \cdot 112\texttt{bytes}$ where n is the length of the string.

Although, as `sys.getsizeof()` only calculates the memory print of an object and its references without following these, and we have the full string on an edge saved as a reference from each Node, we need to include these too explicitly.

Python uses `49 bytes` to hold the empty string in memory, as stated by calling `sys.getsizeof('')`. This is exact, as the function promises to work correctly for built-in types. Each character the string is expanded by increases the size of the string by `1 byte`

A tree with a maximum of $2n-1$ nodes will have a maximum of $2n-2$ edges. Each of these is a string with a base memory print of `49 bytes` in python, plus one additional byte per character in the string. These varies, but will not ever become longer than the initial string, as they are all substrings.

In total, this yields a conservative approximation of $n \cdot 112$ `bytes` $+ (2n-2) \cdot (49+(n+1))$ `bytes` in memory for storing a suffix tree using our implementation on an input string of length n

5 Construction & searching estimates

The construction of the tree when given the text in `ancient-mariner.txt` as input takes roughly 2.22 `sec`, based on a few runs to minimize process priority influence from the OS.

Searching for the text *Albatross* in the generated tree is successful after approximately 0.1 `ms`