Suffix Tree, Trie, and Array

Mathis Fituwi

May 7, 2025

1 Introduction

Suffix data structures are vital tools used in several computer science applications, such as data compression. In this paper, we will examine their use case in relation to DNA nucleotides and their variations: the suffix trie, suffix tree, and suffix array. A suffix trie is a digital tree that branches out into all possible suffix variations found in some arbitrary string T, where T in this case can represent a string of DNA nucleotides. A suffix tree is an extension of this structure that compresses the trie's auxiliary space from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ while maintaining its speed for query searches—i.e., finding a pattern within T. A suffix array is a simpler implementation that sorts all possible suffixes by their starting indices.

We will compare these three data structure below. There, we will see a distinct difference in runtime and memory usage: during the construction of these data structures, both in terms of memory and execution time, the suffix trie consumes the most resources. However, during query searches, we observe the suffix array doing the same.

1.1 Results

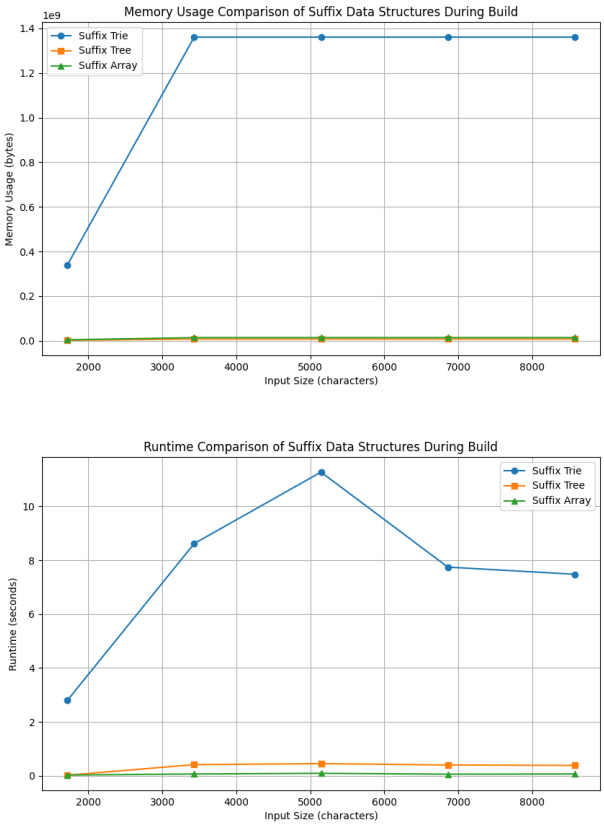

Figure 1: Runtime and memory usage during the suffix's build phase with increasing pattern size.

In Figures 1 and 2, we see that during the build time of the data structures, the suffix trie consumes the most resources in both runtime and memory. These measurements, as well as the figures below, were obtained by taking a subset of *wuhana-hu.fa* and testing it on increasingly larger segments. Consistently, we observe that the suffix trie performs the worst. This is expected, as the trie data structure has an average runtime and space complexity of $\mathcal{O}(n^2)$, whereas the suffix array takes $\mathcal{O}(n \log n)$ time to construct. The suffix tree, the more efficient version, takes $\mathcal{O}(n)$ time and space to construct.

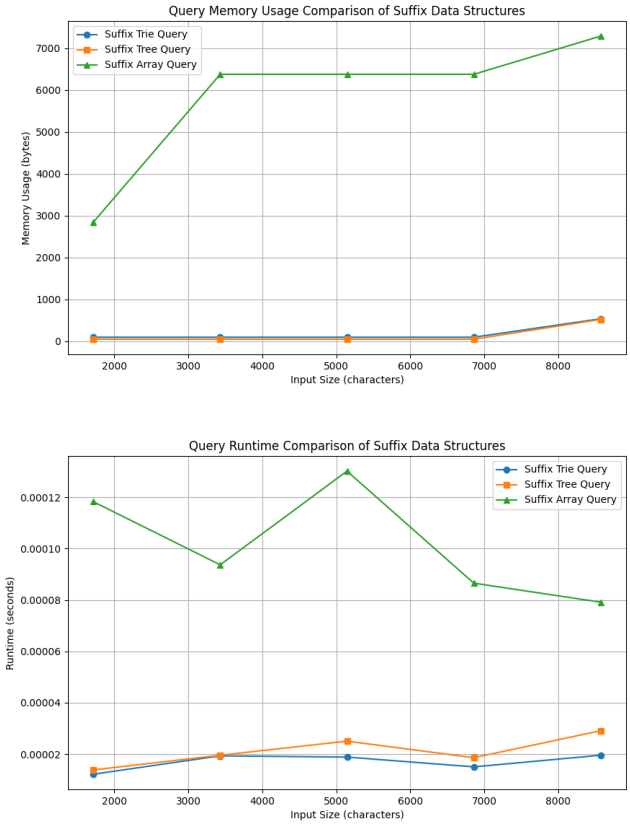

Figure 2: Runtime and memory usage during suffix's query search with increasing pattern size.

In Figures 2, we see that the suffix array performs the worst in query searches. In terms of runtime, the suffix array uses binary search, which results in a $\mathcal{O}(m + \log n)$ time complexity for exact matches where m is the pattern length. On the other hand, both the suffix trie and suffix tree should theoretically achieve $\mathcal{O}(m)$ query time, since they traverse the structure character by character. We also observe that the suffix tree keeps up with the suffix trie in terms of speed while using significantly less memory. Lastly, another interesting find is the excessive memory consumption of the suffix array. This may be due to additional temporary storage needed during binary search operations.

1.2 Method

The basic construction for the following data structure goes:

Suffix Trie Using Python's nested dictionaries, the algorithm iterates through each position in T, extracting a splice, and creating a path in the trie for each suffix. From my research, it is customary to append "$" as a flag to mark the end of a string suffix in the trie, which that was done. Each character becomes a key in the nested dictionary structure.

Suffix Tree Similarly, the algorithm splices segments of T and calls the method *add_suffix* to handle its construction.

Suffix Array Built by first constructing a suffix tree and then traversing it to extract all suffixes. Using DFS approach with a stack to explore the tree structure. For each node encountered, the algorithm accumulates the suffix string along the path and once a terminal node ("$") is found do we append it to the array. This array gets sorted such that then it's search can be done in a binary fashion.

1.3 Reproducibility

To replicate these experiments, follow these steps:

```
$ git clone git@github.com:cu-compg-spring-2025/assignment-6-suffix-index-MatFit.git
$ cd assignment-6-suffix-index-MatFit/src
$ python intermediate.py \
    --reference ../data/wuhana-hu.fa
    --query TCGATCGATCGA
```