Minimizer-Space Suffix Array Techniques: Implementation and Analysis

Ryan El Kochta
UID: 117886092
relkocht@umd.edu

Tien Vu
UID: 117157321
vut@umd.edu

CMSC 701
Spring 2025

Abstract

The string search problem is a common problem across computer science, including bioinformatics. A standard suffix array is a data structure that enables faster string searches in exchange for requiring additional storage space. At the same time, minimizers are k-mers selected to represent some group of k-mers in a string. Combining the two results in a minimizer-space suffix array, a suffix array computed not over characters but over minimizers representing some set of characters. In this paper, we discuss our implementation of such a data structure, its performance, and its storage space requirements. The implementation is open-source code which can be found at https://github.com/cephi-sui/mssa.

1 Introduction

The *string search problem* is a common problem across computer science, including bioinformatics. In the string search problem, we are presented with a reference string S, and we aim to find the index of (as we consider, *any* occurrence) of a query string Q in the reference string, if one exists.

The naïve approach to solving this problem involves moving a sliding window of length $|Q|$ along the string S, and at each iteration comparing the window to the query string Q. We don't need any additional data structures to do this, but since the strings must both be compared at each step, this has time complexity $\mathcal{O}(|S|\ |Q|)$, which is quite slow.

An alternative approach is to use an *index,* which is pre-computed and stored instead of the original string, and allows faster lookups. A common index is the *suffix array,* wherein the indices of each suffix are computed and stored in lexicographical order. This allows a binary search to be performed in $\mathcal{O}(|Q|\log|S|)$. This is better, but in practice this is not fast enough for very large strings.

For this project, we implemented a possible accelerant for this algorithm from [1]. Specifically, instead of computing the suffix array over the raw bytes in the reference string, we compute the suffix array over *minimizers* of the k-mers in the reference string. Consecutive windows (represented as *super k-kmers*) with identical minimizers are deduplicated at build time, leading to a possibly much smaller amount of data to be stored in the index. At query time, queries are converted into a sequence of super k-kmers (represented by minimizers). We then perform the usual suffix array binary search algorithm, but with (ideally) smaller constants, since a relatively small number of minimizers need to be compared rather than comparing the query string to each suffix character-by-character.

In this report, we describe the following. First, we give some additional details about the minimizer-space suffix array construction, and its construction parameters k and w. Next, we describe our implementation of the above construction. We then describe some interesting findings from Sapling [2] and PLA Search [3], which gave us another optimization that we implemented, based on techniques for piecewise linear regression described in [4] and implemented in the Rust crate `plr`. Lastly, we describe another optimization in maximizing the significance of minimizer matches inspired by the original minimizer paper [5]. Finally, we visualize and discuss our results and conclusions.

2 Methods

2.1 The Minimizer-Space Suffix Array Construction

First, at construction time, the reference string is split into k-mers, which are overlapping substrings of length k in the string. Then, the algorithm computes the *minimizer* within each *window* (of configurable size w) over the k-mers. Consecutively identical minimizers are de-duplicated into super-k-mers and stored alongside their index range in the underlying string.

As an example, consider the string:

`GLOBGLOGABGA`

Assuming $k = 5$ and $w = 3$, we obtain the following windows of k-mers, along with the minimizers within those windows (using lexicographical ordering):

GLOBG	LOBGL	OBGLO
LOBGL	OBGLO	*BGLOG*
OBGLO	*BGLOG*	GLOGA
BGLOG	GLOGA	LOGAB
*GLOGA	LOGAB	OGABG
LOGAB	OGABG	*GABGA*

The consecutively-de-duplicated minimizer chains are stored:

- `GLOBG`
- `BGLOG`
- `GLOGA`
- `GABGA`

alongside an encoding of their start and end positions. These are known as super-k-mers.

The suffix array is constructed over these super-k-mers, and at query time, the query string is converted into a sequence of super-k-mers and a normal suffix array binary search is performed. Many fewer comparisons (at least, in theory) are performed, leading to possibly improved performance. However, a brute-force over the resulting super-k-mer window must be performed on the *original* string after finding a match, since the minimizer chain matching is not certain to imply an actual match. (We call the rate at which this must happen the *false positive rate*).

