858D Programming Assignment 2 Report

Yiheng Xu

Notes:

1. Link to github: https://github.com/HenryYihengXu/858D
2. I used SDSL library for constructing suffix array. I used cereal library for serializing/deserializing my data structure. These two libraries are installed at /usr/local on my machine. Make sure you have them installed in right place when running code on your machine. Other than those I didn't refer any existing code.
3. I downloaded 5 reference sequences from NCMI GenBank: Chlamydia (1.1MB), Coxiella (2MB), Caulobacter (4.1MB), Ecoli (4.7MB), Pseudomonas (6.4MB)
4. The program should be compiled with c++20. A makefile is provided. Executables will be in ./bin
5. See README for directory structure

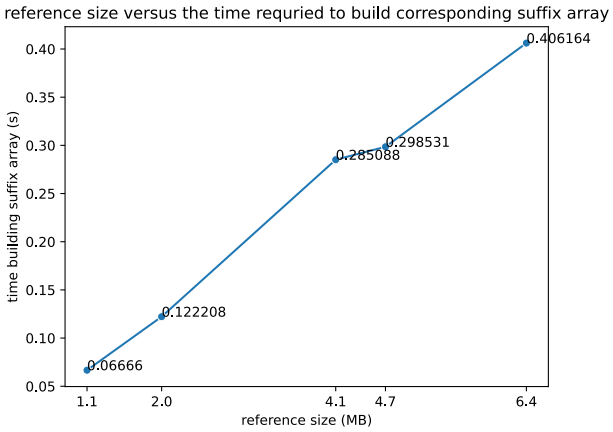
Part (a) buildsa:

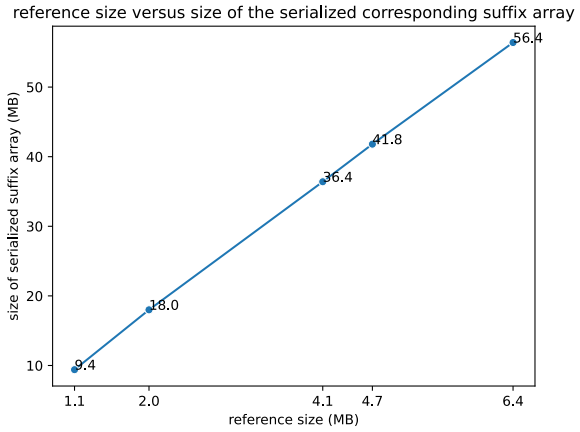
There isn't really a challenge implementing this part. I'll simply mention some details of the implementation. The suffix_array class contains a string that holds the reference sequence, an int_vector<64> that holds the suffix array, a std::unordered_map (which is a hash map) that holds the prefix table if requested. The suffix array is constructed by sdsl::algorithm::calculate_sa.

For constructing the prefix table, I do a linear scan of the suffix array. At any point, I keep the common prefix up to now and the start index of this prefix. I compare it with the suffix at this point. If it matches, I keep going. If not, I add this prefix to the table, with the start index and current index – 1 as the end index. Then I keep the new prefix. Since the suffix array is sorted, this way guarantees to find the complete interval for all possible prefixes.

For serialization, I use cereal library, which is able to serialize string and unordered_map. The SDSL int_vector also has a serialization method. So I simply open a ofstream and serialize my data structures one by one. In this way everything is serialized to the same file. I do the same thing for deserialization.

