CMSC 701 HW2 Writeup

Part 1 - Rank

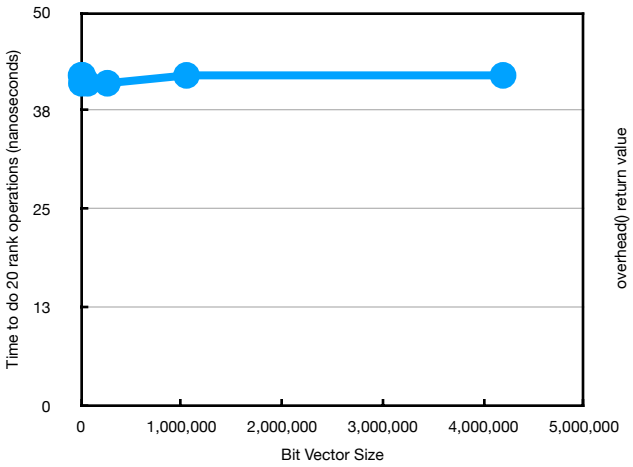
Description

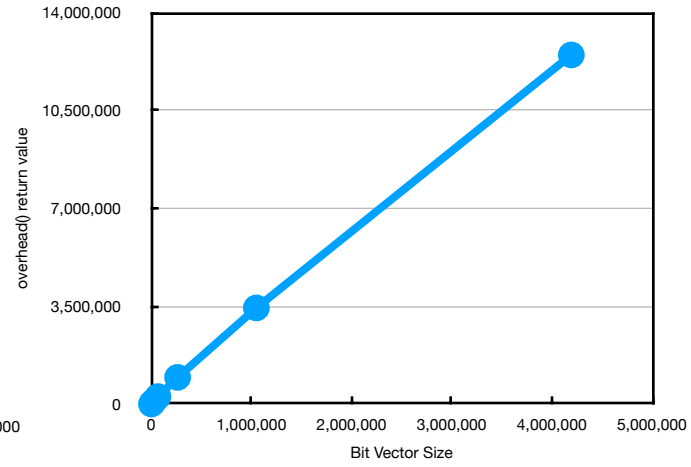
In my implementation, I used sdsl::bit_vector to create a bit_vector as well as two int vectors to store the cumulative ranks of chunks and all the relative cumulative ranks of sub-chunks respectively. In the constructor, I computed the size and length of chunks and sub-chunks using the size of the bit vector, and then filled in the cumulative ranks and relative cumulative ranks. When rank1() is called, it will first find what chunk it is in, look up cumulative rank, then find the relative cumulative rank in the sub-chunk. Finally, it uses popcount() to get the rank within the sub-chunk, and then added up the cumulative rank and relative cumulative rank and return the value.

Difficulties

When implementing this part, I had difficulty figuring out how to build and run it so I could debug. I used SDSL, which is an external library, and had a lot of issues with linking it to my project during building. I tried several methods, including moving the "include/sdsl" folder to my project folder root directory, using CMake, using VS Code plug-ins, but I did not have success in either of them. Then I finally figured out that I can use g++ in command line following SDSL's official documentation. Another issue I had was with save() and load() functions. It took me a bit time and some research to figure out how to store vectors to binary file (you need to first store the size of the vector). Then when I test my code by first initializing a bit vector with a larger size, then loading one with smaller size, I kept having this bug where the chunk_ranks are not displaying fully. After some googling, I learned that in my load() function, I need to first clear the original vectors and then resize them with the new size read from the binary file.

Plots





From the plots we can see that it takes constant time to do rank operations no matter the size of the bit vector. This is as expected because I implemented Jacobson's rank method, which

has O(1) time. However, it is worth noting that sometimes the output time would show as 0; this might be within margin errors. As for the overhead() values, the plot shows a linear relationship between the bit vector size and overhead() values, which is as expected, since Jacobson's rank method requires O(n) extra space.