2.2 Our Implementation

We created an implementation of the above construction in Rust. Our implementation is structured as follows. We conceptually "transform" the reference string into a minimizer-based string at the beginning. We begin by compressing the alphabet from a list of raw bytes (each of which can take on any value) into a new alphabet mapped only from the characters that actually occur in the string. For example, for a DNA sequence, the alphabet size will be 4 (A, C, T, G), and as a result each character only take up 2 bits in the transformed alphabet.

2.2.1 k-mer representation

The next problem is how to represent the k-mers efficiently. For this, we have an `enum Kmer` that represents a k-mer, and can either be the sentinel k-mer \$ or a k-mer represented using an *int vector,* which was mostly adopted from one group member (Ryan)'s implementation from Assignment #2. The int vector implementation uses the rust `bitvec` crate under the hood. (This was probably a mistake, as we will justify in a moment).

We were initially torn between three ways of representing k-mers:

1. Representing them as just an index into the underlying substring.
2. Representing them as a Rust *slice* into the underlying substring.
3. Representing the *owned* sequence of characters in the k-mer.

The issue we initially had with option 1 was that this is unergonomic; every time we want to access some property of the k-mer, we have to pass around a reference to the underlying string and index into it, in addition to making the error-handling more tedious.

Our issue with option 3 was the asymptotic space complexity of this approach; as k grows large, this requires well above linear space in the size of S. Similarly with option 2, even though Rust slices don't store a copy of the underlying data (and are basically just fat pointers with a base and length), when we try to serialize this, `bincode`/`serde` will create a copy of every k-mer, which has the same space complexity issue as option 3.

We settled upon option 3 upon realizing that the maximum integer size we have in Rust on a 64-bit machine is a `u128`; for an alphabet of size 4, this means we can have up to $k \approx 64$ and still bound the k-mer representation's size to two machine words of data. In the real world, this is plenty.

The mistake was using the exact `IntVec` structure that we used. As we later found, there is a *lot* of overhead from this structure, from the bit count that needs to be stored alongside of it to the overhead from the implementation details of the `bitvec` crate. (Accessing `IntVec` elements is also indirect, since we have to follow a pointer to the heap). We suspect that this is a primary driver of our high memory usage; as we will discuss later,

we likely should have stored this as a `u128` and enforced that the k-mer fits within this bound.

2.2.2 Constructing the index

In addition to `Kmer`s, we also have a `KmerSequence` data structure, which stores a `Vec<Kmer>`, along with `occ`, which stores the relative abundance of each k -mer (to be used for different minimizer orderings, as will be discussed).

After constructing an alphabet, a `KmerSequence` is constructed from a byte string. From the `KmerSequence`, a `SuffixArray<T: QueryMode>` is constructed; the `QueryMode` trait allows us to define many different forms of queries. We implemented the following query modes:

1. `GroundTruthQuery`, which performs the naïve $\mathcal{O}(|S|\ |Q|)$ string search (not even with a suffix array).
2. `StandardQuery`, which performs the basic minimizer-space suffix array query, with no further accelerants.
3. `PWLLearnedQuery`, which adds to the `StandardQuery` a lookup in a piecewise linear approximation function based on the first minimizer in the query, significantly narrowing down the search space. See Section 2.3.

The suffix array construction logic involves constructing a minimizer chain from the underlying k-mers and de-duplicating them into `SuperKmer` structures, which encode both their start positions and lengths (which are necessary for the "brute-force" used to validate the minimizer chain at the end of the query process). Currently, this is implemented naïvely using an $\mathcal{O}(nkw)$ algorithm. We did implement a linear-time algorithm for this using a monotonic queue, which did help index construction time a bit, but unfortunately this code was lost in a tragic git accident. The suffix array is then naïvely constructed over this minimizer chain.

The query logic is as described earlier.

2.3 Piecewise Linear Regression (PLR) Accelerant

Sapling [2] describes an approach that relies on a key observation: when binary searching for the query string, because of the fact that the search window grows in half at each iteration, there are very few iterations at the beginning of the search where the search window is too large to fit in a single cache line. So if we had a function, say $f(x)$, that could, given the first m minimizers in Q represented as an integer x (ordered in the same way as the minimizer chain), give an *approximate* location in the suffix array where Q would be placed in the given ordering, we could shrink down to a cache line much more quickly, possibly dramatically improving performance for many queries.

