Homework Assignment 2 Report

Yunheng Han

March 2023

Task 1, 2

I implemented the bitvector data structure in C++, using Jacobson's rank and logarithmic select based on binary search. For Jacobson's rank, the size of the chunks are fixed, which is determined by the theoretical values corresponding to the bitvector of 2^{64} bits. In this case, the size of a chunk is 2^{12} bits, the size of a subchunk is 2^5 bits. The bitvector itself is stored in an array of 32-bit integers, whose size is the same as a subchunk, so that the final rank in the subchunk can be computed using population count conveniently.

I think the most difficult part of the implementation is manipulation of bit values, and choice of the size of the chunks and subchunks. Iniitally, I strictly followed the theoretical bound, i.e., $\log^2 s$ for chunks and $\log s/2$ for subchunks. However, it turned out to be inconvenient because the sizes did not fit into machine types.

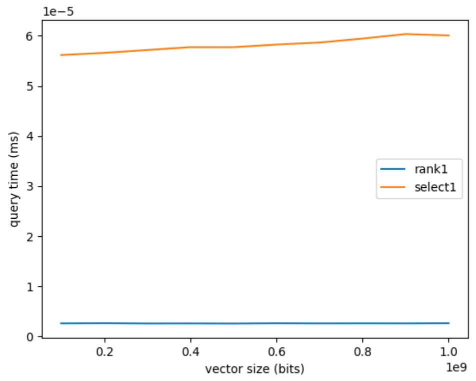

Figure 1: Bitvector query time

The query time is shown in Figure 1, including `rank1` and `select1`. The rank query takes constant time while the select query is logarithmic. As a result, `select1` is more time-consuming than `rank1` (see Figure 1). In terms of the space cost, we plot the size of the index, as well as the size of the bitvector, in Figure 2. We observe that the size of the index does not exceed that of the original bitvector. As shown in the figure, it takes approximatedly half of the size of the bitvector to store the index. It does not match the theoretical bounds because here the size of the index is a linear function of the size of the input bitvector due to constant choise of chunk/subchunk sizes.

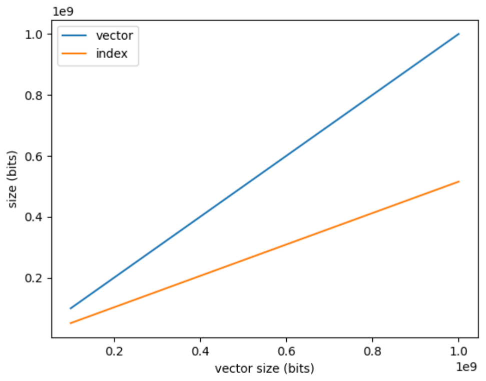

Figure 2: Bitvector index size

Task 3

In this part, I store the strings and their positions in a list besides the bitvector. All functions are based on `rank1` and `select1` in the previous parts.

The implementation of the functions in this part is not complicated given the bitvector data structure, while I spent more time implementing the functions of saving and loading the index because the strings may have different lengths.

As shown in Figure 3, `get_index_of` is most time-consuming because it takes $O(\log n)$ time, while other queries are more efficient and have similar time cost. In Figure 4, we observe that the space cost is reduced a lot by using the sparse representation. The naive approach which stores empty strings needs even more space as the length of bitvector increases.

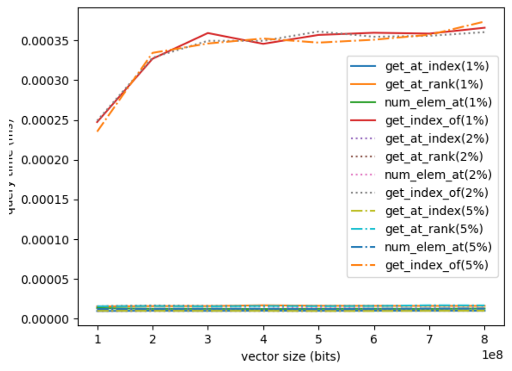

Figure 3: Sparse vector query time

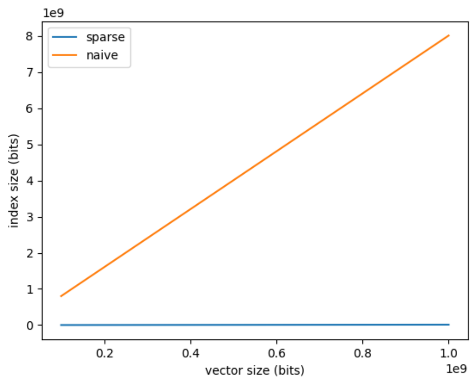

Figure 4: Sparse vector index size