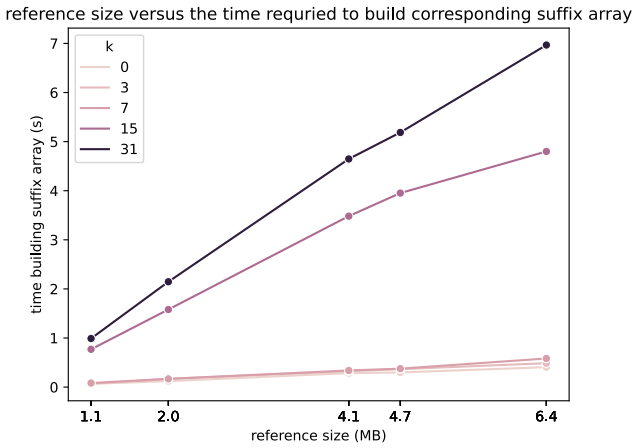
To evaluate my implementation, I downloaded 5 reference sequence with different size from NCMI GenBank: Chlamydia (1.1MB), Coxiella (2MB), Caulobacter (4.1MB), Ecoli (4.7MB), Pseudomonas (6.4MB). I'm also testing it with 5 different prefix table sizes: 0 (meaning no prefix table), 3, 7, 15, 31. So in total, I ran 5x5 = 25 experiments. For each experiment, I build the suffix array and save it to a binary file. I time the building of it and note down the size of each saved file. Here are the results:

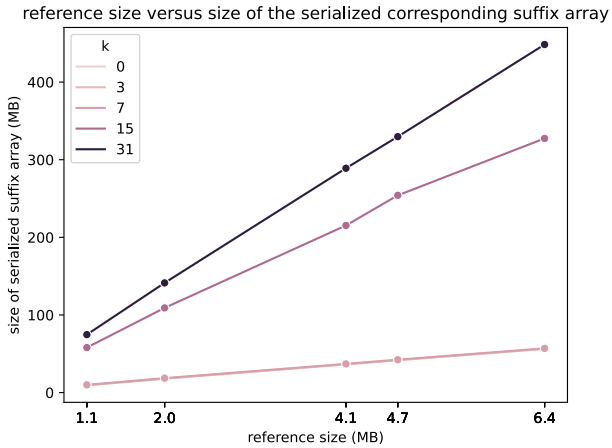




These two plots show the time and size of building and saving suffix arrays with sequences of different sizes without the prefix table. The left one shows the time needed is linear to the reference size, which is expected because the time complexity of the best construction algorithm is O(logn). And the time is pretty fast: less than half a second for a large sequence.

The right one shows the size of serialized suffix array. I'm using 64 bits unsigned int for the suffix array, and 1 byte char encoding for the reference sequence. Take the Ecoli (~4.7MB) as an example, there will be ~4.7M elements. The whole structure will be ~4.7M * (8 + 1) = ~42.3, which is very close the actual size 41.8M. It's also as expected that the size grows linearly to the reference size.





Now let's see how time and size look like with prefix table. These two plots show the time and size of building and saving suffix array with prefix tables with different k on those reference sequences. It is trivial that larger k needs more time and space. But it is interesting that smaller k's have almost the same time and space, and there is a big jump from k=7 to k=15, and then the differences between larger k's are not that significant. This is actually expected, because

the number of entries in prefix table doesn't grow linearly to k. When k is small, like 5, all possible permutations of 5-mer may all appear in the sequence. So before some certain point the size grows exponentially. That's why we see the big jump. But after some point, when k is large enough, only part of permutations of k-mer can appear in the sequence. Even k gets even larger, the number of k-mers existing in the sequence won't change a lot. That's why we see no big differences between larger k's.

And for a certain k, the time and space grow linearly to the reference size. This is expected since what I did to construct prefix table is a linear scan of the suffix array. The time complexity is O(nk). The space complexity is O(k * num k-mers). It's reasonable that the number of k-mers of a sequence grows linearly to the reference size for large k's. So the space should also be linear to the reference size.

Finally, given the scaling above, if I'm not building the prefix table, each element requires 8+1 = 9 bytes. So on a 32GB RAM the longest reference I can hold has around 32GB/9 = 3.56 billion elements. If I build the prefix table, I will choose larger k (not too large) because it gives me more marginal benefits. So say I choose k=31, the line in the plot is something like y = 60x + 50 in MB. So plug in 32GB I get something like x = (32*1024 – 50) / 60 = ~545 M. So I the longest reference I can hold in this setting has around 545 million elements.

