CMSC858D

Algorithms, Data Structures and Inference for High-Throughput Genomics

Homework 1

Ataberk Donmez
Student ID: 118208473

Task 1

Description

Other two tasks build on the `RankSupport` class, so I started the project from this one and the code can be found in `RankSupport.hpp` and `RankSupport.cpp`. In addition to the functions from the project spec, I implemented my own functions for converting decimal to bit vector and vice versa. I use the `SDSL`'s bit_vector class in my implementation. Based on the discussion on Piazza I set my superblock size to be the square of block size, which reduced the difficulty of implementing R_s and R_b tremendously. Finally, I implemented R_p too, instead of using popcount.

Challenges

I spent the most time figuring out how to make popcount work and how to do efficient save/load. In the end, I implemented R_p rather than using popcount and ended up storing only the bit vector and reconstructed the data structures after loading. Storing superblocks and blocks as a vector of bit vectors (instead of just a bit vector) makes the serialization non-trivial. I tried to use C++'s Boost library but couldn't make it work.

Plots

,

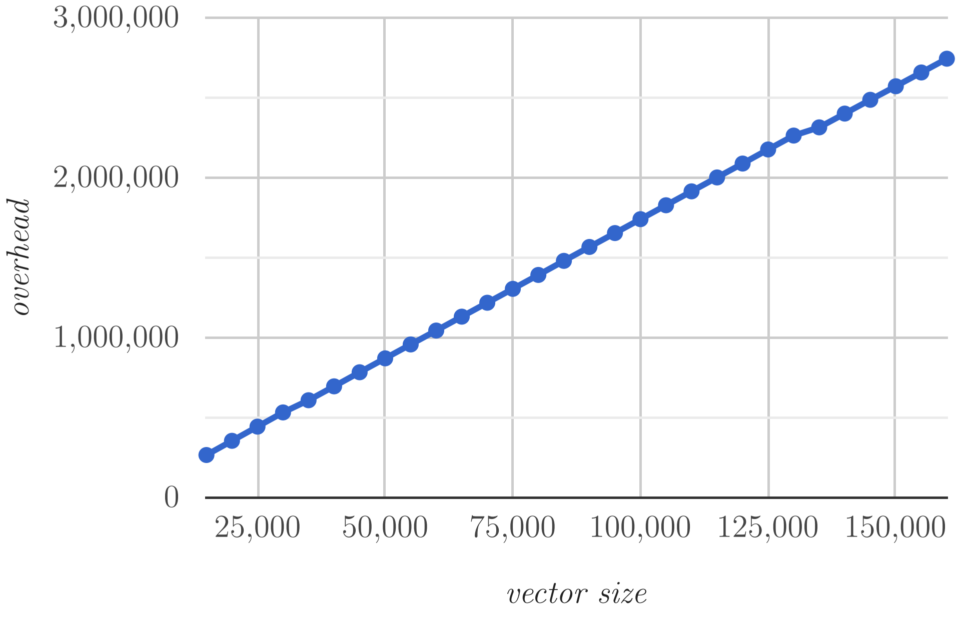

Figure 1: Overhead vs vector size of the bit vector.

Figure 1 shows an almost perfectly linear relationship between the size of the vector and the overhead of the corresponding `RankSupport` object for 30 vectors with varying sizes. Considering the the space requirement for this data structure is $O(n)$. The overhead indeed match the expected theoretical bound. For finding the size of each bit vector I used SDSL's `size_in_mega_bytes` function for each bit vector and summed the results.

