Bioinformatics, 2024, 05–16

Computational Genomics

Sparse Suffix Arrays with Minimizers

Dev Bhardwaj, Neal Machado and Michael Xie

Abstract

Motivation: Given that suffix arrays are great ways to quickly query large genomes, but often take up a significant amount of storage, we wanted to investigate ways to decrease the amount of storage while maintaining fast query speeds. To limit storage without severely impacting query speeds, we sampled suffix arrays using minimizers.

Results: After testing four different minimizers and observing various parameters for each minimizer such as the window size and the minimizer size, we were able to highlight the storage and speed trade-offs for each configuration.

Contact: dbhardwa@umd.edu, nmachado@umd.edu, mxie1@umd.edu

1 Introduction and Related Work

Sequence comparison is one of the most fundamental tasks in computational genomics. One of the primary ways of completing this task is through suffix arrays, a full-text index that presents all the suffixes of a string in lexicographical order. The problem with traditional suffix arrays, however, has largely been its size. As the traditional suffix array typically uses O (n $\log_2$n) bits, for large enough genomes, the suffix array can be unfeasible. Thus, the challenge arises to mimic the functionality and efficiency of the suffix array while reducing some of the space requirements.

In the past, some ideas to solve this problem included sampling suffixes evenly from the suffix array to create a sparse suffix array (SpaSA). However, a major drawback of this technique is that although it does reduce the space required compared to a traditional suffix array, queries generally take much more time. By sampling suffixes deterministically through minimizers rather than at regular intervals, we hope to retain the benefits of naïve sparse suffix array while providing a more competitive time-space tradeoff with traditional suffix arrays. Because there are many decisions regarding how to sample minimizers, our project aims to determine what combination of minimizer scheme, window size, and minimizer length best mimics the functionality and efficiency of traditional suffix arrays while maintaining significantly less space.

Our sampling method relies upon the idea of minimizers. Given a sequence X of window size *w*, a minimizer length *m*, and an ordering σ, the minimizer of X is the length-*m* substring of X with the earliest ordering under σ. There are many ways to define such an ordering; we will go into each of the ones we used in the next section. Within the space of minimizers there is a good amount of literature and clever minimizer strategies such as weighted minimizer schemes (Jain et al., 2020) or creating an ordering σ using deep learning (Hoang et al., 2022). An important fact to know is that "real sequences rarely follow a uniform distribution" (Hoang et al., 2022), meaning that a lexicographical minimizer may not produce as evenly/sparsely sampled suffixes as may be desired. Instead, a randomly created ordering σ (which can be easily created using a hash function with a fixed seed) will result in a better sampling scheme in the average case.

Furthermore, using minimizers with a window size of *w* offers a theoretical lower bound on density of O(1/*w*), with a random ordering offering "an expected density of O(2/*w*)" (Hoang et al., 2022).

2 Methods

In our project, we tested the performance of four minimizer schemes as well as the default suffix array (labeled "baseline" in our graphs). The first minimizer scheme we considered was a naive lexicographical ordering (labeled "lexicographical"). The second minimizer scheme is a hash-based minimizer (labeled "hash"), where we hash

each subsequence of length m and use the result as the ordering. The third minimizer scheme (labeled "character" in later graphs) was inspired by one of the schemes discussed in Section 2.4 of the 2004 paper *Reducing storage requirements for biological sequence comparison* (Roberts et al. 2004), and essentially works by creating a new ordering for our alphabet based on character frequency. The most frequent nucleotide was ranked with '00', the second most with '01', third most with '10,' etc. Using this character-to-value scheme each of the *m* characters in a minimizer's string was converted to a 2-digit binary string, and the entire string concatenated made up the minimizer's value in the ordering scheme. Our final minimizer scheme (labeled simply "scheme") required ranking each possible minimizer by its frequency in the reference string, and then using these rankings as the ordering scheme.

