Comparison of Performance Across Suffix Tries, Trees, and Arrays

Jason Hunter

March 13, 2025

1 Introduction

In my analysis, I implemented and ran comparisons on several suffix data structures - specifically suffix tries, suffix trees, and suffix arrays. I looked at both the performance runtime and memory usage of these data structures, with the goal of determining which suffix structure is the best choice for sequence processing tasks relevent to bioinformatics. I developed methods for building these structures and querying them. In my search for more efficient data structures, I came across an optimization which constructs suffix trees and suffix arrays in linear time, called Ukkonen's algorithm. I used this algorithm to make *Ukkonen's* suffix trees and arrays, and compared their performance to that of suffix tries, as well as to the naive suffix tree and array construction approaches. The experiments I ran showed that suffix arrays are the most efficient data structure in terms of runtime and memory usage, particularly at scale when the size of the input sequence is large. This is even more pronounced when the suffix array is constructed using Ukonnen's algorithm. Suffix tries were very memory intensive, and could only be used for small input sequences. Suffix trees were more memory efficient than suffix tries, but still used more memory than suffix arrays, especially when using the naive construction approach. Ukonnen's algorithm was able to construct suffix trees and suffix arrays in close to linear time, which was a significant improvement over the naive construction approach. Ultimately, I found that the suffix array constructed using Ukonnen's algorithm was the only data structure that could handle large input sequences larger than 100,000 efficiently, and is the best choice for sequence processing tasks in bioinformatics.

2 Results

I developed a benchmarking framework to measure the construction and query performance of suffix tries, suffix trees, and suffix arrays. This involved constructing and querying each structure with varying input sequence sizes while recording runtime and memory usage. The results are presented in the figures below.

2.1 Runtime Comparison

Figure 1 below displays a comparison of the runtime for constructing each of the suffix data structures, as it scales with the size of the input sequence. The red and green lines representing Ukonnen's suffix tree and suffix array construction algorithms respectively, show the best scaling with the size of the input sequence. The purple line represents the suffix trie, which is by far the slowest and most memory intensive to construct. My laptop ran out of memory when trying to construct a suffix trie for an input sequence larger than 10,000. My gaming PC with significantly more available memory was able to run the computations at a higher level, but for practical purposes, the suffix trie is not a viable option for large input sequences. It is clear that the suffix array constructed using Ukonnen's algorithm is the most efficient data structure in terms of construction time, and is the best choice for large input sequences, while the suffix trie is the least efficient and should be avoided for large input sequences.

Figure 2 below shows the runtime for querying each of the suffix data structures, as it scales with the size of the input sequence. In this case, the suffix array constructed using Ukonnen's algorithm is the most efficient data structure for querying, followed by the suffix array constructed using the naive approach. Its somewhat surprising that the suffix trie is more efficient here, but the graphs trend indicates that the suffix trie is likely to become less efficient if the input sequence size is increased, while the suffix arrays seem to remain more stable at scale.

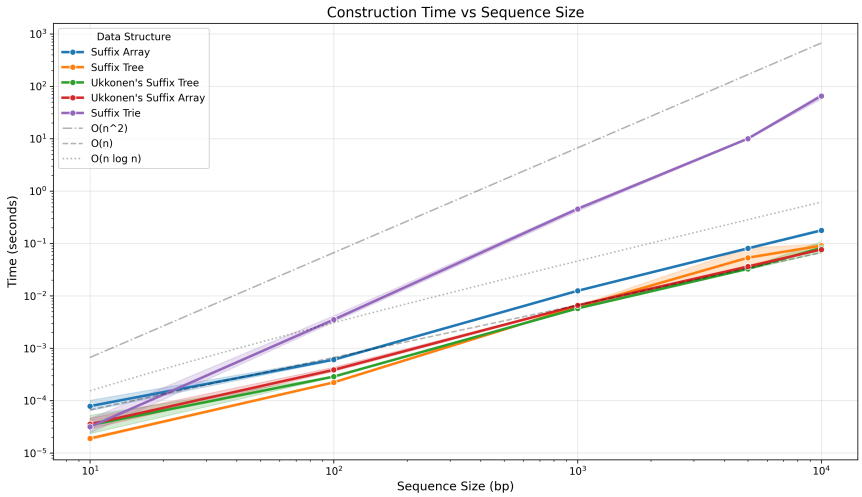

Figure 1: Runtime comparison of suffix tries, trees, and arrays for construction.

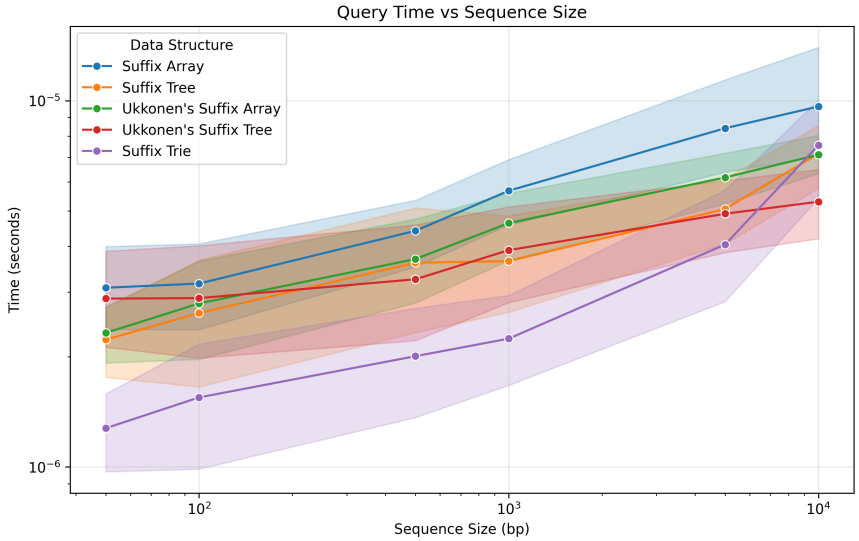

Figure 2: Runtime comparison of suffix tries, trees, and arrays for querying.