During construction, given some function $f(x)$, we compute over all x and then compare to the true position in the suffix array. The maximum delta between these two values for *any* x is E, and is configurable in our implementation as γ. So upon querying, we consult the function $f(x)$ given the first minimizers of the query string, and create a window from $[f(x) - E, f(x) + E]$. These are the initial bounds of the binary search.

How do we create this function, then? Sapling describes two approaches: first, using a neural network, and second, using a *piecewise linear function (PWL)*. We are interested in the second approach. Essentially, the spectrum of all inputs x is modelled as a "roughly" linear function outputting the position in the suffix array. Figure 1 shows the ground-truth for this function for both coronavirus and monkeypox genomes.

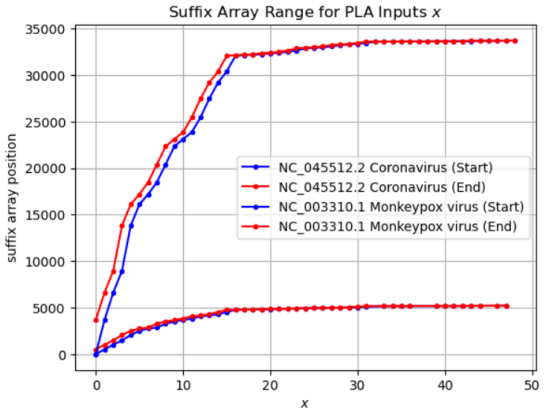

Figure 1: Suffix array positions over each k-mer, with k=3, w=10

The idea is to model this function as a piecewise function consisting of *linear approximations,* such that we minimize the number of segments in the piecewise function while guaranteeing a maximum error bound for any evaluation of the function.

The thing we did not understand about the approaches of [2] and [3] is: a suffix array position for a set of beginning minimizers isn't a single position, it's a *range* of positions. We do not understand how both papers are able to sidestep this; instead, we made two piecewise-approximation functions, one for the beginning of the range starting with a sequence of minimizers and one for the end of that range. That way, we are able to, given a query string Q, quickly determine the start and end range, to within some maximum error $E = \gamma$. The bounds we begin the binary search at are $\left[f_{\text{begin}}(x) - E, f_{\text{end}}(x) + E\right]$.

Abrar and Medvedev [3] point in the direction of *O'Rourke's algorithm,* which is implemented

in the Rust `plr` crate. We use the `GreedyPLR` with a command-line-configurable γ, and construct a piece-wise linear regression-based approximation of $f_{\text{begin}}(x)$ and $f_{\text{end}}(x)$ alongside suffix array construction. We also simplify the construction by only considering the *first* minimizer, which for strings of the lengths we looked at was enough to narrow down the search window to single-digit percents of the full suffix array (depending on the query and value of γ). This helps to explain the (small) performance increase we observe from this query mode later on.

2.4 Minimizer Match Significance

In "Reducing storage requirements for biological sequence comparison" [5], there is discussion on the "orderings" of the alphabet used to determine the minimizers. The paper uses a lexicographic ordering of the alphabet but notes that it is not desirable to do so because a string with consecutive characters will result in several consecutive k-mers being selected as minimizers. In terms of minimizers as a concept, this "counteracts [the] goal of sampling a fraction of the k-mers" [5]. The paper goes on to suggest alternate orderings of the alphabet, such as assigning values based on frequency of a letter's occurrence. In conclusion, they state "in general, we want to devise our ordering to increase the chance of rare k-mers being minimizers, thus increasing the statistical significance of matching minimizers." [5]

In the context of this suffix array implementation, we would like to maximize the statistical significance of matching k-mers. Doing so can theoretically help with false positives – the cases where the minimizers of a suffix array entry and query match but the actual strings represented by those minimizers do not. Moreover, false positives are not computationally or temporally cheap as they require manually comparing every character for every matching super k-mer. Ideally, this would never have to be performed; it must be performed to prevent erroneously returning a match when there was none. As such, reducing the number of false positives reduces the total amount of raw character comparisons.

Initially, our approach to increasing the statistical significance of matching minimizers involved following the paper directly and applying some of the alternate alphabet orderings they proposed. However, as previously quoted, the paper states that the ultimate goal of these alternate orderings is "to increase the chance of rare k-mers being minimizers." [5]. This suggests that alternate alphabet orderings were a means to an end – proxies to make the process of selecting rare k-mers as minimizers easier. Instead of comparing different ways to order the alphabet such that we increase this chance, what if we just maximized the chance with a data structure?