The final two minimizer schemes, which favor infrequent characters or minimizers, are designed as such with the idea that minimizers which appear more sparsely in the reference have a higher chance of being shared with other windows than minimizers which appear frequently. Another important note about these two minimizer schemes is that they require additional space to store the ordering scheme. Comparing two strings lexicographically doesn't require any extra storage space, and storing a seed for our hash function requires basically no additional space either. For the character-based minimizer scheme, however, a HashMap mapping characters to integer rankings is required, and for the minimizer-frequency based scheme we use a HashMap mapping each minimizer to integer rankings. Note that the minimizer-frequency scheme – along with other intelligent/learned sequence-specific explicit ranking schemes – requires at least $O(|\Sigma|^m \log_2 |\Sigma|^m)$ space where Σ is our alphabet.

In order to test the performance on these variants of minimizer schemes, we constructed the sparse suffix array from the salmonella_sub reference, and then ran a series of queries on this suffix array. We measured how minimizer scheme, window size, and minimizer size play a role in the construction time, query time, and storage space.

2.1 Constructing the suffix array

Prior to constructing the suffix array, we first select values for two hyperparameters, a window size *w* and the minimizer length *m*. Next, for minimizer schemes that require preprocessing we inspect the reference string to create frequency maps of characters/minimizers, and then use this data to create an ordering for all minimizers. After this, we use a sliding window of size *w* over the string and only sample suffixes beginning with the minimizer of a given window. We ensure that no duplicates are added by comparing the tuple containing the index and the suffix with the current last element in the minimizer array. If these two elements are the same, then we have a duplicate and we do not add the current tuple. If they are different, then we can safely add the tuple to our minimizer array. This method also ensures that if two suffixes begin with the same minimizer prefix but are at different positions in the string, we keep both suffixes as they are still unique suffixes. Eventually, our suffix array consists only of the indexes of suffixes that begin with a minimizer prefix. While we are constructing our suffix array, we keep track of the time it takes to complete the entire process. We will use this time to measure the efficiency of constructing the minimizer suffix array for different combinations of window size, minimizer length, and minimizer scheme.

2.2 Querying the suffix array

Something that is important to note prior to querying the suffix array is that the pattern we are querying must have length at least *w*, the window size.

When we are querying a pattern p against the suffix array S, the first thing that we do is find the offset i of the minimizer *m* in the window p[0..w]. Afterwards, we perform a binary search of the pattern p[i..]$ and p[i..]# against the suffix array. The reason for the concatenation of special characters $ and # is to find the start and end index of the range of suffixes that begin with p[i..]. We then loop through each suffix of the suffix array beginning from the start and ending with the end index and verify that the pattern is truly a match. This is necessary because although the suffix may match with the pattern from offset i onwards, there is no guarantee that the characters prior from the offset match. To verify that this is the case, we compare the substring p[0..i] with the characters from the original string preceding the suffix. If there is a match, then we have a successful hit for our pattern.

2.3 Metrics of success

Three elements will define our metrics of success. The first is the time it takes to construct the suffix array. The second is the amount of space the suffix array and any auxiliary structures will take up. The third will be the amount of time it takes to run a set number of queries against our minimizer suffix array.

Before measuring these metrics of success, we first need to ensure the accuracy of our minimizer suffix array. In order to do this, we cross checked the construction of the suffix array as well as the accuracy of the queries with the

code we were given in Assignment 0. We will use the salmonella_sub genome, and compare our results running the sal_sub_reads.naive.map queries with the output provided in Assignment 0. This is also the genome that we will be using the measure the metrics of success.

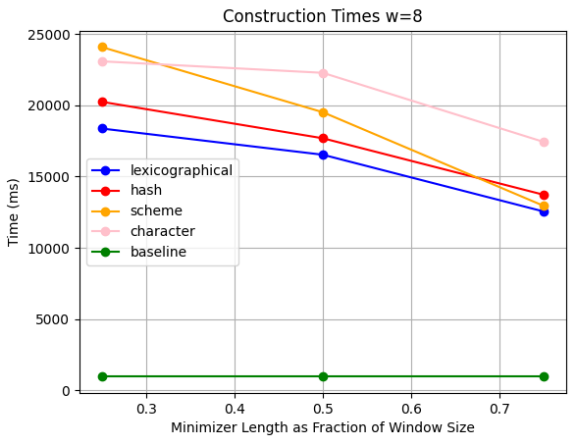
The three parameters we will be measuring these metrics of success against will be window size, minimizer length, and minimizer scheme. For each of the three minimizer schemes we have defined earlier in the paper, we will try different combinations of window sizes and minimizer lengths. For each of these combinations, we measure the amount of time needed to construct the suffix array and any auxiliary structures that may be necessary. We will also measure the amount of space the binary encoded suffix array will take up, along with any additional structures that may be needed. Finally, we will be running a fixed number of queries against our suffix array. The amount of time needed to complete all of these queries will be measured.

