1 Part 1

I started my implementation using C++ with the SDSL library for suffix array construction. I ended up abandoning this work and switching to Python with the pydivsufsort package due to issues understanding how to use cereal for serialization as well a general concern regarding the time I had to work on the project. In hind site, while Python was certainly more comfortable for me to work in, I don't think it saved me much time because most of my problems were in part 2.

For evaluating the performance of my code I built suffix arrays on subsequences of the ecoli reference you sent the class of size 10%, 30%, 50%, 70%, 90%, and 100% the size of the original reference. For each of these I also created prefix tables of size 2, 4, 8, 10, and 12. My algorithm for creating the prefix table is just a linear scan through the suffix array, keeping track of whether the k-length prefix of the current index of the suffix array differs from the prior one. In principle the size of the prefix array explodes exponentially in the size of k, but in practice this behavior will depend on the structure of the reference. Because I did not design the algorithm for building the suffix array it is not totally clear what its asymptotic growth in run-time should be, but in practice it was quite fast.

I archived the data structures using Python's builtin pickle module as a 4-tuple of (sequence, suffix array, prefix table, size of prefix).

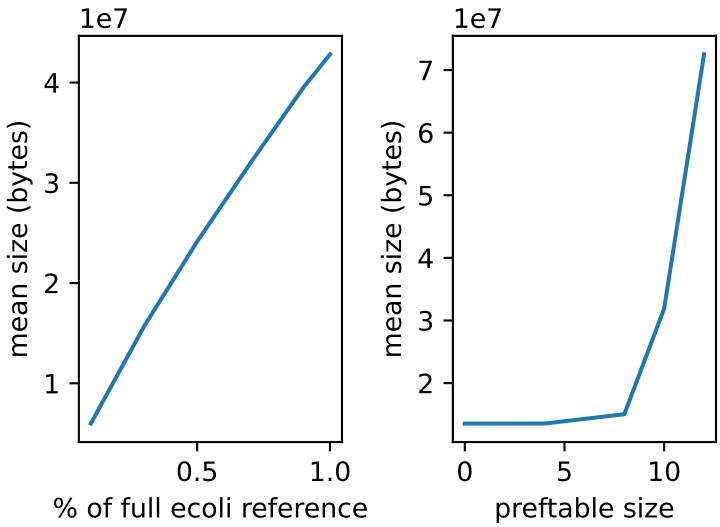

Figure 1: mean size of the archived data. Left shows that the size grows linearly in the size of the reference (which is to be expected) and grows exponentially w.r.t the length of the prefix as expected.

