Programming Assignment 1: The Suffix Array

Student: Jessica Zhu

Link to code: https://github.com/jhzsquared/CMSC701\_hw1

Part 1: Building the suffix array

The most challenging part of implementing the ‘buildsa’ program, besides it being my first time coding a project in Java, was using protocol buffers to binary encode the suffix array output. This was also my first time building my own protocol messages, so it was challenging to ensure the data types for Java and Protobuf were aligned when I did not have a complete understanding of the data types available in either language, nor a strong understanding of how to read and write Protobuf messages. However, it’s what I arbitrarily chose to use and eventually figured it out.

I experimented with three different reference genomes of varying sizes. Table 1 displays the results, listing the size of the reference genome, time it took to build the suffix array, time it took to build the prefix table (for varying prefix tables), and total output file size. I ran out of memory for building a length 16 prefix table for the mosquito reference genome.

The space it takes to build the suffix array increases with only slightly higher than linear rate compared to the size of the reference. However, the time it takes to build the suffix array increases with significantly faster than linear rate, though less than quadratic time, as the reference size increases.

The time it takes to construct the prefix tables converges to near constant time as the reference sequences and length of prefixes (k) increase. The additional space the prefix table takes is near negligible for sufficiently small prefix size (<8 characters),but increases significantly as the number of permutations explode.

On a machine with 32GB of RAM and assuming I can allocate all of it to this program, I

could probably construct the suffix array for a 900,000 character sized reference genome, if I did not need to construct the prefix tables. I had to allocate 6GB of maximum heap space to build suffix arrays for the mosquito reference genome with up to k=4 sized prefix tables. I had to allocate 7GB for k=8, which resulted in an output of almost the same size as the k=4 sized prefix table. This means almost the full 6GB of RAM was likely consumed during creation of the basic mosquito suffix array. As such, given the slightly greater than linear rate increase in space that the suffix array consumes and the 5+-fold increase in RAM allocated, we would expect to be able to construct a reference genome’s suffix array that is almost 5 times as large. A 900,000 character sized genome is about (less than) 5 times the size of the mosquito reference genome.

Table 1: Time and file size for variants of the suffix array

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Genome Size (# of characters)** | **Time for SA (ns) /Size (kb) (no prefix table)** | **Time for prefix table (ns) / Size (kb) (k=2 prefix table)** |
| **Covid** | 29,751 | 19,159,300 / 101 | 4,922,100 / 101 |
| **Ecoli** | 4,639,675 | 2,010,129,600 / 20,591 | 123,519,200 / 20,591 |
| **Mosquito** | 184,084,636 | 221,604,163,800 / 896,787 | 19,863,672,400 / 896,787 |
|  | **Time for prefix table (ns) /Size (kb) (k=4 prefix table)** | **Time for prefix table (ns) /Size (kb) (k=8 prefix table)** | **Time for prefix table (ns) /Size (kb) (k=16 prefix table)** |
| **Covid** | 8,885,300 / 105 | 24,924,400 / 526 | 31,326,800 / 939 |
| **Ecoli** | 183,521,900 / 20596 | 251,207,400 / 22,066 | 3,707,114,600 / 158,561 |
| **Mosquito** | 19,614,961,100 / 896,792 | 18,458,269,000 / 898,323 | OOM |

Part 2: Querying the suffix array

The most challenging part of implementing the query program was making sure I incremented the binary search left and right limits correctly by adding or subtracting by 1. Prior to correct implementation, I found my program getting trapped in the where loop by edge cases.

Tables 2 through 4 breakdown the amount of time it takes for the queries to run on varying sized references, queries, and prefix tables. Queries were generated by pulling random strings from the ecoli reference. As such, matches did not exist in all the references.

The speed of the naïve algorithm was only marginally slower than the simple accelerant algorithm on small queries and small references (without prefix tables) (e.g. ecoli and covid). With larger references and smaller queries, the simple accelerant was significantly faster (up to 2x). However, as the query size increased, the naïve algorithm was comparable if not faster than the simple accelerant.

