858D Programming Assignment 1 Writeup

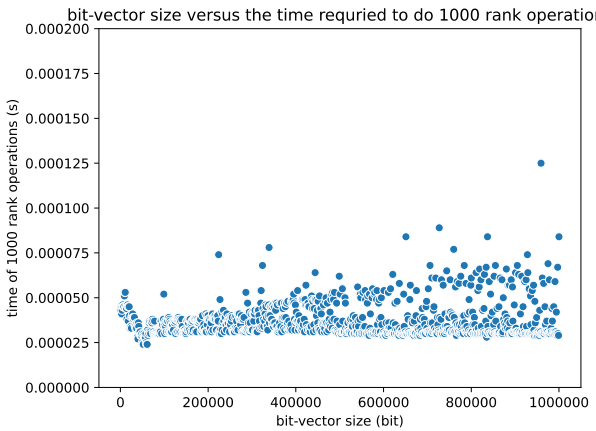
Notes:

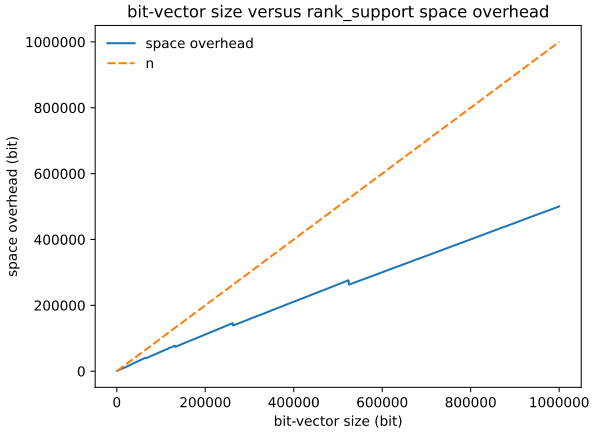
1. Link to github: https://github.com/HenryYihengXu/858D
2. I used compact_vector library for this assignment. I did some modifications to fix bugs and better support my implementations.
3. The program should be compiled with c++20. A makefile is provided.
4. See README for directory structure

Task 1:

The rank_support class contains three compact-vector pointers: b, Rs, Rb. b is the bit-vector passed in, Rs is a vector of super-blocks, and Rb is a vector of blocks. In the constructor, it first computes the following for Rs and Rb respectively: 1. The number of bits in b it covers, 2. The number of bits needed for each super-block/block, 3. The number of super-blocks/blocks, namely the length of Rs/Rb vector. Then it goes through the bit-vector and set values in Rs and Rb. In rank1, it simply locates the correct super-block and block, then sums up the values in them and the popcount of the word. The popcount takes some considerations. It accesses the underlying data of the bit-vector and casts it into words and finds the corresponding 1 or 2 words. Besides, I implemented a to_string function so I can test it effectively. For the save and load, I serialize all b, Rs, Rb, so no computation is needed in load.

One challenge in this task is to figure out how to choose the size of super-blocks and blocks and how to pad them. Inspired by a Piazza question, I choose block size to be ceil(logn) and super-block size to be $(\text{ceil}(\log n))^2$. This made sure that blocks will always align with super-blocks (except the last one). Another difficult part is actually save and load. Although the library provided on piazza has serialize and deserialize, I have to change them so they take in a file stream instead of a file name. Then I can use a single file stream to sequentially serialize all three compact-vectors. It also took me some time to experiment if I can call deserialize a vector on an existing vector without resizing or deallocate it. It turns out I can do that. In addition, the popcount also took me some time.



