CMSC858D

Algorithms, Data Structures and Inference for High-Throughput Genomics

Homework 2

Ataberk Donmez
Student ID: 118208473

Github link: https://github.com/atabeerk/HTG/tree/main/hw2

Introduction

This project consists of two main parts: `builds` and `querysa`. Each part has their own class and header files along with their driver files. The driver files simply parse the command line arguments, call the functions from the class files in the correct order and save the results to a file. Correctness of the programs are tested against the given example input/output files. Programs are compiled without optimization flags.

buildsa

Description

Compiling the `SA-driver.cpp` creates an executable named `buildsa`. This executable takes the command line parameters described in the assignment description. If the argument `--prefix k` is passed, a prefix table for size `k` is created and the total execution time is affected by the value of `k`. Otherwise creating the suffix array takes less than 2 seconds.

Challenges

Similar to the first assignment one of the challenges was figuring out serialization/deserialization. In the end, I used the `cereal` library. However, because `cereal` mostly supports `std` containers and primitive types, I had to use `SDSL`'s own serialization method for saving the suffix array. This required two files two be created: one that stores the suffix array and another file that stores everything else. The files have the same name but different extensions, thus the command line interface is still the same. The prefix table is an unordered_map<string, vector<uint32_t>>. The vector that corresponds to the values of this table stores the start and end positions of the suffixes that share the same `k` length prefix.

Another challenge was creating the prefix table efficiently. My first approach was for each suffix, looking at the suffixes that come before and after it. I would keep going left and right until a mismatched prefix is found. In the end, I would insert the mismatch locations into the table. Later I realized that this was highly inefficient as I didn't need to do this for each suffix. I could simply start from the first suffix in the suffix array and keep going right. When a mismatch is found, log that position as the end position for suffixes that share the current current prefix. This approach yielded much better performance. Finally, I realized that extracting the suffix from the whole genome, and then extracting the prefix from this suffix was too slow. Thus, I did my operations on the original genome by carefully using indices that indicate start position, prefix end position etc. on the original genome. Although not a not an improvement on the algorithmic complexity, this also improved the performance a lot. For further speed up, I tried using a modified binary search for finding the last suffix with the current prefix but it ended up being slower than the current implementation I have.

Plots

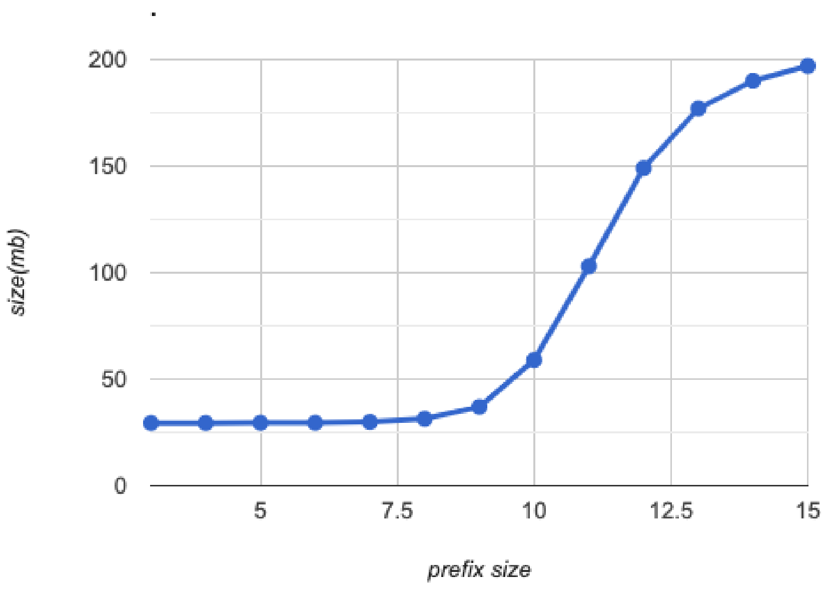

Figure 1: total size (mb) of the both files vs the prefix size

Figure 1 shows the change in the total file size (sizes of .sdsl and .bin files) depending on the prefix table. As expected, as the number of prefixes stored in the table increases, the file resulting file size also increases.