Part (b) querysa:

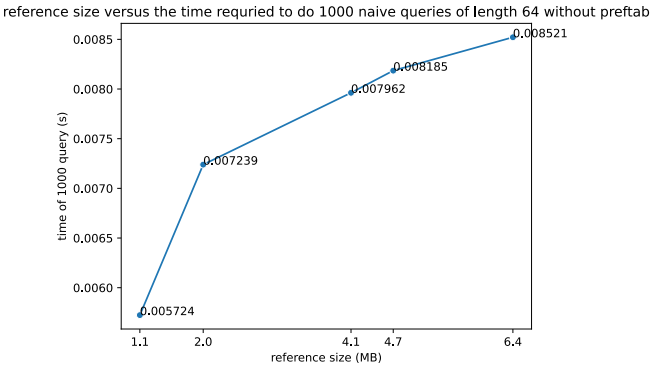
Parts that are bit challenging perhaps are finding the interval boundary and the simple acceleration. To find the left and right bound of the interval I did two similar binary searches. For the left bound search, I compare the center string up to the pattern length. If it's larger or equal to the pattern, I move right to center. Otherwise I move left to center. In this way I'm sure to find the first occurrence of the pattern (the left bound) because I'm moving towards the left even the center is already a occurrence. For the right bound, I'm doing the same thing, but only move right to center when the center string up to the pattern length is strictly larger than the pattern.

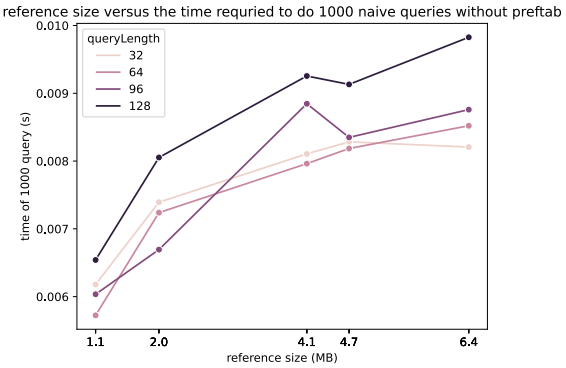
For the simple acceleration, I spent some time looking for c++ standard string comparison functions that can both tell me the relation and the lcp. I wasn't able to find one so I implemented a string comparison myself. The idea is to compare char by char starting from a specified position and keep track of the lcp. At the end I return an int vector containing both the relation and lcp. It is straightforward but it's not giving me as good performance as I expected for small k's. I'll discuss this later in the evaluation.

One thing I want to mention about the correctness. I have validated my querysa with the provided queries by comparing my results to the provided results. They are the same, but just FYI the provided results are separated by "^I" instead of \tab. And it is sorted in the lexicographic order of query names, whereas mine is in the input query order. In case you are using diff to check correctness, please consider these things.

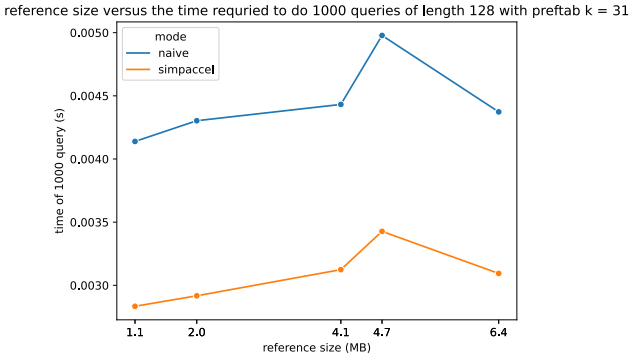
Now to evaluate querying, there are several varaibles: the reference length n, query length m, prefix table parameter k, and the query mode. For references and k, I use the same ones in part (a), so 5 reference lengths and 5 k's. I then created query sets of 4 different lengths: 32, 64, 96, 128. Therefore, in total I ran 5x5x4x2 = 200 experiments.

