Suffix Array Benchmarking and Analysis

Author: Nikhil Pateel

`buildsa`

My implementaiton of `buildsa` uses `cereal` and `sdsl`'s `csa_wt` class to build the suffix array table. The way that the prefix table was built was through using an

$$O(kn)$$

algorithm that looped through each entry in the suffix array and got the first and last indices of each prefix of length k.

Below, we'll perform benchmarking based on the time it takes in seconds to build the suffix array with no preftable and a preftable of $k = 8$. We use this number because it is sufficiently useful for prefix tables and in the middle of our prefix table benchmarking.

We find that, overall, there is a very heavy cost of computing a prefix table (2s for a ~100k NT sequence and approx 20s for E.Coli), but the overall value of the k does not matter too much in the index build time. (Most of the slowdown happens in actually building the prefix table).

Building the suffix array takes less than a second for any NTs less than 100k, but then started to take more and more time. The Size increased in proportion to the sequence length.

Here's a table of the results for fixed k (preftab = 1) (but they're all similar except for size)

Nucleotide Size	SA build time	Size
700	0.047s	4 kB
7k	0.040s	13 kB
70k	0.040s	156 kB
700k	0.21s	1.4 MB
7M	2.1s	13.7 MB

When k changes, the prefix table time increases slightly, but the size of the serialization greatly increases. Here are results at 7M nucleotides

k	Prefix Table Build Time	Size
-1	0	13.7MB
1	55.85s	13.7MB
2	55.55s	13.7MB
4	55.55s	13.7MB
8	56.61s	23.6MB
16	61.90s	220MB
32	62.90s	360MB

So we can see that for lower values of k (less than 8), the size doesn't get prohibitive, but more than that can cause some serious problems with compression.

With this scaling, and no prefix table, I expect that we'd be able to go up to a sequence length of 16 GNTs (16 billion) seeing that the overall overhead of the SA is 2 times the number of nucleotides in bytes (there probably are some issues with compression causing such bloat).

In [ ]:
```
import os
import subprocess

sizes = [10, 100, 1000, 10000]
ks = [None, 1, 2, 4, 8, 16, 32]
for size in sizes:
    for k in ks:
        args = ['../bin/buildsa']
        if k is not None:
            args.append('--preftab')
            args.append(str(k))
        ref_file = '../' + os.path.join('data', 'samples', 'ecoli-' + str(size) + '.fa'
        out_file = '../' + os.path.join('cache', 'ecoli-' + str(size) + '-' + str(k or
        args.append(ref_file)
        args.append(out_file)
        args.append('-b')
        args.append('benchmarking.csv')
        print(args)
        subprocess.run(args)
```

querysa

I found `querysa` much harder to implement in general. I'd say that I spent at least twice as long on this project. I found it especially hard to get the prefix table working like I wanted to, and ran into a lot of issues specifically with random edge cases in the prefix table. A very big source of annoyance was that the human genome dataset has a lot of features and noise (lowercase vs uppercase). While my code works for the reference genome and other genomes that I've written myself, it can't seem to work for the human genome overall. There might be some latent bugs, or input issues that are causing this problem.

Nevertheless, here are the results I was able to scrap together.

When we control for everything besides sequence length, we get the following: (naive k=4, 64 long queries)

Nucleotide Size	Average Query time (ns)
700	2413
7k	17172
70k	62661
700k	108024

We see that the runtime is approximately logn with respect to sequence length.

When query size changes... (with 70k NTs)

Query Size	Average Query time (ns)
32	61863
64	62661

Query Size	Average Query time (ns)
128	63196.2
256	62021.9

That is to say that while the expected runtime to increase linearly, we see that the runtimes are largely constant. It's possible that there are large constant factors making this scale better than the expected time complexity.

Next, we visit the impact of having a prefix table (assuming 70k NTs, 64 long queries) and query method | k| Average Query time (ns) (naive) | Average Query time (ns) (simpaccel) | | -1| 96173.9| 122531| | 1 | 40020.9|114547| | 2 | 31798.9|118347| | 4 | 17172.4 |103008| | 8 | 645.452 |99520.5| | 16 | 118.22|126894| | 32 | 120.99|126896|

We see that there are diminishing returns as k gets larger and larger. It seems like $k = 4$ is a "sweet spot". Furthermore, it looks like there's a performance penalty with `simpaccel`, this likely has to do with an intentional slowdown in the lcp compare method (for some reason using `start = smallest` doesn't work in line 169 of `querysa.cpp`, smallest becomes way too big for no reason, but only on lcp with prefix tables).

All in all, I believe that having prefix tables are useful, but might be expensive in pre-computation time. It really depends on if an index will be used a significant amount or not. Regardless, since k has little effect on the time complexity, I would suggest a value of $k = 4$ or 8 to balance the space complexity with the performance gain on query times.

In [ ]:

```python
# generate queries
import random
queries = [32, 64, 128, 256]
for query in queries:
    queryfile = '../' + os.path.join('data', 'queries', str(query)+'.fa')
    with open(queryfile, 'w') as f:
        for i in range(1000):
            f.write('> ' + str(i) + '\n')
            f.write(''.join(random.choices('ACTG', k=query)) + '\n')
```

In [ ]:

```python
# generate results
algos = ['naive', 'simpaccel']
sizes = [10, 100, 1000, 10000]
ks = [None,1, 2,  4, 8, 16, 32]
for size in sizes:
    for k in ks:
        index = '../' + os.path.join('cache', 'human-' + str(size) + '-' + str(k or 'no
        for algo in algos:
            for query in queries:
                args = ['../bin/querysa']
                args.append(index)
                queryfile = '../' + os.path.join('data', 'queries', str(query) + '.fa')
                args.append(queryfile)
                args.append(algo)
                args.append('output')
                args.append('-b')
                args.append('querybenchmark.csv')
                print(args)
                subprocess.run(args)
```

In [ ]:
```

```
In [ ]:
```

```