Eldon Lake

Description

The suffix automaton (SA) of a string S is the smallest possible deterministic finite automata that can recognize every possible suffix of S. By traversing the automaton, we can calculate & store information associated with each state that allows us to efficiently solve several problems related to strings. The problems I have included to test my implementation are:

1. Checking if a string T is a substring of S
2. Finding the position of the first occurrence of a substring T in S
3. Finding all positions of a substring T in S

SA can be used to solve problems including but not limited to finding: the size of the set of unique substrings in S, the longest common substring between S and another string, the shortest possible string not found in S, the kth substring in lexicographical order, and many more.

Complexity

Short answer: Given an alphabet size k, my implementation achieves O(n) memory use and O(nlogk) time complexity for the initial construction of the SA, or O(n) time complexity if you consider k to be constant. The methods that use the SA to solve problems have the following complexity: Checking if a string T is a substring of S takes O(T) time. Finding the position of the first occurrence of T takes O(T) time. Find the list of positions for every occurrence of T takes an initial preprocessing step of O(n), however all subsequent queries for any number of distinct strings T then take O(num(T)) where num(T) is the number of occurrences.

Long answer with peculiarities: After mild over analysis, I initially decided to use a vector to store transitions, and just do linear lookups. I did later include a version with maps, so I could rightfully say I implemented an O(nlogk) solution, but let's ignore that for now. The linear search vector solution creates an O(k) worst case per-operation when it checks for a given transition / state, meaning your theoretical worst-case construction overall could be O(nk) if not amortized. My rationale was that the average number of transitions per state would be so low that the real-world performance benefits of sequential data searches (cache hits, etc) would outweigh the theoretical time complexity advantage of a binary search tree, hash table, etc. The average number of transitions for each state cannot exceed 3, and in practice tends to be less than 2 for large strings. That number is deceptively low, however, as states with more transitions tend to represent a disproportionate number of lookups during construction. I ran tests on many random mixed-case strings up to size 10E6, and the average number of transitions that the linear search could check in each case maxed out between 18-20, though obviously it stops as soon as it finds a match and does not check them all. The structure of natural human languages brings this down substantially. On large public domain English texts (the Bible, novels, etc), the average number of transitions in each search is only 3-4. As an English-speaking human who rarely consumes giant blocks of random characters, this sealed the deal. I did some rough benchmarks comparing the vector version and the map version, which you can see at the end of this document.

Eldon Lake

Resources

- The Wikipedia article on SA is great and covers the most useful information including discussions of complexity, an overview of construction (both verbal and pseudocode), and links to a ton of great references. https://en.wikipedia.org/wiki/Suffix_automaton
- CP-Algorithms has more detailed verbal descriptions of the algorithm and some of the approaches used to solve problems with it, as well as sample implementations. I chose not to look at their implementations until after I had working solutions of my own, but their verbal descriptions ended up being my primary reference for building my implementation, and the language of my comments mirrors theirs accordingly. https://cp-algorithms.com/string/suffix-automaton.html
- I found this paper really interesting. It gives some insight about how suffix automatons can be used to solve problems you would not necessarily think of as being string related. The example they look at in depth is music identification.
  Mohri, M., Moreno, P., & Weinstein, E. (2009). *General suffix automaton construction algorithm and space bounds. Theoretical Computer Science, 410(37), 3553–3562.* https://doi.org/10.1016/j.tcs.2009.03.034

Assumptions

1. My implementation assumes that only non-empty strings are provided as input for the construction of the SA or as input for the supported queries.
2. I assume that your input consists of a string of chars, which limits the alphabet size to 256.
3. I used ints as indexes, meaning you cannot have more than 2E31-1 states. The maximum number of states is 2n-1, so the maximum input size for your string is quite large at 2E31/2