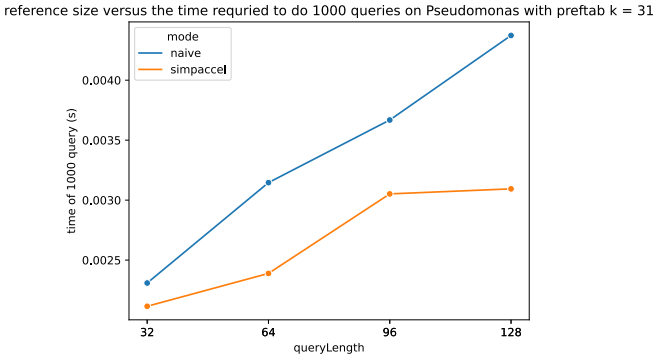
I wasn't able find resources of query sets for those references with specified size, so I created them by myself. The way I created them is as follows: for a reference sequence, I randomly pick 1000 position, and then take m chars starting there as a query pattern. In this way the position of the pattern is random, and repetitions of it is completely random as well. Since I noticed the provided query file has almost all patterns actually appear in the reference, I decided it's okay to not include patterns that don't exist in the reference. Ignoring this tiny potion shouldn't affect the time. I then ran each query set for each reference, which is 1000 queries, and time them all together. Here are the results:

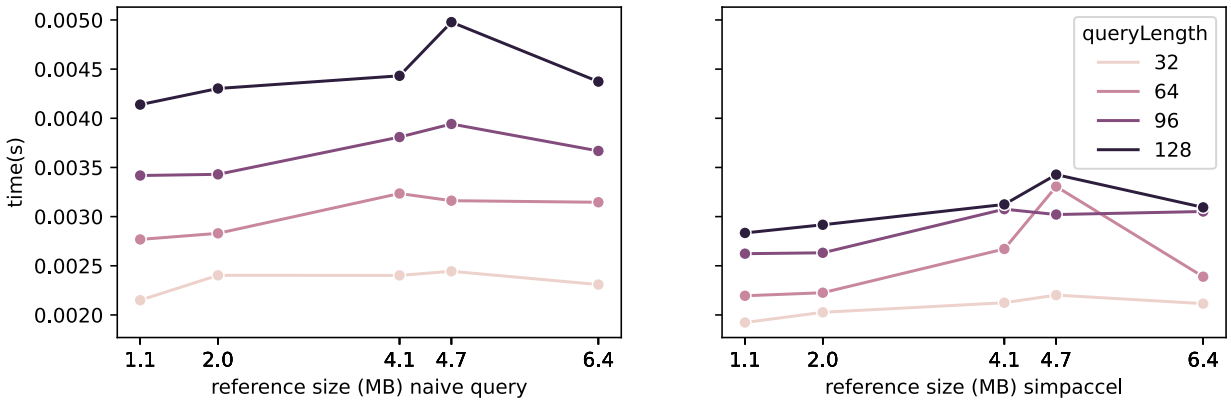




Starting from the simplest setup, let's look at the naïve query time of different reference sizes without prefix table. Above, the left plot shows the time for query length 64, and the right one shows time for all query length. It's easy to see the query time is logarithmic to the reference size, and longer queries take slightly longer. This is as expected as the naïve query time complexity is O(mlogn).