In conclusion, false positives in this data structure are a source of wasted computation and time. The number of false positives can be reduced by increasing the statistical significance of matching minimizers. The statistical significance can be increased by maximizing the chance that rare k-mers are selected to be minimizers. This chance can be maximized by creating a hash map of every k-mer, storing the number of times they appear, and using these occurrences to determine the minimizer of any given k-mer window.

The creation of the hash map takes $\mathcal{O}(n-k)$ time and it just adds an additional $\mathcal{O}(1)$ computation to the k-mer comparison for minimizer determination. This implementation also has the added benefit of filtering out query sequences before the suffix array is ever queried. Since the query needs its minimizers computed based on the occurrence of k-mers in the reference string, if the query ever has a k-mer that does not appear in the hash map, then the query can be rejected immediately and in $\mathcal{O}(n-k)$ time.

3 Results

In order to evaluate the performance of our suffix array, we compared its performance building and querying three datasets: complete sequences of two viruses and one bacteria. The first virus – and the smallest – was the Zika virus [6] weighing in at 11,000 bases. The second virus was the Monkeypox virus [7] weighing in at 196,000 bases. And finally the bacteria was Chlamydia pneumoniae TW-183 [8] with 1.2 million bases. These sequences were selected to obtain a spread of sequence lengths and types. Due to time constraints and performance problems, we decided not to experiment with longer sequences such as the Y chromosome of Drosophila melanogaster [9]. Such limitations are discussed further in Section 4.

For each sequence, we built and queried three minimizer-space suffix arrays: a "standard query" with lexicographical ordering of the k-mers (hereby referred to as the baseline suffix array, despite still being minimizer-based), another standard query with occurrence-based selection of the minimizer k-mers as described in Section 2.4 (referred to hereon as just the occurrence suffix array for simplicity despite *not* being related to alphabet occurrence-based ordering), and a "piecewise linear function (PWL) learned query" which uses the piecewise linear regression accelerant described in

section Section 2.3 using $\gamma = 10$. This will be referred to as the PWL suffix array.

Subsequent visualizations will also display each sequence based on length, since size is a major reason for the results depicted. From left to right, the sequences depicted are Zika, Monkeypox, and Chlamydia.

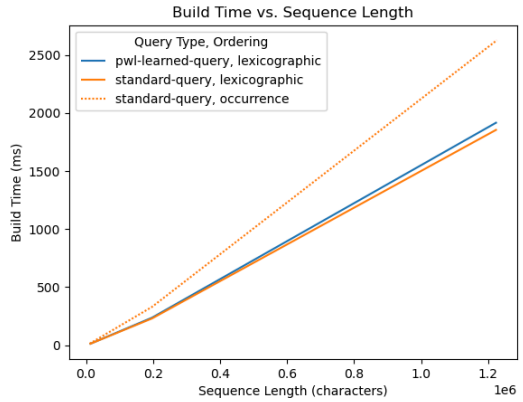

Figure 2: Build Time vs. Sequence Length
k = 3, w = 3

Naturally, the suffix array must be built before it can be queried. Figure 2 visualizes a comparison of the build time for each suffix array. Note that the PWL suffix array adds a small amount to the build time over the baseline suffix array. This contrasts heavily with the occurrence suffix array, which adds quite a bit of build time.

This is great for the PWL approach and not so much for the occurrence approach, since the occurrence suffix array needs to iterate over the set of every k-mer in the sequence, hash it, and store an occurrence tally. Fortunately, we see that this does not incur an extremely large storage requirement in Figure 5 and provides a noticeable performance increase in Figure 3.

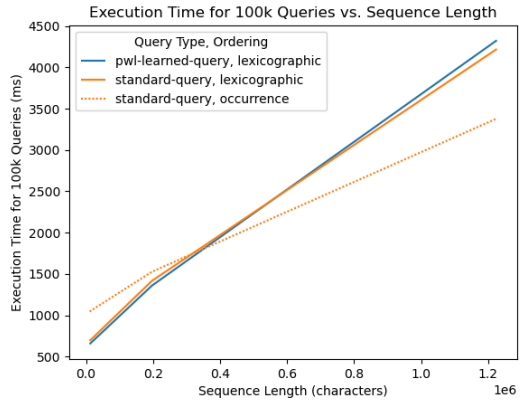

Figure 3: Execution Time vs. Sequence Length
k = 3, w = 3

Once the suffix array is built, 100k queries are run on each. These queries are randomly generated with 90% of them being some random subsequence from the sequence itself and the other 10% being random sequences with the same alphabet as the queried sequence.