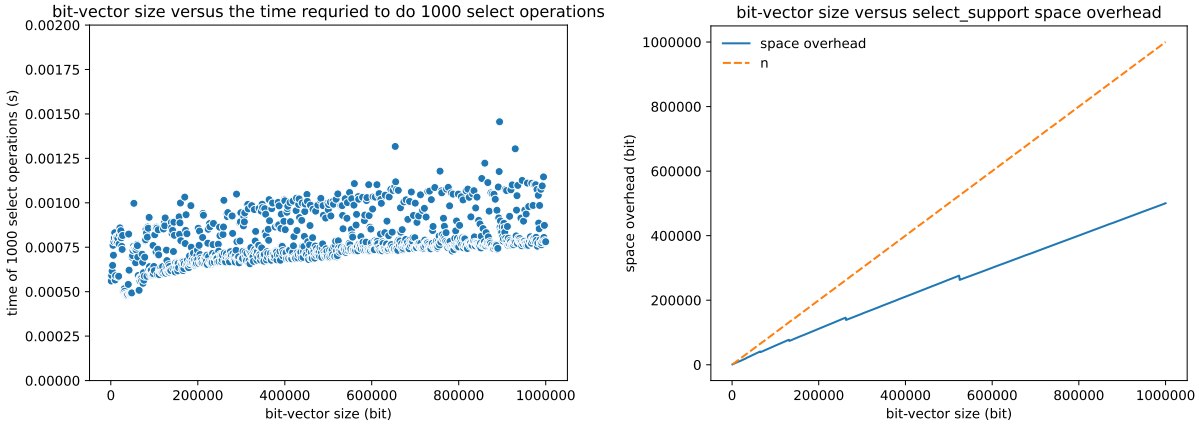


To evaluate my implementation, I tested my rank_support with 1000 hundreds bit-vectors with different sizes in [1000, 2000, 3000, …, 9998000, 9999000, 1000000]. For each rank_support, I run 1000 rank operations and time them. The left plot demonstrate that the rank operation takes constant time as expected. There are many outliers possibly due to system variability. But clearly most points are at the bottom, forming a horizontal line, indicating it takes constant time. The right plot demonstrates that the extra space required by rank_support is strictly less than the size of the bit-vector, which is as expected.

Task 2:

Based on task 1, the select_support is very simple. In terms of data structure, it's nothing but a rank_support. For select1 function, I simply implemented a binary search.

There is nothing really difficult in this task given most difficult issues are in task 1. The only thing needs a bit more consideration is when range in binary search gets really small.



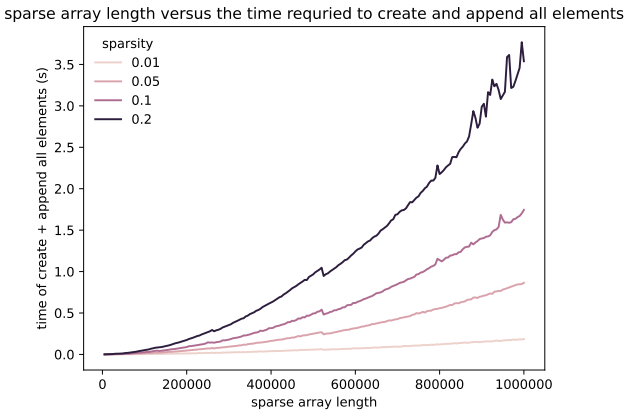
The evaluation setup is the same as task1. The plots contain 1000 data points. The left plot demonstrates that the time for the select operation is logarithmic to the size of bit-vector. Again, although there are outliers due to system noises, you can clearly see most points are at the bottom and forms a logarithmic line. The plot on the right shows the space overhead is strictly less than the size of bit-vector. And it is exactly the same as the rank_support because select_support is nothing but a wrapper of rank_support, nothing else added.

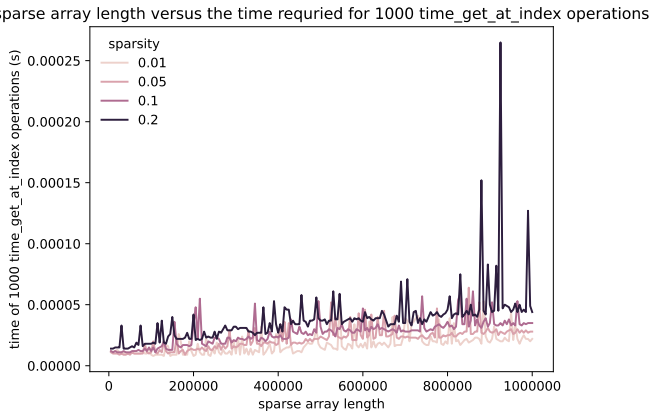
Task 3:

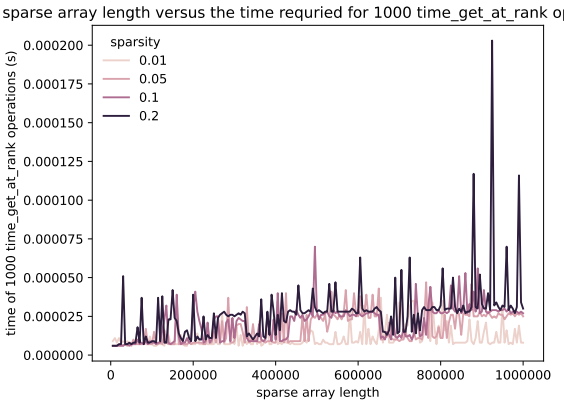
I implemented the sparse_array class to be generic for any type by using c++ template (though I only evaluated string type, and some functions like to_string and overhead may not work for non-basic type). It contains a rank_support pointer and a pointer of a vector that stores the present elements. The create function simply creates an empty bit-vector of length size and