Part 2 - Select

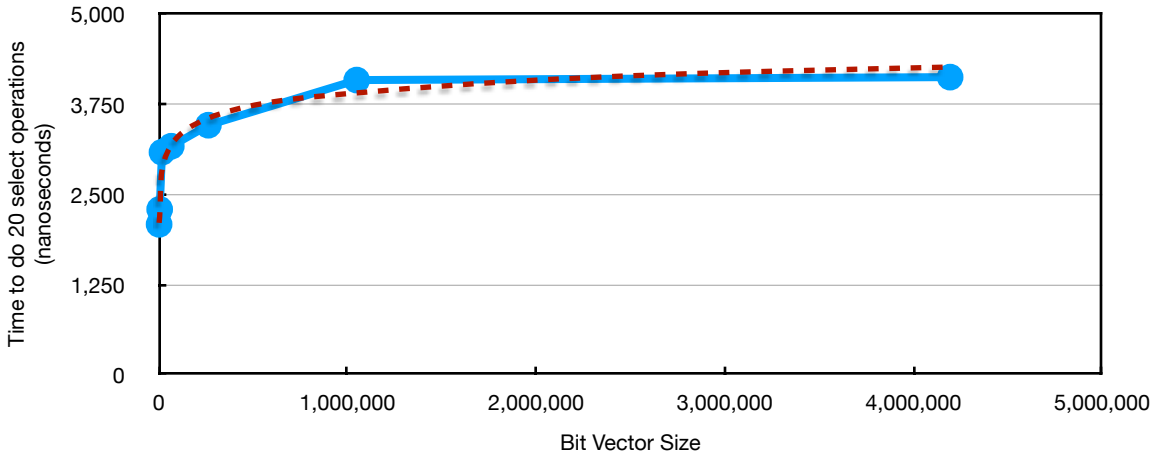
Description

When implementing this part, I reused basically everything from my previous implementation of rank, since we need to let select_support access rank_support, which need to access the bit vector via a pointer. The only function I need to write from scratch is the select1() function. The logic here is straightforward and simple—just use binary search over ranks to get the select value.

Difficulties

One difficulty I had was with dealing edge cases. When traversing till then end in the binary search, if there is a match in rank, but the bit in that index position is not 1, then we need to look at the next one to see if the next one has 1. Another major problem that got me stuck for a while was dealing with non-standard size. This means when the size of the bit_vector is not 2^n; in this way, the bit vector could not be sliced perfectly into (log_2(n))^2 length chunks. To deal with this, I have changed the size of bit vectors in my test code from 10^k to 2^k.

Plots



From the plot we can see that doing a fixed number of select operations takes about O(log(n)) time, which is expected because I went with the binary search method instead of Clark's select. The red dotted line is a Logarithmic trend line which roughly overlaps with the original plot. The overhead function is directly inherited from rank_support and thus the output for overhead() is exactly the same, so please refer to the plots from the previous section.

Part 3 - Sparse Array

Description

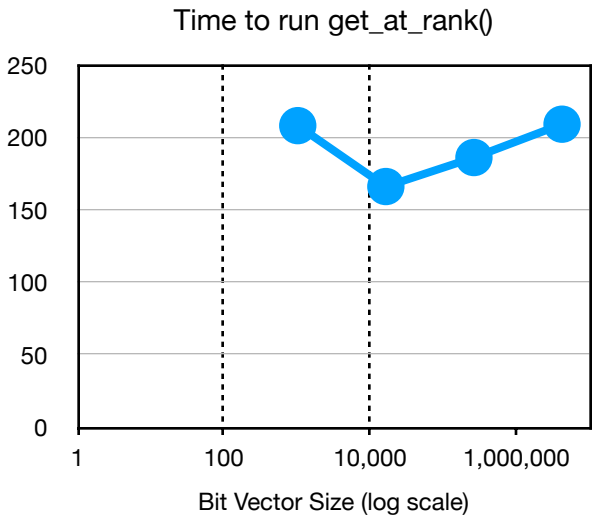
Since we already implemented rank_support and select_support, and we have sdsl::bit_vector for bit vector creation, we only need one more string vector to store the elements, and these are the only four variables in my implementation. When the user creates a sparse_array, a new bit_vector with the given size is created, and after appending the elements to it, we can call the finalize() function after which a new rank_support and a new select_support are created. In functions like get_at_rank() and get_index_of(), we can just use the previously implemented rank and select to get the values we want.