3 Results

3.1 Window Size 8 Results

The first window size we decided to test was 8, since each of the queries was generally 8 characters long. Thus, it made sense to have the window size be around the size of the query. We also chose a multiple of four in particular because we decided to express our minimizer lengths as fractions of the window sizes. We chose three minimizer lengths of size 0.25, 0.5, and 0.75 of the window size.

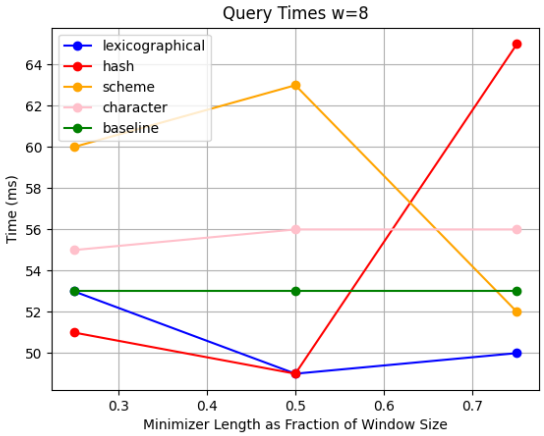
3.1.1 Construction Times



The above graph shows how long it takes for different minimizer schemes at different minimizer lengths as a fraction of the window size 8 to construct a suffix array for the salmonella_sub genome reference. In general, we see that construction time scales inversely with the minimizer length. This makes sense theoretically as a larger minimizer length means less iterations per each instance of a window. All of the minimizer schemes take significantly more time to construct than a traditional suffix array, which is in line with theoretical results as well. Although this may seem very discouraging, it isn't very significant at all because constructing the suffix array should realistically only be a onetime procedure.

Generally, we see that out of all the minimizer schemes, lexicographical and hash-based take less time to construct the suffix array than scheme and character-based minimizer.

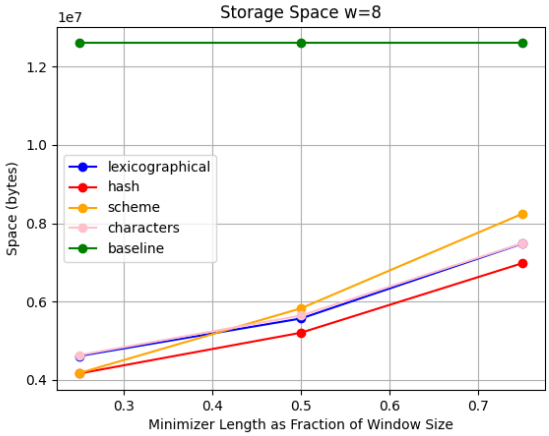
3.1.2 Query Times



This graph shows the amount of time it takes for different minimizer schemes at different minimizer lengths to run all the queries in the read_sal_sub.fq file. All of the minimizer schemes have query times in the range of around ~50-64 ms. Based on these results, it seems like the minimizer length has little to no effect on the average query time. This seems a little counterintuitive because we expected query times to go down as minimizer length increased. This is because we expected the verifying process to take longer for small minimizers since the length of the string to verify would on average be longer. We believe our results are inconclusive partially because the lengths of the queries we were running were too small. We believe that larger queries may better express the downsides of small minimizer lengths as the verification time would be much longer for longer queries, and thus longer window sizes.

On average, we see that all of our minimizers have query times that are competitive with traditional suffix arrays, which is encouraging as it shows that the reduction in space does not lead to a large increase in query time, at least for smaller queries.