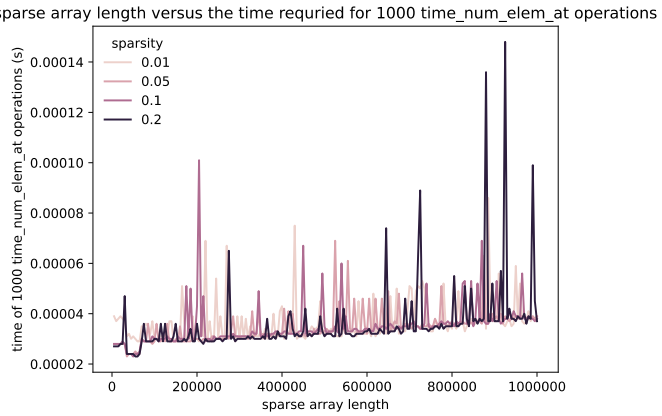
calls the constructor of rank-support to initiate a rank_support for that empty bit-vector. The append function will first check if the position is in-order and in bound. If all good, it appends the element to the vector. Then it sets the position in bit-vector to 1, and update Rs and Rb in the rank_support. For Rs, it will find the correct super-block and increment all super-blocks from that one by 1. For Rb, it will find the correct block and increment all blocks within the same super-block from that one by 1. The get_at_rank function simply pick the rth element in present element vector. The get_at_index function calls the rank operation on the rank_support if the bit is 1. The return value will be the index of element in the present element. The num_elem_at calls the rank operation as well and the rank will be the number of elements up to that position. The save/load first calls the save/load of rank_support. It then serialize/deserialize the present element vector.

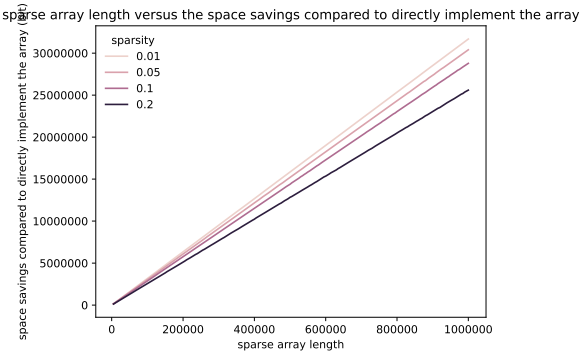
The most difficult part in this task, and also probably the most difficult part for the whole assignment, is to locate and fix a bug in the underlying compact_vector library. The bug is, when creating a new empty compact vector of size n, it somehow randomly has some bits uninitialized. Originally, I assumed by newing a vector, all bits should be set to 0. And the trickiest part is it does set most bits to 0, but has some 1's with a very low probability. I didn't get the error until I tested with bit vectors with about 10,000 bits. A 10,000 bits long vector will have 5-10 bits be 1 instead of 0. I haven't really investigated it, but it's indeed weird because if it simply doesn't call memset, it would have 50% 1's instead of so few 1's. This bug eventually causes my sparse array has some 1's at the before appending any elements. Then after appending all elements, the number of elements won't match with the number of 1's. The fix is very simple, which is explicitly memset the vector after I new it, but the debugging took me a long time. I do want to dive deeper into this bug and potentially contribute to the compact_vector library. If this is a valid discovery, please let me know.











To evaluate my implementation, I tested it with 200 hundred sparse arrays with different sizes in [5000, 10000, 15000, … , 9990000, 9995000, 1000000]. Each size I further tested 4 different sparsity in [0.01, 0.05, 0.1, 0.2]. So in total there are 800 runs. Each line in plots above corresponds to a sparsity.

The first plot shows the total time for creating a sparse array and appending all elements to it. Since append mush be in-order, it's hard for me to test the time after I completely build the array. As the time of appending elements depends on the position, appending to the end of the array after I build it cannot reflect the actual time. Therefore, I timed the total time of building the array with certain sparsities. In theory, a single append should on average take about linear time the number of super-blocks, since it has to increment all super-blocks starting from the insertion location. Then in total it will be $n/\log^2 n * n * \text{sparsity}$, which is close to n^2. The first plot demonstrates this as expected. You can see the lines are quadratic. And as sparsity grows, it takes longer because more append operations are needed.

Plot 2, 3, 4 shows get_rank_at, get_index_at, get_num_elem all have about constant time. This is expected as get_index_at and get_num_elem both simply call the rank operation, which takes constant time. The get_rank_at is simply an array indexing. Moreover, sparsity doesn't really affect the time for them because the rank operation is independent from the sparsity of the bit-vector.

The last plot shows the space saving of sparse array compared with storing 0 as empty strings. For each empty string, it takes 4 bytes or 32 bits to store the pointer. So the saving is 32 * n * sparsity. The last plot correctly shows this: savings grows linearly on n, and lower sparsity leads to more savings.