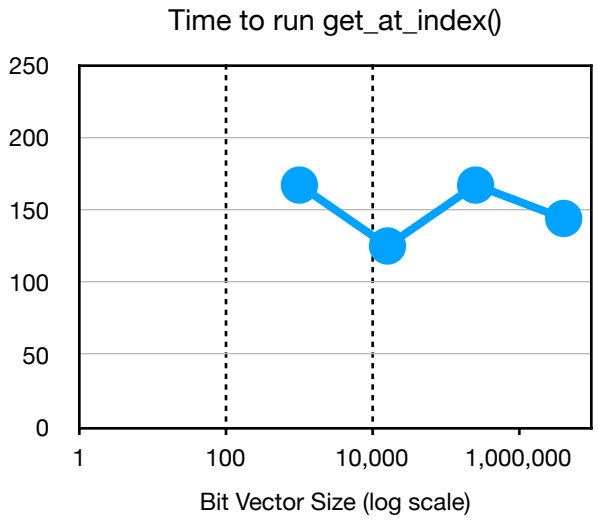
Difficulties

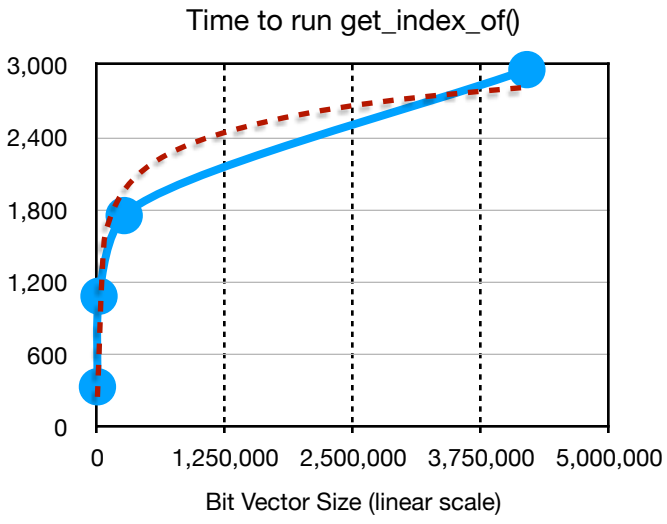
I did not have any major issues when implementing this part, but I did spend quite a lot of time on thinking how to implement this efficiently, and how to take advantage of the previously implemented part as much as possible so I don't need to add more things to it.