The prefix table increased speeds for up to a maximally large k (approx. k<8), at which point there were diminishing effects, if not slower speeds. However, increases in query length pushed the prefix size at which slower speeds occurred. In other words, with sufficiently large queries (e.g. 1000 characters), both naïve and simple accelerant algorithms benefited from the prefix table

Given the memory and time requirements demonstrated, assuming a larger sized reference (e.g. ecoli or larger), I think trading off a marginal amount of space and time to compute a small sized prefix table (e.g. k=4) would be worth the potential significant speed up for varying sized queries. I also think implementing the simple accelerant over the naïve algorithm is worth the additional storage of LCP sizes, as it is almost always faster.

Table 2a: Time(ns) for naïve algorithm on Covid (29751 char)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Prefix size/query length** | **No prefix table** | **K=2** | **K=4** | **K=8** | **K=16** |
| **3** | 173800 | 195300 | 1201700 | 29368200 | 31679700 |
| **10** | 71200 | 84500 | 141800 | 40300 | 32027500 |
| **100** | 30400 | 44400 | 50600 | 6200 | 20600 |
| **500** | 46700 | 26800 | 31300 | 5800 | 4800 |
| **1000** | 100200 | 96500 | 134500 | 22100 | 13400 |

Table 2b: Time(ns) for simple accelerant algorithm on Covid (29751 char)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Prefix size/query length** | **No prefix table** | **K=2** | **K=4** | **K=8** | **K=16** |
| **3** | 162100 | 198500 | 866200 | 21475800 | 26291000 |
| **10** | 45500 | 137400 | 75000 | 27800 | 32111100 |
| **100** | 38500 | 42300 | 33500 | 6100 | 20200 |
| **500** | 47100 | 42600 | 33200 | 3700 | 5400 |
| **1000** | 151900 | 148800 | 159600 | 10100 | 15700 |

Table 3a: Time(ns) for naïve algorithm on Ecoli (4639675 char)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Prefix size/query length** | **No prefix table** | **K=2** | **K=4** | **K=8** | **K=16** |
| **3** | 9464600 | 10879300 | 12106500 | 48647400 | 4236456800 |
| **10** | 114000 | 123300 | 105800 | 105400 | 4543208700 |
| **100** | 85100 | 75900 | 37600 | 66300 | 31400 |
| **500** | 101000 | 99600 | 61500 | 31300 | 24300 |
| **1000** | 84900 | 84400 | 86100 | 46700 | 36900 |

Table 3b: Time(ns) for simple accelerant algorithm on Ecoli (4639675 char)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Prefix size/query length** | **No prefix table** | **K=2** | **K=4** | **K=8** | **K=16** |
| **3** | 8459000 | 10879300 | 13405400 | 43083700 | 4222880900 |
| **10** | 83800 | 123300 | 110700 | 111100 | 4504611800 |
| **100** | 66200 | 75900 | 107300 | 39700 | 39100 |
| **500** | 122400 | 99600 | 43000 | 32700 | 50800 |
| **1000** | 82000 | 84400 | 134900 | 67300 | 48100 |

Table 4a: Time(ns) for naïve algorithm on Mosquito (184084636 char)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prefix size/query length** | **No prefix table** | **K=2** | **K=4** | **K=8** |
| **3** | 183,121,100 | 77,719,200 | 250,610,900 | 148,952,500 |
| **10** | 238,700 | 266,000 | 391,300 | 208,400 |
| **100** | 253,600 | 111,000 | 56,300 | 42,700 |
| **500** | 342,200 | 82,200 | 65,000 | 33,000 |
| **1000** | 276,700 | 351,200 | 233,100 | 134,000 |

Table 4b: Time(ns) for simple accelerant algorithm on Mosquito (184084636 char)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prefix size/query length** | **No prefix table** | **K=2** | **K=4** | **K=8** |
| **3** | 94,926,300 | 211,722,000 | 248,581,500 | 292,948,800 |
| **10** | 191,100 | 249,900 | 238,400 | 223,500 |
| **100** | 161,300 | 168,700 | 105,200 | 54,500 |
| **500** | 133,700 | 212,200 | 55,600 | 47,500 |
| **1000** | 759,700 | 118,600 | 192,700 | 202,800 |