3.1.3 Storage Space



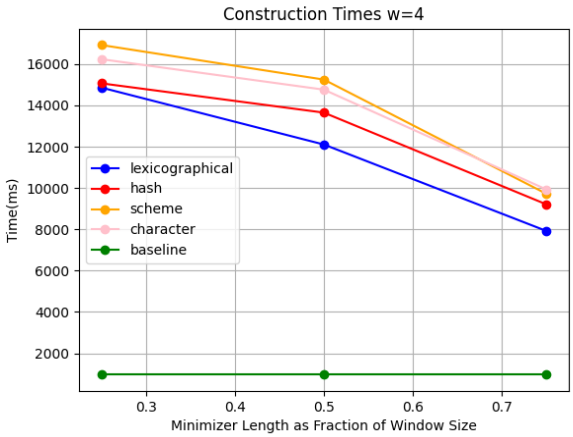
As the ratio between the minimizer length and window size grows closer to 1, the storage space grows what seems closer to linearly. The baseline corroborates this, since if the minimizers' space continues to grow linearly, it would be at within about 10% of the space of the baseline. Conceptually this also makes sense, since at that point, you will store every suffix in the suffix array because the minimizer of the window is just the window itself.

Note that the scheme minimizer uses the most storage out of the minimizers, especially when the ratio of minimizer size to window size is 0.75. We expected that the scheme minimizer would use less storage than most of the others, if not the least, because it considers how frequently minimizers occur to try and reduce the total numbers of minimizers that we will need. However, this approach requires preprocessing to determine frequency and ranking amongst all of the minimizers. Normally a hash map with reference to the minimizers and their rankings wouldn't take up too much space. However, we needed to serialize the hash map and had to store the strings themselves in the hash map. That's why especially at window size of 8 and minimizer size of 6 that we can really see the extra space it takes, because it is storing the length 6 strings of all of the minimizers in the hash map. If given more time, we could have used some reference-based approach to get the same benefits without this scaling cost of storing the strings for each minimizer.

3.2 Window Size 4 Results

The first window size we decided to test was 8, since each of the queries was generally 8 characters long. Thus, it made sense to have the window size be around the size of the query. We also chose a multiple of four in particular because we decided to express our minimizer lengths as fractions of the window sizes. We chose three minimizer-to-window ratios: 0.25, 0.5, and 0.75.

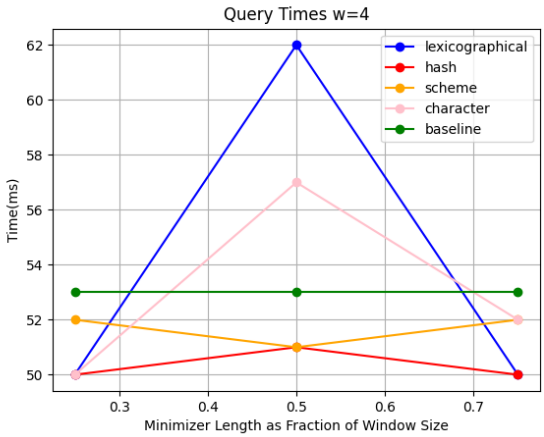
3.2.1 Construction Times



In general, we see the same pattern with a window size of 4 as we did when we used a window size of 8: as *m* increases, we have less possible minimizers per window, so it takes less time to find which is the actual minimizer.

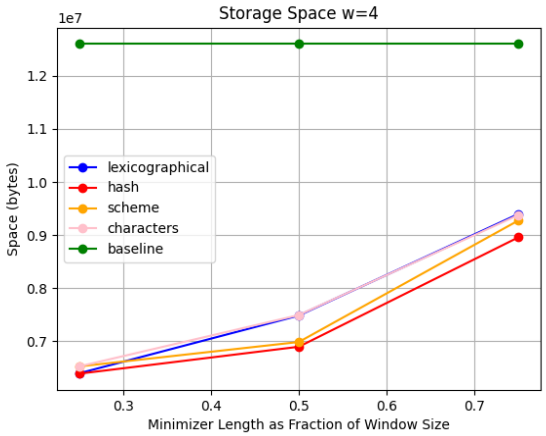
When comparing the minimizer schemes to each other, we see that the frequency-based minimizer scheme (labeled "scheme") has the longest construction time, which follows intuitively from the fact that it requires the inspection of every minimizer in the original reference. The character-based minimizer scheme also requires inspection of the whole reference string, and it takes the next longest to construct. The hash scheme takes slightly longer than the lexicographical minimizer, probably because of the extra time it takes to hash the strings instead of comparing them directly.