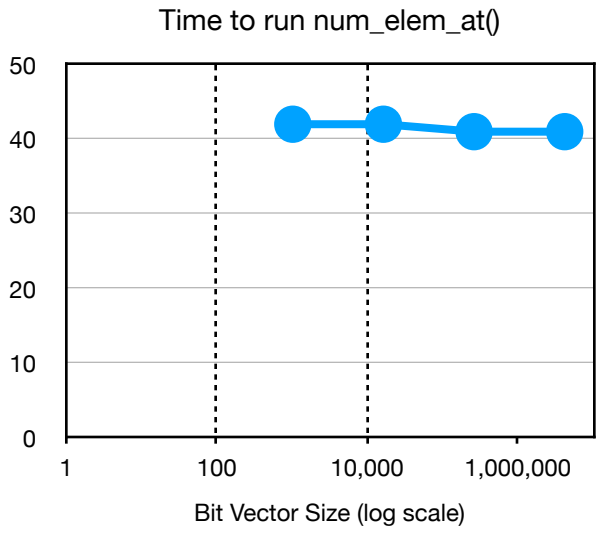
Plots

(a) fixed sparsity, different sizes



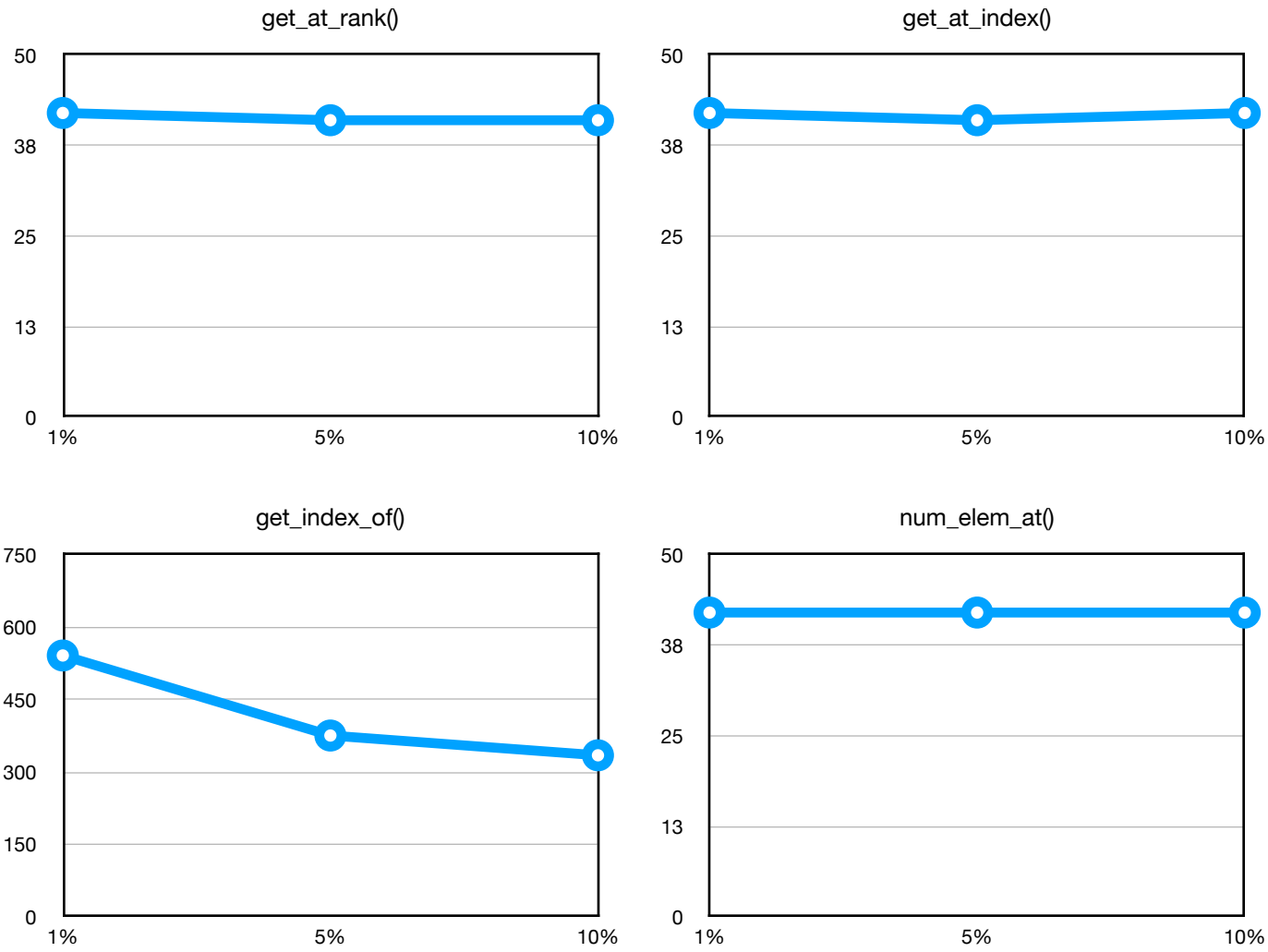






I first tested the time to run the four functions (get_at_rank(), get_at_index(), get_index_of(), and num_elem_at()) on sparse arrays of different sizes (2^10, 2^14, 2^18, 2^22). Note that the sparsity is all 5% for the tested sparse arrays. From the four plots above we can see that the size of the sparse array only affects the speed of get_index_of(), and it looks like it takes O(n) time. This is as expected because only get_index_of() calls select1(), which takes O(n) time, while the other functions either do not call select1() or only call rank1() which takes O(1) time.

(b) fixed size, different sparsities



Next, I tested the time to run the same four functions on sparse arrays with different sparsities (size is fixed for each array). The results show that get_index_of() is once again the only function that seems to be affected by the sparsity of the sparse array.

(c)
If all of the 0 elements are stored as empty elements, then the memory each 0 element takes would be 8 times of what it would take in the sparse array. This is because in the sparse array, it

only needs one bit to indicate empty element, while an empty string needs one byte or 8 bits in memory. In theory, ideally, when the size of the sparse array is large enough that the overhead size is insignificant, and when the amount of 0's in the sparse array is large enough, you can save approximately ⅞ space. However, the savings highly depend on the sparsity. The savings can be maximized when the array is extremely sparse. With the increase of density, the savings will decrease, because the benefit from savings space from 0 elements is diminishing.