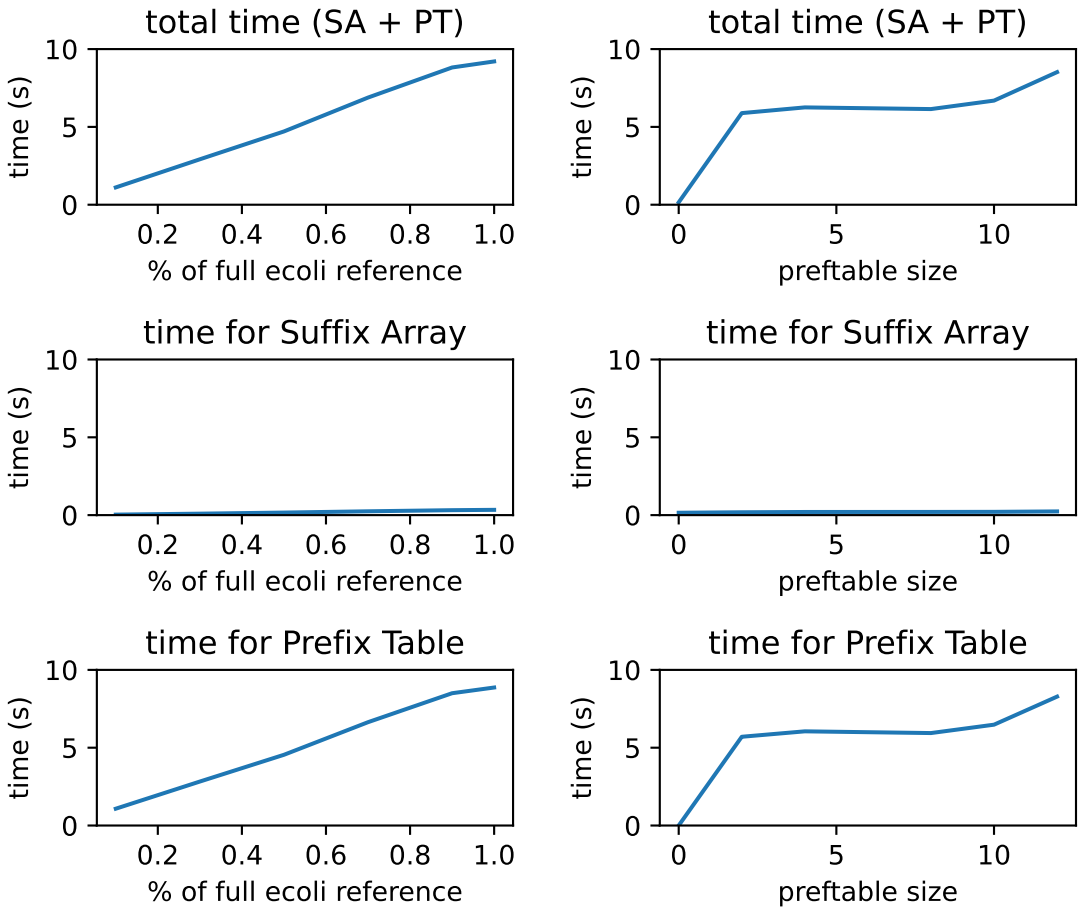

Figure 2: The time to build the suffix array and prefix table. Time to build the suffix array is negligible compared to the prefix table, and does not depend on the prefix size as we expect.

2 Part 2

The hardest part of part 2 for me was that I was mistaken about what the simple accelerant algorithm was for a long time, and was trying to do the super accelerant version instead. Whoops! Once I got that sorted out things seemed to fall into place smoothly. To test my code I generated 10 replicates each of queries of sizes $10, 20, \ldots 100$. I generated these sequences uniformly and randomly on the dictionary of $\{A, C, T, G\}$, which I believe will make queries unlikely to appear in the reference as the size of the query grows (since we know that genomes do not follow a uniform distribution).

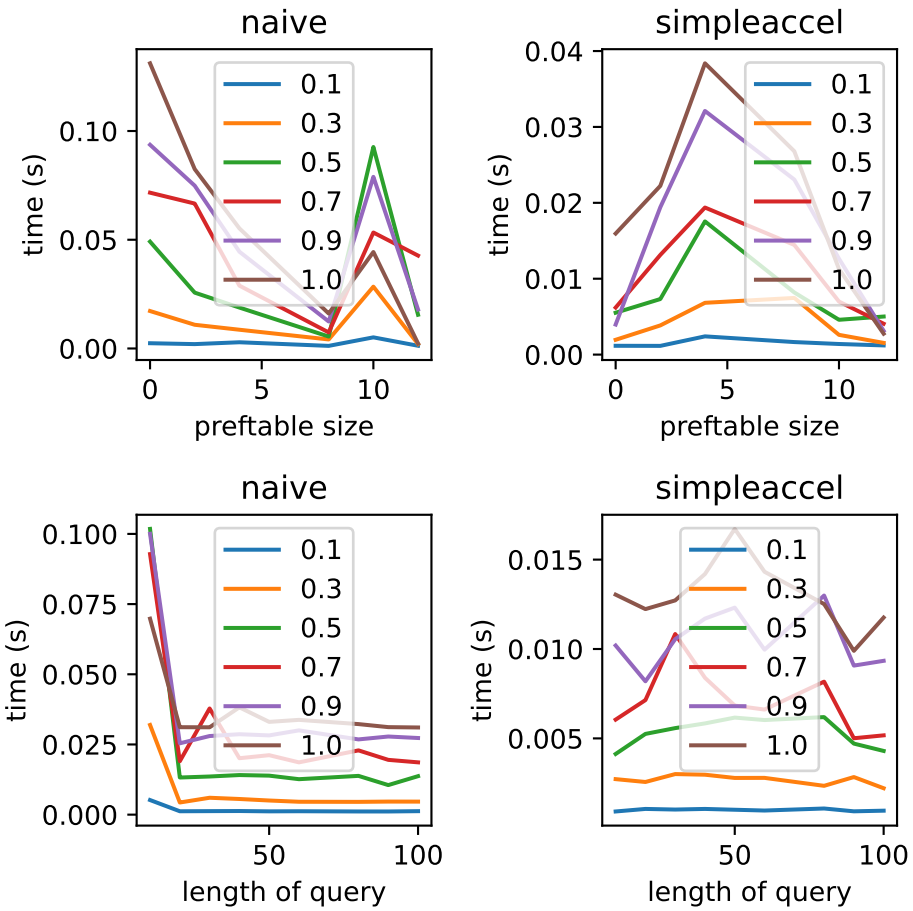

Figure 3: The time to query the suffix array across all sizes of reference w.r.t the size of the prefix table and the length of the query.

My timing results here are clearly quite noisy, and in hind sight I think I would have gotten better results if I used more replicates. What the plots seem to show is that the length of the query has little effect on the overall runtime, and the length of the prefix seems to

in general reduce the the runtime of the algorithm (though for simple accel it mysteriously has deletorious effects for small non-zero prefix). The main determinant of runtime here is apparently the size of the reference, which makes sense. The simple accel is faster by a constant factor in practice than the naive algorithm though!

Given the results here I would probably abstain from using the prefix table unless I had a ton of memory spare, since its growth in size is exponential and it does not provide exponential time reduction. It seems like there is very little downside to using the simple accelerant over the naive algorithm.