Programming Assignment 2: Bit vector rank

Student: Jessica Zhu

Github link: https://github.com/jhzsquared/CMSC701\_hw2

**Task 1: bit-vector rank**

My implementation matches the expected theoretical bounds (Figure 1 and 2). With the implementation of Jacobson’s rank structure, we expected the overhead to be approximately O(n) and for the rank operations to be O(1). As shown in Figure 1, which is a plot of the size in bits it took to store the rank structure of a bit-vector of length N, this did in fact take approximately linear space. Likewise, in Figure 2, a plot of the aggregate time it took to conduct 50 rank operations of bit-vectors of length N, this took near constant time.

The most difficult part of the implementation was getting accustomed to Rust, since this was my first time using it, especially figuring out how to organize structures and traits.

Chart, line chart, scatter chart

Description automatically generated

**Figure 1: Rank Overhead**

Chart

Description automatically generated

**Figure 2: Rank Duration**

**Task 2: bit-vector select**

My implementation of bit-vector select using Jacobson’s rank (requiring linear space) successfully achieved no more than log-time select operations as shown in Figures 3 and 4. Figure 3 closely matches with Figure 1 as we use the same Jacobson’s rank structure for select. Figure 4 shows the time it took to conduct 50 random select operations over varying bit-vector sizes. As depicted, it does not require more than log-time to complete.

The most difficult part of the implementation was a getting the binary search to properly obtain the first instance of the request value.

**Chart, line chart, scatter chart

Description automatically generated**

**Figure 3: Select Overhead**

**Chart

Description automatically generated**

**Figure 4: Select Duration**

**Task 3: Implementing a sparse array using bitvector rank and select**

Expect for the get\_index\_of operation (Figure 9) , the speed of the different functions are generally constant as a factor of the overall size (Figures 7,8,10). While get\_index\_of’s speed decreases with increased size, it still appears to hold an at most log-time speed as expected given it is the only one of the functions to require a select operation. Sparsity does not appear to significantly impact the speed for any of the functions (Figures 7-9). Denser arrays may be a bit slower with the get\_at\_index operations (Figure 8) and at extremely large array sizes but otherwise there is marginal impact on speed given the Sparse Array structure used.

Sparsity does make a significant difference on overhead space. If we estimate the size of an explicitly stored empty string to be 32 bits (derived from Rust’s documentation that the offset between an array of chars would be 4 bytes), we result in the estimated overhead size of the “not” sparse array as depicted in Figure 6. We added the number of null values as a factor of the size of the empty string to calculate the updated overhead size of the array (values and rank structure). As you can see between Figure 6 and 5, this results in an at least 2x savings in space even with a sparse array of 10% density. The space savings are even more significant as sparsity increases. At the largest N value tested, this is a savings of almost 4x the space for an array with 1% sparsity.

The most difficult part of this implementation was trying to initialize an empty bit vector with a set capacity because Rust’s bitvec crate was somehow slipping in random bonus true values into the bitvector for large sparse arrays, which then ended up messing up the rank operations.

**Chart

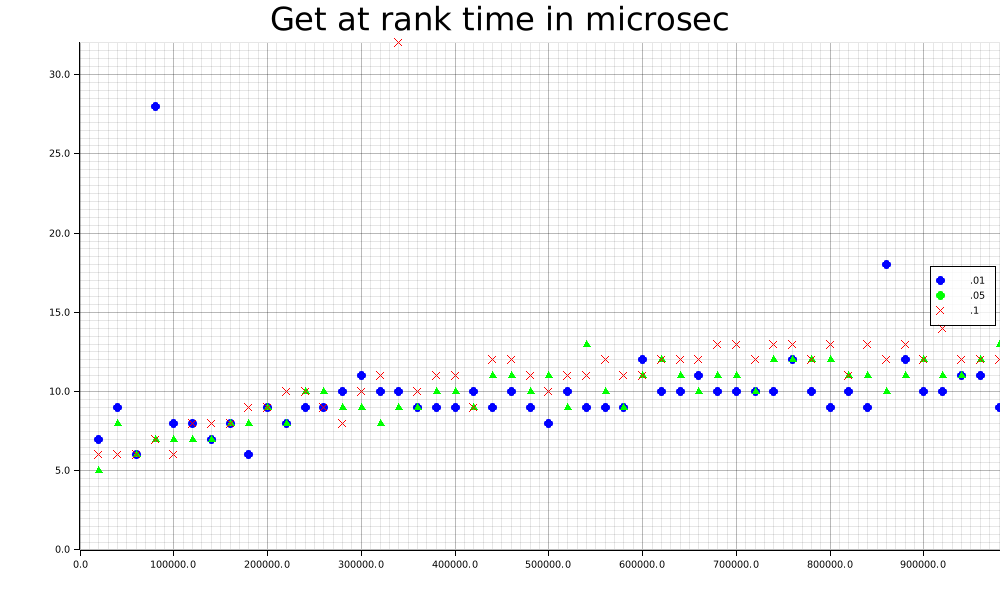
Description automatically generated**

**Figure 5: Overhead of sparse array structure of varying bit-vector lengths**

**Chart

Description automatically generated**

**Figure 6: Hypothetical size of sparse array if “empty” elements were explicitly stored**

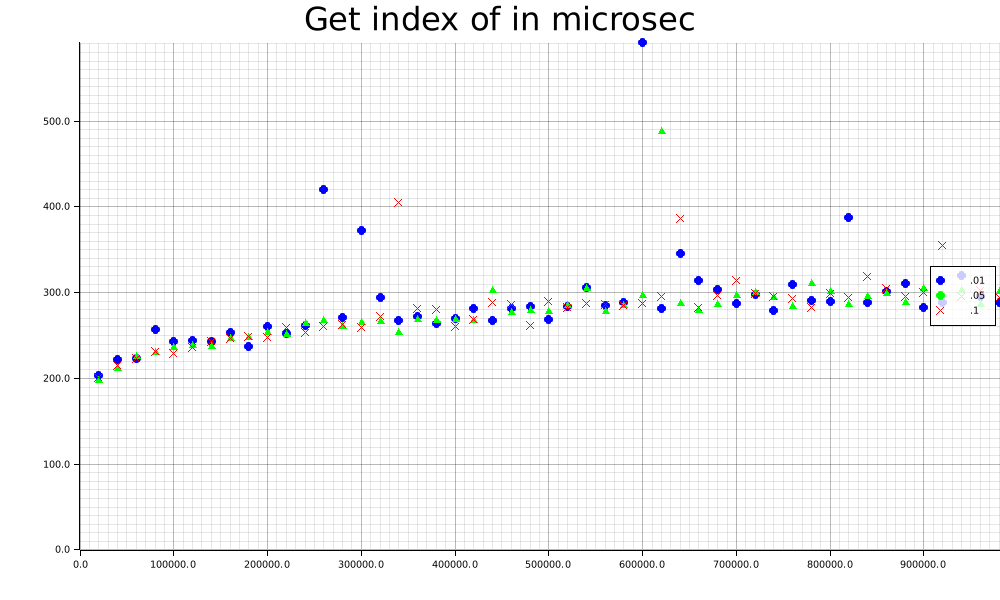
****

**Figure 7: Time required for get\_at\_rank operations (aggregate of 30 trials)**

**Chart, scatter chart

Description automatically generated**

**Figure 8: Time required for get\_at\_index operations (aggregate of 30 trials)**

****

**Figure 9: Time required for get\_index\_of operations (aggregate of 30 trials)**

**Chart, scatter chart

Description automatically generated**

**Figure 10: Time required for num\_elem\_at operations (aggregate of 30 trials**