Eldon Lake

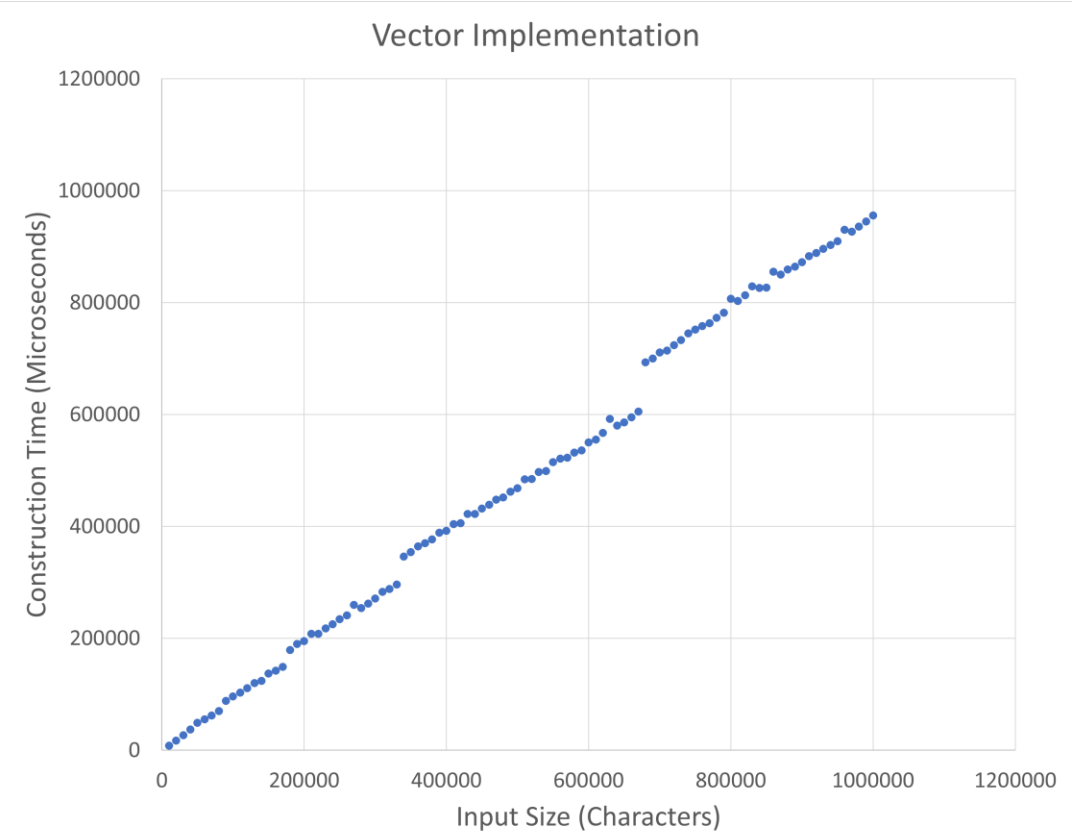
Instructions

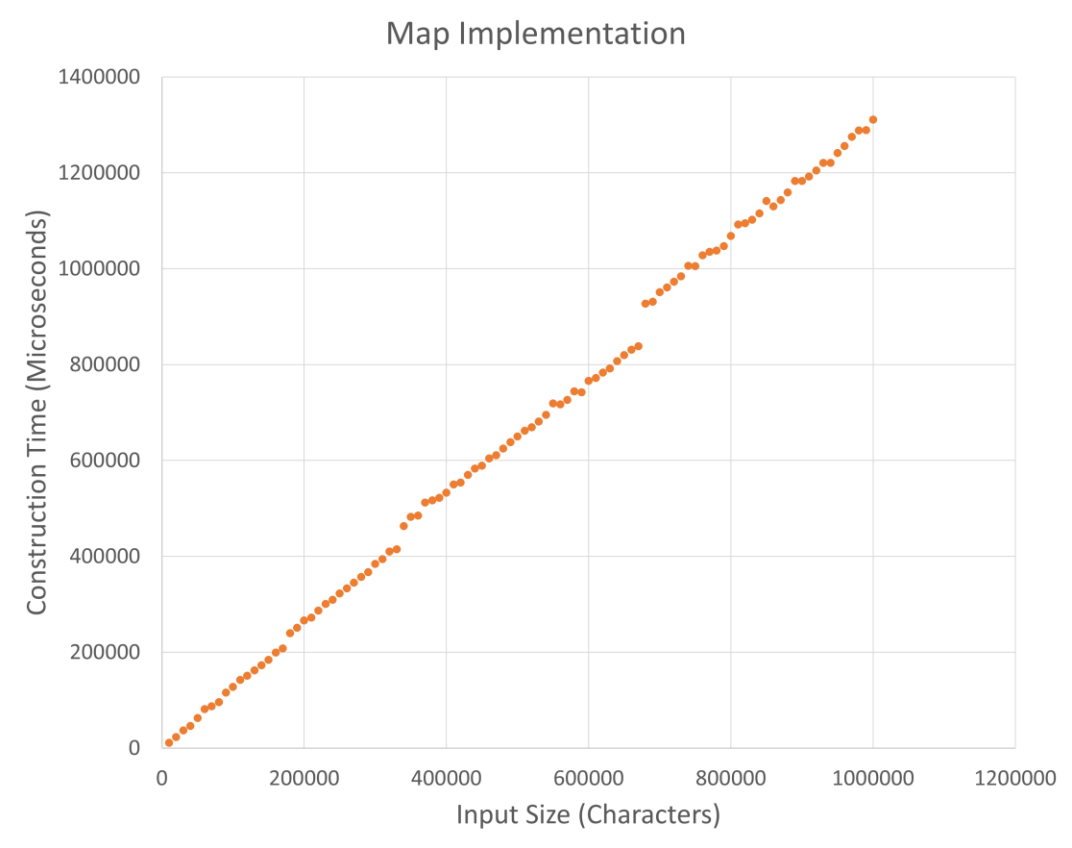
- Use Linux. The implementations themselves are portable, but the tests assume unix line endings and the make files were made for Linux, so if you want to use Windows, you'll need to compile yourself and mimic the testing commands from the make files.
- Interactive Testing: SuffixAutomaton.cpp contains a main function that will allow you to input your source string to construct the automaton, and then allow you to select from the available queries and enter your search string. Your results will be displayed in the terminal.
    - Interactive mode can be executed by running "make run"
- Automated Testing: There are three source files used for automated testing: PositionTest.cpp, VectorTiming.cpp, and MapTiming.cpp. The *.in files are used as inputs for 7 automated tests.
    - Automated tests can be executed with "make testK" where K is a number between 0 and 6. E.g "make test0" … "make test6"
- Input Generators: This folder contains the original public domain texts I used for testing purposes, as well as some Python scripts I used to generate the input files. You won't need any of this to run the tests, but if you want to generate your own test data, there you go.
- Automated Test Descriptions:
    - 0: Correctness tests – Constructs automatons based on a variety of variable length input strings, and then uses the positions(S) method to find all of the positions of select substrings. The input data contains the correct number of occurrences for each substring, and the main function checks each position that's produced to confirm that the substring is found at that location in the source text. I had originally planned to use Heidegger for this, because his tendency for hyphenating would have made for some great suffix humour, but Disney won't let his works slip into public domain, so I had to go with Freud. Pretty low hanging fruit if I'm being honest.
    - 1-3: Timing tests – Constructs 50 automatons with increasing length strings from 10,000 to 500,000 and measures the time it takes to construct. They will output the ratio of execution time to input size for each one, so that you can see the linear relationship. Timings are using chrono::high_resolution_clock so you may seem some variance based on system load and other external factors, but it's accurate enough to be useful. Test 1 uses English text, 2 uses specific strings that produce the largest possible number of *states* for their input size, and 3 uses specific strings that produce the largest possible number of *transitions* for their input size.
    - 4-6 Timing tests – The same as 1-3 but with the map implementation for comparison.