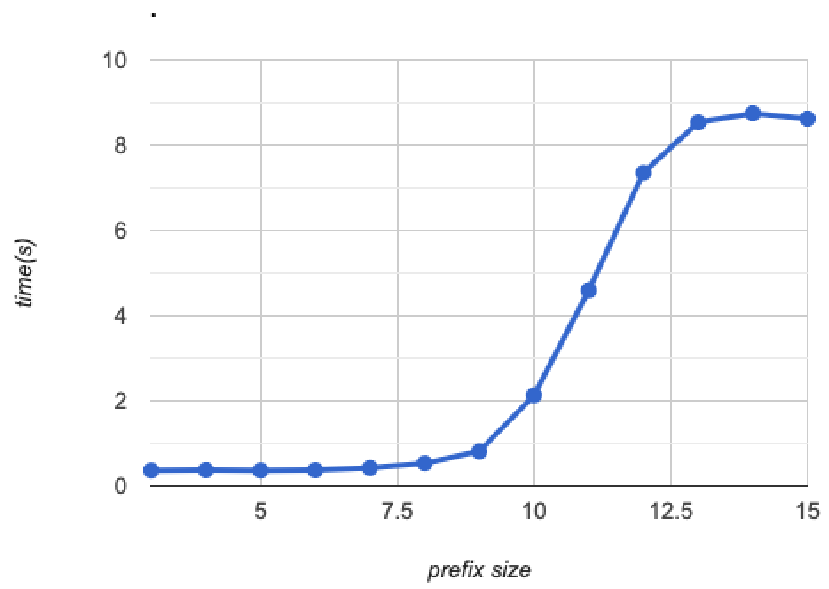

Figure 2: prefix table creation time(s) vs prefix size

Figure 2 shows the time it takes to create a prefix table. By applying the tricks mentioned in the challenges part, I was able to create prefix tables of considerable size very fast. As for the increase around prefix size 8, I am not quite sure. If I were to guess, the data become too large to fit into L1/L2 caches or I started using another program in my computer.

Discussion

As long as the genome size is the same, the suffix array creation time remains the same. Since I haven't tested different genome sizes very much I don't have enough data to generate a plot. However, my tests show the the file size and the suffix array creation time is linear in the size of the genome. Given the ecoli genome with size 4MB requires 30MB storage (approximately 8 times the size of reference genome), with a 32GB RAM we can at most store the suffix array for a genome of size 4GB. Otherwise, we wouldn't be able to read the file back into the memory.

querysa

Description

Compiling the `querysa-driver.cpp` creates an executable named `querysa`. This executable takes the command line parameters described in the assignment description. After reading the binary files created by the `buildsa`, each query in the input file is processed. If the read binary files contains a prefix table, it is used to narrow down the search space in the suffix array.

Challenges

Interestingly, for most queries, the `simple_accel` algorithm takes longer to complete than the `naive` approach and I think I know why. It is possible to compare two strings while also keeping track of the longest common prefix (LCP) of these strings. This can be done in a single function that returns both the comparison result and the LCP. As far as I know, there is no built-in function in `std` library that performs such a thing. So, I wrote my own function. But of course, the built-in string comparison function (which is used in the naive approach) is much faster than my function that compares the strings and computes the LCP. As a result, the `naive` approach outperforms the `simple_accel`. I tested my hypothesis by removing the lcp function call. I could've created a fair playground by using my function in `naive` approach but why would I make something slower? // While running some experiments for the report I realizes that my algorithms are slower when the binary files have prefix tables with greater length. After some inspection I realized that I was passing around an object that contains the prefix tables by copy. After modifying my code to pass the object by reference it started to produce expected results.

The most challenging part was figuring out a way to speed up the prefix table construction.