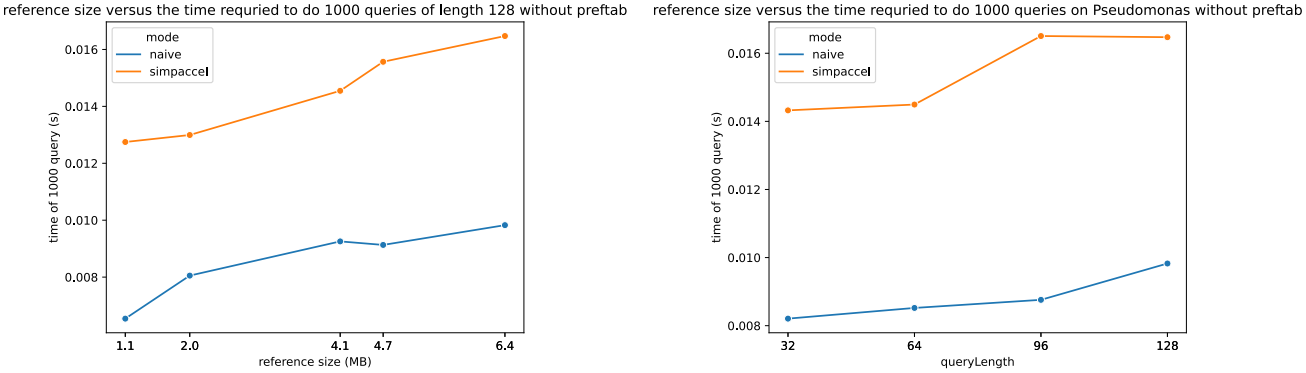


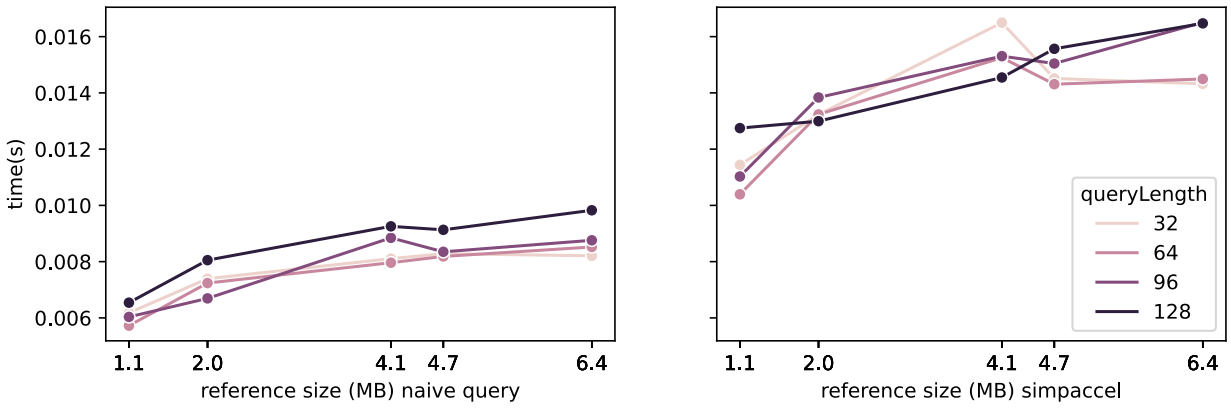




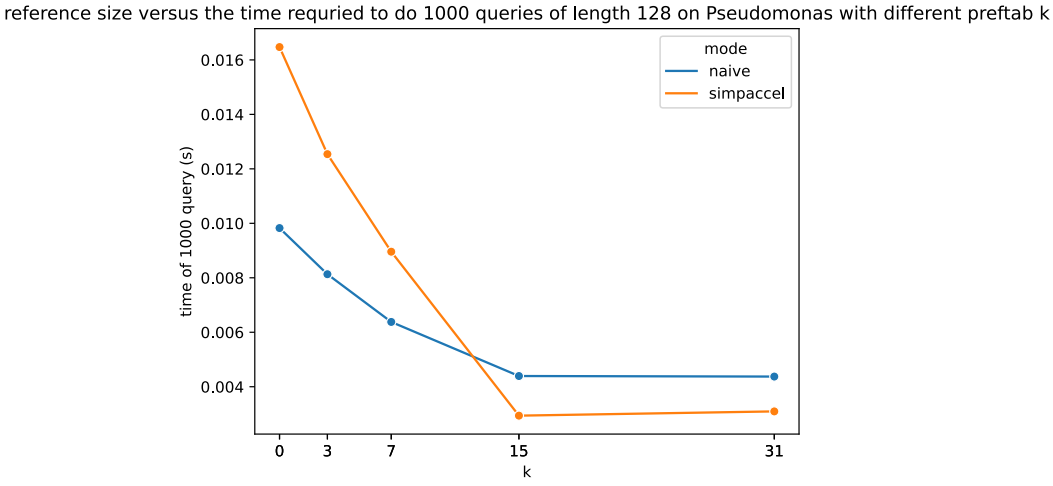
Now let's look at how naïve query compared with simple acceleration, and how prefix table affects query time. The plots above shows the two modes of query time on fixed query length =128 but different reference size (upper-left one) and on fixed reference size (Pseudomonas 6.4MB) but different query length (upper-right one). The bottom plots show a more comprehensive comparison. The bottom left one is naïve query and the bottom right one is simple acceleration. All plots have k = 31.