,

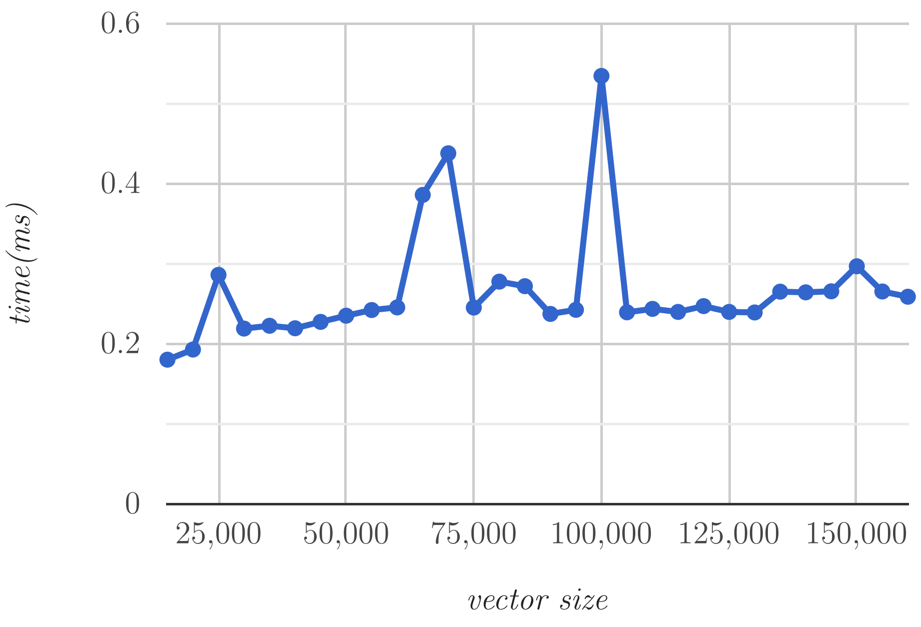

Figure 2: `rank1` time(ms) vs vector size of the bit vector.

Figure 2 shows the relationship between the vector size and the total time for 100 rank1 queries for 30 vectors with varying sizes. Despite the outliers, the general trend is almost constant time as there doesn't seem to be an increase in total time correlated with the vector size (again, excluding the outliers). The few data points with high variance may be due to the background applications running at the time of the experiments for that vector.

Task 2

Description

The implementation of this task can be found in `SelectSupport.hpp` and `SelectSupport.cpp` files. After implementing the first task, this one was straightforward. I use a slightly modified version of binary search to get the result of the select1 queries. As this class doesn't store any additional data structures, overhead is the equal to its member `RankSupport` object.

Challenges

Compared to Task 1, this task was simple. Most challenging part was making sure that I was returning the index of the 1 with the desired rank and not the subsequent 0s that may follow, which also have the same rank.

Plots

,

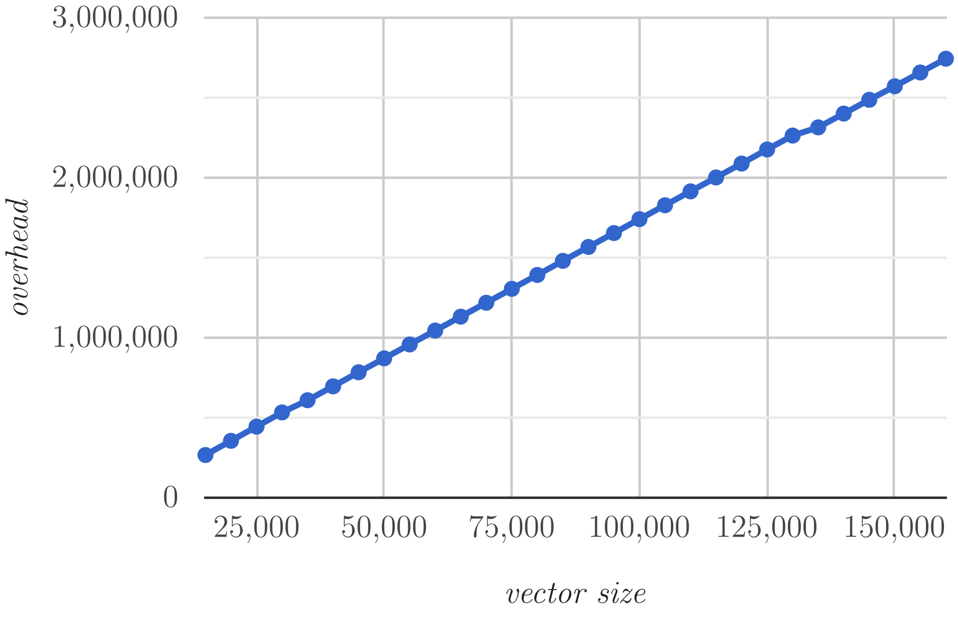

Figure 3: Overhead vs vector size of the bit vector.