We can see that the PWL suffix array performs quite similarly to the baseline suffix array. It manages to have a lower execution time for both viruses but actually exceeds the baseline suffix array for Chlamydia. This is likely due the γ staying constant across sequences instead of scaling relative to the size of the suffix array.

We can also see that the occurrence suffix array performs almost inverse to the other two suffix arrays. For smaller sequences, it performs worse, likely as the cost of looking up the occurrence of every k-mer in the query string outweighs the benefits gained from doing so. However, for Chlamydia, it shows a great performance increase with about a 20% decrease in execution time compared to the baseline suffix array. The next graph, Figure 4, depicts why this may be the case.

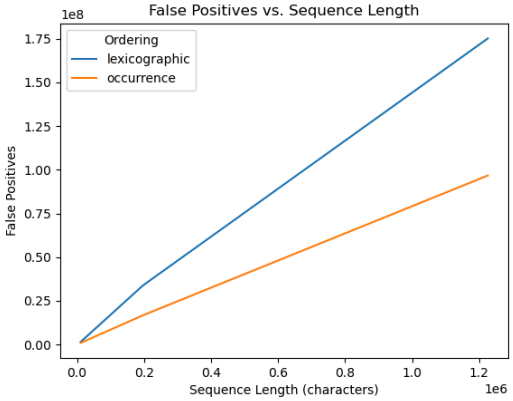

Figure 4: False Positive vs. Sequence Length
k = 3, w = 3

As described in Section 2.4, the theory behind selecting minimizers using k-mer rarity is that it increases the significance of matching minimizers, thus decreasing the false positive rate and the time and computation spent comparing non-matching subsequences. Figure 4 confirms that this approach works as the the occurrence suffix array has just a fraction of the false positives that the baseline does. For Chlamydia, the occurrence SA has 45% fewer false positives than the baseline suffix array. As per Figure 3, this translates quite well into a decrease in execution time for longer sequences.

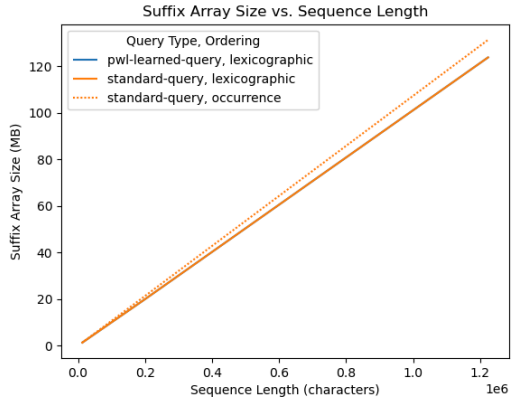

Figure 5: Suffix Array Size vs. Sequence Length
k = 3, w = 3

Finally, Figure 5 compares the suffix array data structure sizes, with all suffix arrays maintaining a similar size. Notably, the PWL suffix array and the baseline suffix array sizes are so similar that they only have about 1 kB of difference between them at most, demonstrating the PWL suffix array's high efficiency in storage. Also notably, these sizes are about 100x the size of the original sequences. For comparison, the suffix array created by us for assignment one takes just 11 MB for Chlamydia. Meanwhile, the FASTA file for Chlamydia pneumoniae TW-183 takes just 1.2 MB.

4 Discussion

For starters, the storage size of the suffix array for both optimizations are poor because they extend off of a poor base. There are quite a few overlooked places where storage sizes could be cut down drastically, such as our `Kmer` struct which stores a (compressed) copy of the entire k-mer. Ideally, we would use a different abstraction, such as our `KmerSequence` struct, to store a compressed version of the original string and then simply index into the string to obtain k-mers. This is not only a large consumer of storage space but also memory. When attempting to construct the suffix array on chromosome 1 of the human genome, we found that our program actively consumed 20 GB of RAM. Seeing as the sequence was 200 MB, this follows the 100x sequence size pattern found previously in Section 3. We found that this memory usage occurs despite the fact that not all k-mers need to be in memory during suffix array construction! Unfortunately, we were not able to remedy this problem given time constraints.