2.2 Memory Usage Comparison

Figure 3 shows the memory usage for constructing each of the suffix data structures, as it scales with the size of the input sequence. This is the most interesting graph, with a very clear trend that the suffix trie is the most memory intensive, and Ukonnen's suffix array and tree being the least memory intensive. As the size of the input sequence increases, the suffix trie becomes less and less viable, while Ukonnens suffix array and tree remain stable and efficient at scale, displaying a linear memory usage. The suffix array constructed with the modified BFS approach is more memory intensive than Ukonnen's suffix array, but still significantly more efficient than the suffix trie.

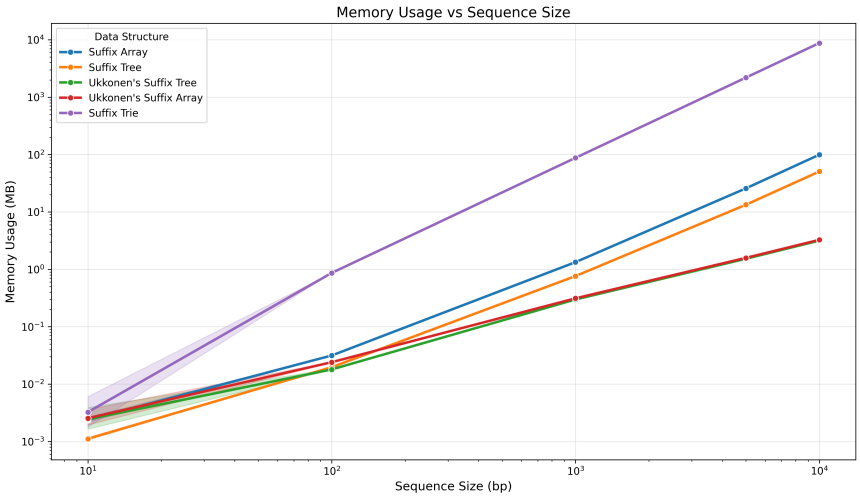

Figure 3: Memory usage comparison of suffix tries, trees, and arrays for construction.

3 Methods

3.1

3.2

3.3

3.4

4 Conclusion

In conclusion, I found that the suffix array is the most efficient data structure for sequence processing tasks in bioinformatics, particularly at scale.

5 Dependencies

- numpy
- subprocess
- psutil
- pandas
- seaborn
- matplotlib
- tqdm
- argparse
- gzip
- os
- glob
- GraphicViz

5.1 Installing GraphicViz

To install GraphicViz, the package used to generate the trees and tries in this document, run the following command:

```
$ pip3 install git+https://github.com/cjdrake/ipython-magic.git
```

6 Reproducibility

6.1 Commands to collect this data and reproduce the results

I ran these commands to generate and collect the data for the figures in this report.
NOTE: You may need to install some dependencies specified in the README.md file and section 4 of this document.
ALSO NOTE: These commands are shell scripts that can take up to 30 minutes to run, depending on how fast your CPU is.

```
$ git clone https://github.com/cu-compg-spring-2025/assignment-6-suffix-index-JasonHunter95.git
$ cd assignment-6-suffix-index-JasonHunter95
$ python3 src/benchmark.py
```

6.2 Commands to run the suffix trie, tree, and array visualizations

I ran these commands to generate the suffix trie, tree, and array visualizations in this document.
NOTE: You may need to install some dependencies specified in the README.md file and section 4 of this document.

```
$ python3 src/suffix_trie.py \
--string BANANA \
--query ANANA NANA ANA A NA $ BANANA$
```

```
$ python3 src/suffix_tree.py \
--string BANANA \
--query ANANA NANA ANA A NA $ BANANA$
```

```
$ python3 src/suffix_array.py \
--string BANANA \
--query ANANA NANA ANA A NA $ BANANA$
```

6.3 Commands to visualize a region of a FASTA file for the suffix structures

```
$ python3 src/suffix_trie.py \
--reference data/chr22.fa.gz \
--query ACGT \
--region 18800-18900
```

```
$ python3 src/suffix_tree.py \
--reference data/chr22.fa.gz \
--query ACGT \
--region 18800-18900
```

```
$ python3 src/suffix_array.py \
--reference data/chr22.fa.gz \
--query ACGT \
--region 18800-18900
```

```
$ python3 src/ukonnens_suffix_tree.py \
--reference data/chr22.fa.gz \
--query ACGT \
--region 18800-18900
```

```
$ python3 src/ukonnens_suffix_array.py \
--reference data/chr22.fa.gz \
--query ACGT \
--region 18800-18900
```

To reproduce these, clone the repository, ensure that you are in the root directory and run the above commmands. The data files are located in the data directory and the scripts are located in the src directory in the repository root.