Figure 3 Shows the relation between the vector size and the overhead for 30 vectors with varying sizes. As there are no additional data structures for `SelectSupport` other than the `RankSupport` object it stores, this figure is identical to Figure 1

,

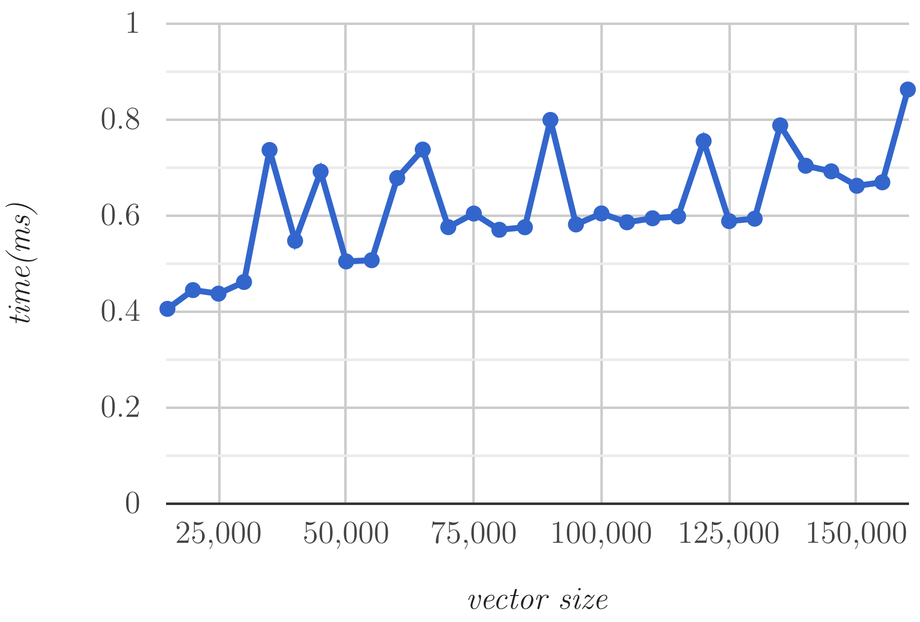

Figure 4: `select1` time(ms) vs vector size of the bit vector.

Figure 4 Shows the relation between the vector size and total time for 100 select1 queries for 30 vectors with varying sizes. Unlike Figure 1, we can observe an increase in total time as the vector size increases. Considering the run time complexity for the select1 function I am implementing is $O(logn)$, this matches the theoretical bounds.

Task 3

Description

The final task makes use of the previous two tasks I have implemented and it can be found in files `SparseArray.hpp` and `SparseArray.cpp`. It contains both a `RankSupport` and `SelectSupport` object as member variables. However, in hindsight, I see that there was no need for a `SelectSupport` object as none of the required functions needs to compute the select1.

Challenges

Thanks to the examples and great explanations on the homework page it was straightforward to implement most of the functions. However I couldn't figure out how to serialize both the sparse bit vector and the string vector that contains the actual string values to the same file and than be able to load them back.