Notably, all of the results in Section 3 are bound by $k = 3, w = 3$. Ideally, we would have liked to see how other k and w values effect our optimizations. Unfortunately, the volume of data was not easy to parse for a succinct results section. Similarly, we would have liked to see how varying γ Finally, we would have preferred to compare against existing implementations of minimizer-space suffix arrays but were unable to get them up and running within time constraints.

One possible source of variation in our results is the way queries were randomly generated on a per-sequence basis for the purposes of benchmarking. Originally, we used one constant set of queries but found that it may affect execution time for different sequences. This is because the match rate would differ wildly between sequences given the same shared set of queries, causing different amounts of extraneous computation in the case of non-matches. Our per-sequence query generation solved this problem, but introduced a new one wherein two given runs of a specific suffix array on a specific sequence could have slightly different results. An easy solution to this would to have generated a random set of queries for each sequence and reused those queries instead of generating them on a per-run basis.

One extension we would have loved to have made is the use of a minimal perfect hash function in the occurrence suffix array. Although the hash map does not incur a huge storage penalty over the baseline suffix array, we would be interested in seeing much we could close the gap between them. To this end, there would have been a great Rust crate in the form of `boomphf` [10].

5 Conclusion

This paper was an interesting examination into an alternate construction of suffix arrays and possible optimizations to such constructions. We found two optimizations to the basic idea outlined in Section 2.2, piecewise linear regression and occurrence-based minimizer selection. Section 3 goes over our findings, where we find mostly positive results in execution time, very positive results in false positive reduction, and negative results in storage size. These results point a direction forward for further research in minimizer-space suffix arrays, especially since these optimizations can be further fine-tuned. We also greatly enjoyed learning about the developing such a data structure from the ground up (especially in Rust), even if it meant that we were not able to spend as much time optimizing it.

Bibliography

[1] B. Ekim, B. Berger, and R. Chikhi, "Minimizer-space de Bruijn graphs: Whole-genome assembly of long reads in minutes on a per-

sonal computer," *Cell Syst.*, vol. 12, no. 10, pp. 958–968, Oct. 2021.

[2] M. Kirsche, A. Das, and M. C. Schatz, "Sapling: accelerating suffix array queries with learned data models," *Bioinformatics*, vol. 37, no. 6, pp. 744–749, 2020, doi: 10.1093/bioinformatics/btaa911.

[3] M. H. Abrar and P. Medvedev, "PLA-complexity of k-mer multisets," *bioRxiv*, 2024, doi: 10.1101/2024.02.08.579510.

[4] Q. Xie, C. Pang, X. Zhou, X. Zhang, and K. Deng, "Maximum error-bounded Piecewise Linear Representation for online stream approximation," *The VLDB Journal*, vol. 23, no. 6, pp. 915–937, Dec. 2014, doi: 10.1007/s00778-014-0355-0.

[5] M. Roberts, W. Hayes, a. R. Hunt, S. M. Mount, and J. A. Yorke, "Reducing storage requirements for biologicaluence comparison," *Bioinformatics*, vol. 20, no. 18, pp. 3363–3369, 2004, doi: 10.1093/bioinformatics/bth408.

[6] G. Kuno and G.-J. J. Chang, "Full-length sequencing and genomic characterization of Bagaza, Kedougou, and Zika viruses," *Arch Virol*, vol. 152, no. 4, pp. 687–696, Jan. 2007.

[7] S. N. Shchelkunov *et al.*, "Human monkeypox and smallpox viruses: genomic comparison," *FEBS Lett*, vol. 509, no. 1, pp. 66–70, Nov. 2001.

[8] "Chlamydia pneumoniae TW-183, complete sequence," Mar. 2025, Accessed: May 21, 2025. [Online]. Available: http://www.ncbi.nlm.nih.gov/nuccore/NC_005043.1

[9] B. B. Matthews *et al.*, "Gene Model Annotations for Drosophila melanogaster: Impact of High-Throughput Data," *G3 Genes/Genomes/Genetics*, vol. 5, no. 8, pp. 1721–1736, 2015, doi: 10.1534/g3.115.018929.

[10] "boomphf - Rust." Accessed: May 21, 2025. [Online]. Available: https://docs.rs/boomphf/latest/boomphf/