There are several interesting insights. Let's first focus on comparing naïve query and simple acceleration. With fixed k = 31, we can see that simple acceleration gives better performance for all reference sizes and query lengths. This is expected because simple acceleration avoids repetitively comparing common prefixes. When k is large, the query starting from a range that has long common prefixes as the pattern, so the speedup is more significant.



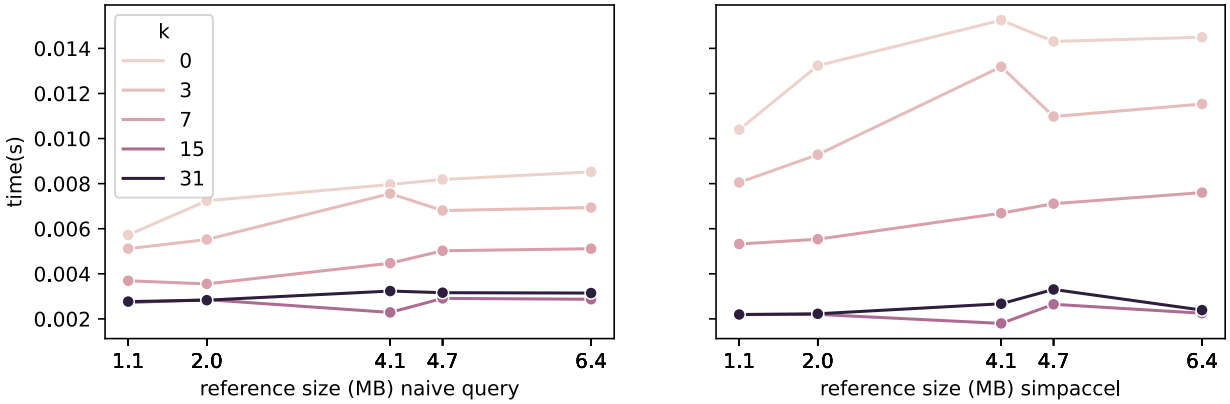


However, as mentioned previously, the simple acceleration didn't perform as good as I expected for smaller k's. The plots above are exactly the same thing as previous ones, but with k=0 (no prefix table). Although it's still true that longer reference and longer query need more time, the performance of naïve query and simple acceleration is the other way around. Naïve query now performs better than simple acceleration for all reference sizes and query lengths. To be more precise, the plot below shows the performance of naïve query and simpaccel w.r.t. different k's. Approximately we can see for k < 11, naïve query performs better than simpaccel, while for k > 11, simpaccel performs better than naïve query. One possible explanation I can think about is my string comparison implementation has overheads. Perhaps it adds overheads at returning an int vector, constructing it, extracting the relation and lcp from it. And compared with simple string comparison, these overheads are not negligible. Therefore, the benefits only show up with large k, when the prefix is long enough at very beginning of the search to compensate the overheads. Or, I might just need even longer references and queries to see the benefits for small k's.



Finally, let's generally talk about the impact of k on either query mode. Another interesting insight from previous plots is that for small k (k=0) the query time is logarithmic to reference

size, but for large k (k=31) the query time is almost constant. This might be because when k gets large, the prefix interval size becomes more and more irrelevant to the reference size, as large k-mer gets rare in any references (maybe not true for even larger reference). The plots below show a more clear relationship between query time and k. We easily see for both query modes, larger k gives notably better performance. This is expected as larger k gives more precise interval. The number of binary searches reduces. At the same time cache performance improves with small interval. Furthermore, this is especially true for simpaccel. We can see k=31 gives almost 10x speedup than k=0. As argued before, large k gives long common prefix from very beginning, making benefits of simpaccel more significant.



In conclusion, given the memory requirements of each type of index, I personally think it makes sense to have suffix array with larger k and use simpaccel query. Although larger k has significant larger memory requirement, the time is also significantly shorter. Moreover, it doesn't make sense to have middle size k because the space grows exponentially before some certain k, but grows little for k's larger than that threshold. Small k has bad query time, so I would use larger k to trade faster query time with simpaccel.