3.2.2 Query Times



The results for window size 4 are very similar to our results for window size 8 in the sense that all the minimizer schemes take a similar amount of time to run the fixed set of queries. We also see that it is consistent with the fact that minimizer length does not seem to have a clear effect on query time. This could also be because of the queries being too small. An interesting pattern to note is that for some of the minimizer schemes, it seems as if having the minimizer be at half the window size yields slower query times than at 0.25 or 0.75 the window size. Although there may be a reason behind this, it could also easily just be because of variance. In terms of relative performance to the traditional suffix array, we see that many of the minimizer schemes do just as well if not better, which is very encouraging.

3.2.3 Storage Space



The storage space follows a similar trend overall as when the window size was 8. However, we see a slightly different ordering among the minimizers. Lexicographical, character-based, and hash-based all are almost the same relative to one another. Due to how we store the preprocessing hash map for scheme (see Section 3.1.3), scheme doesn't take the most storage space when window size is 4. To reiterate, scheme stores the strings of the minimizers as the keys for ranking them by frequency, and when the minimizer size is 3, it doesn't take much space to store the hash map. Therefore, even with the extra preprocessing data structure, the scheme minimizer can reduce the number of minimizers we store to still take less space overall.

4 Conclusion

Overall, we found that sparse suffix arrays with minimizers were very competitive with traditional suffix arrays in terms of querying performance while taking significantly less space. One of the noticeable downsides of minimizer schemes would be the construction time of the suffix array, but this should not be a very relevant issue because as mentioned earlier, the construction of the suffix array should only need to be done once.

Most of the minimizer schemes we chose are comparable (modulo a couple trade-offs), but if we had to recommend just one minimizer scheme, we would choose the simple hash-based minimizers. This scheme takes the least amount of storage space and has the second lowest construction time of all the sampled suffix arrays (lexicographic performs better, but construction is a one-time occurrence). Query times for the hash-based suffix array were the least for w=4 and fluctuated for w=8 (which could be due to variance).

4.1 Storage Space

All of our minimizer-based suffix arrays took a lot less space than the original, non-sampled suffix array, with the most storage improvements occurring when $m << w$. Even with the $\Omega(|\Sigma|^m \log_2 |\Sigma|^m)$ of extra space that storing an explicit ordering σ (big Omega notation since we use a HashMap), our storage space still is quite small for the frequency-based minimizer, which leads us to believe that computing/training other explicit minimizer orderings is still a useful endeavour.

4.2 Future Work

In the future, something that could be expanded upon would be incorporating a larger dataset and generating much larger queries for that dataset. With larger queries, the verification step will take much longer. Therefore, we will be able to determine whether minimizer scheme and minimizer length has a significant impact on the query time. If query times remain competitive with a standard suffix array, then minimizers are an extremely valuable strategy for reducing storage of suffix arrays by a constant factor.

Appendix

Code for the Experiment:
https://drive.google.com/file/d/1hSJvdCQgruBSqvSGy46_PxmRWgT6EzCo/view?usp=sharing

Acknowledgements

Thank you to Rob Patro and Noor Singh.

Bibliography

Fischer, J., I., T., & Köppl, D. (2015). Deterministic Sparse Suffix Sorting in the Restore Model. *ACM Transactions on Algorithms (TALG), 16*, 1 - 53.

Grabowski, S., & Raniszewski, M. (2015, September). Sampling the suffix array with minimizers. In International Symposium on String Processing and Information Retrieval (pp. 287-298). Cham: Springer International Publishing.

Hoang, M., Zheng, H., & Kingsford, C. (2022). Differentiable learning of sequence-specific minimizer schemes with DeepMinimizer. *Journal of Computational Biology*, 29(12), 1288-1304.

Kosolobov, D., & Sivukhin, N. (2021). Construction of Sparse Suffix Trees and LCE Indexes in Optimal Time and Space. *ArXiv, abs/2105.03782.*

Roberts, M., Hayes, W., Hunt, B., Mount, S., Yorke, J., Reducing storage requirements for biological sequence comparison, *Bioinformatics*, Volume 20, Issue 18, December 2004, Pages 3363–3369, https://doi.org/10.1093/bioinformatics/bth408

Prezza, N. (2016). In-Place Sparse Suffix Sorting. *ACM-SIAM Symposium on Discrete Algorithms*.