Plots

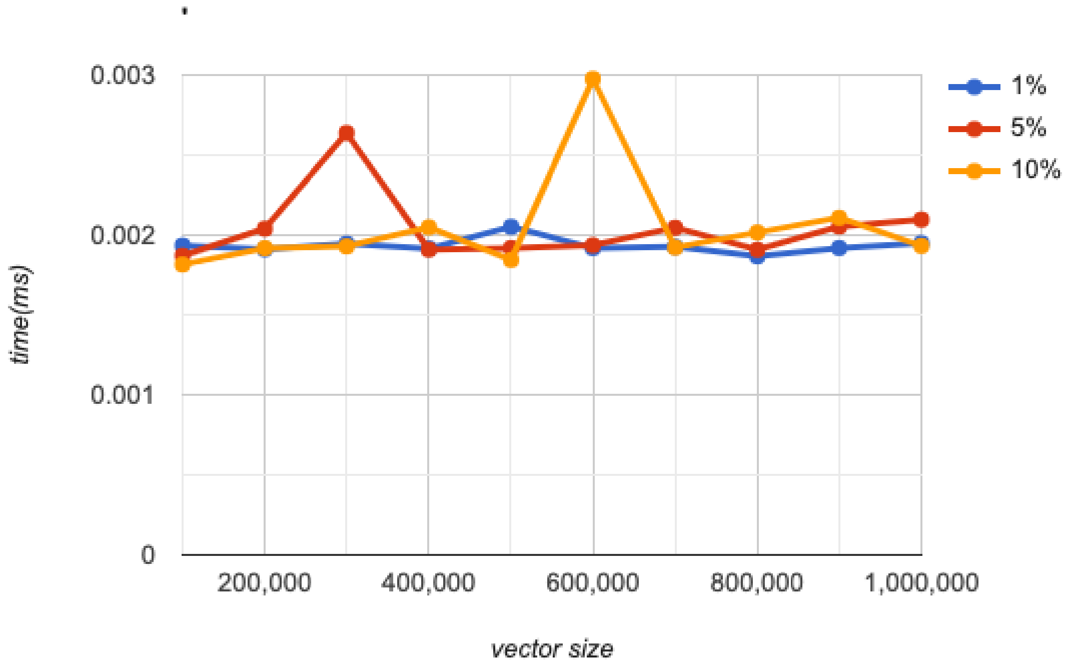

Figure 5: `get_at_rank` time(ms) vs vector size across different sparsity values

Figure 5 shows the relation between total time (ms) for 100 `get_at_rank` queries and vector size across 3 different sparsity values Looking at the figure, it is hard to distinguish between the three lines. Since this function is simply a vector access operation I wouldn't expect the time to change based on the vector size or the sparsity.This figure shows that the results support my expectations.

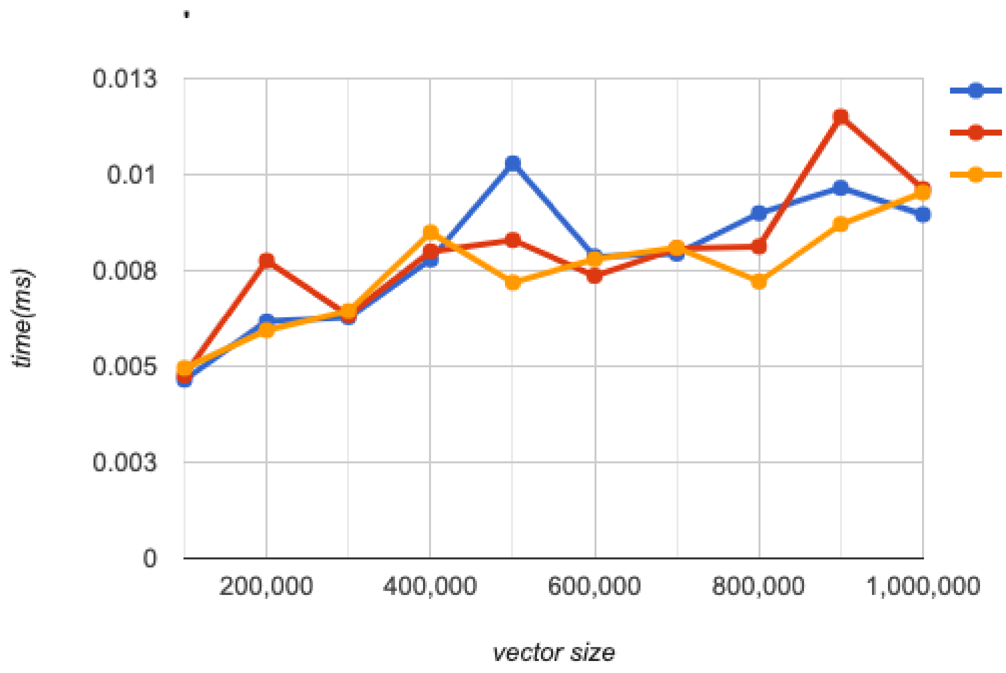

Figure 6: `get_at_index` time(ms) vs vector size across different sparsity values

The plot is not very helpful in finding a relationship between the time and the sparsity. I would expect more sparse vectors to be queried faster. This is because we would hit 0 bits more often with these vectors and as a result we do less processing. I guess this does not make much of a difference and because the rank1 function is constant time, all `get_index_at` calls take similar time for all sparsity values.

The size of the vectors mostly affect the append method as I basically recreate the `RankSupport` object every time an item is appended.

If we were to explicitly store empty std::string instances, each empty element would take 24 bytes on my machine. Thus, for each such element we save up to $24 * 8 - 1 = 191$ bits. Instead of std::string, if we were to store char*, we would save $8 * 8 - 1$ bits per empty string. The more sparse our vectors are the more space we save. This is because we don't save any space from the non-empty strings (we have to store the strings themselves anyways) and our savings come from sparsity.