Plots

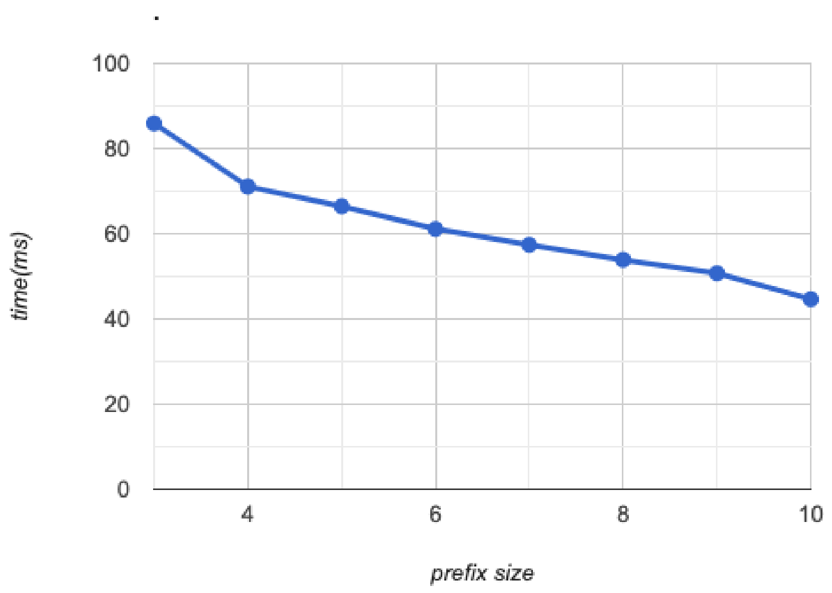

Figure 3: Total execution time (ms) of 10,000 queries vs prefix size for the naive algortithm

Figure 3 Shows the relation between the query execution time and the prefix table size. Longer prefixes narrow down the search space much more aggresively, thus the decrease in the running time is expected. The times shown in the graph are for 10,000 queries from the given example file.

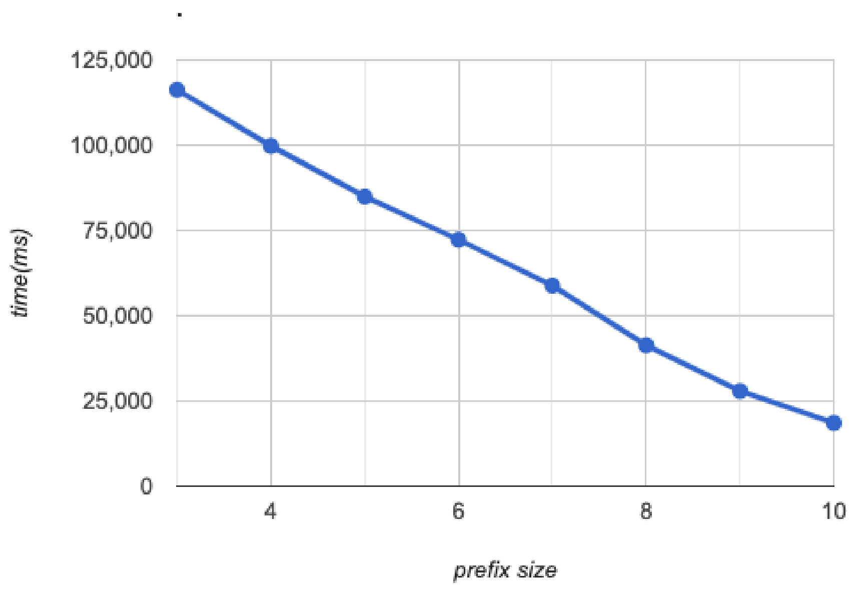

Figure 4: Total execution time (ms) of 10,000 queries vs prefix size simple_acc algorithm

The trend in Figure 4 is similar to the Figure 3 but the decrease in running time is more prominent. The reason for simple_acc algorithm being slower than the naive one is explained in the challenges section. The difference between the two algorithms is similar even when no prefix table is used.

Discussion

Looking at the naive algorithm, it is already very fast. Assuming the simple_acc would be even faster with an efficient lcp implementation, to me, it looks like queries are already being executed fast enough. Thus, if there is any restriction on the memory usage, using a small prefix size or not using a prefix table at all, would be fine. Looking at Figure 3. The biggest improvement is from prefix size 3 to 4. Maybe that could be the sweet spot for the memory vs speed trade-off.

Longer genome causes binary search space to be larger, thus increasing the query time. However, if there is a prefix table, as long as the search space corresponding to query's prefix is the same, increasing the genome size wouldn't affect the query time as our constrained search space stays the same. On the other hand, increasing the length of the query sequence increases the comparison times. In this case a larger prefix size may help by skipping more characters from the beginning of the both strings.
I run some experiments with different genome sizes however I didn't see a significant change in the execution time. This may be either because of the prefix table as I explained in the previous paragraph or it could be because it is bottlenecked by something other than the genome size.

The most challenging part was to trying to understand why the simple_accel was slower than the naive algorithm and finding out why my code was slower with longer prefix lengths.