Eldon Lake

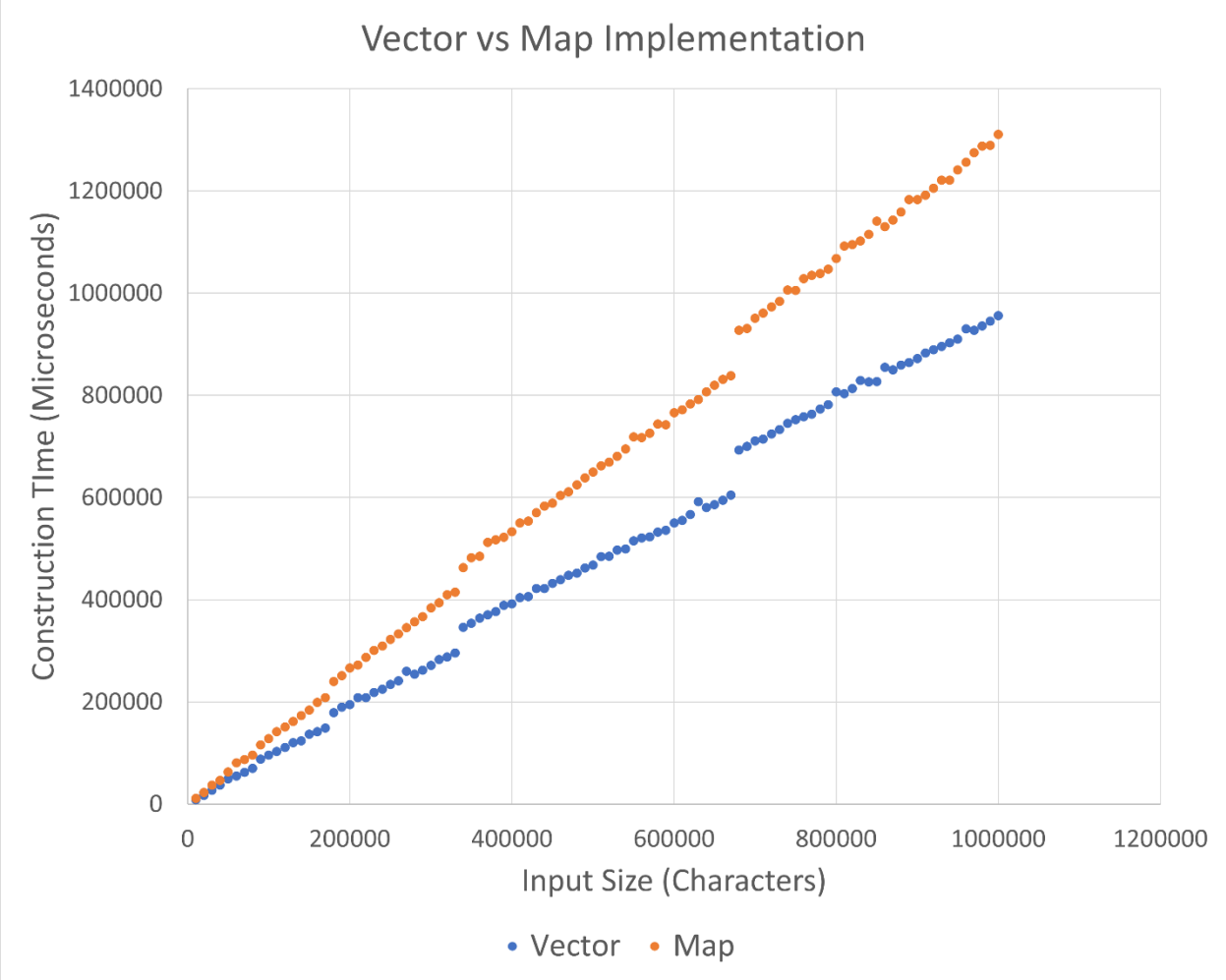
Benchmarks

I have included some charts of the results of the timing tests so that you can see visually that the execution time is linear (providing constant k), and the difference in performance with vector vs map.





Eldon Lake


Vector vs Map Implementation
Construction TIme (Microseconds)
1400000
1200000
1000000
800000
600000
400000
200000
0
0
200000
400000
600000
800000
1000000
1200000
Input Size (Characters)
Vector
Map