Suffix String Alignment

Tavin Turner

May 7, 2025

1 Introduction

Seeing DNA sequence alignment as a string processing problem, we can see matching a pattern to some index in a sequence as equivalent to matching that pattern to the prefix of all of the string's suffixes. In so doing, we unravel a variety of suffix data structures to achieve sequence alignment – namely, suffix tries, trees, and arrays. Suffix tries and trees both index every suffix from the starting root connecting nodes by extending sequences – in tries, this sequence is one character, while trees use the substring until another divergence in the suffixes. Suffix arrays, on the other hand, sort the indices that a suffix begins at in alphabetical order of those suffixes, enabling a binary search procedure. Theoretical statistics for these data structures can be found in Table 1.

Model	Gen. Runtime	Gen. Memory	Size	Align. Runtime	Align. Memory
Suffix trie	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(m)$	$O(1)$
Suffix tree	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(m)$	$O(1)$
Suffix array	$O(n \log n)$	$O(n^2)$	$O(n)$	$O(\log n)$	$O(1)$

Table 1: Theoretical statistics of suffix string alignment data structures.

2 Results

In line with the theoretical expectations, suffix trie took significantly more time than suffix array to generate, although the performance difference of single-character edges is apparent even in construction, where the otherwise identical suffix tree has such dramatically lower construction run time that it is invisible when charted against the suffix trie (Figure 1). Unexpectedly, both suffix trie and suffix tree appear significantly memory-efficient in practical construction than the suffix array, likely since the suffix array explicitly stores every suffix during sorting before discarding later.

With increasing query size, the suffix tree and suffix array appear roughly equivalent in performance over the experimental query range, although the suffix tree empirically shows the theoretical runtime increase with query size (Figure 2). The suffix trie, however, takes up to ten times as long as the other approaches, with less consistent trends of performance over query size. Interestingly, although all of the algorithms show their $O(1)$ theoretical alignment space (Table 1), the suffix array take significantly more memory overall (Figure 2), since it has to index the string separately for each character, which is not stored with the array.

3 Methods

3.1 Suffix trie

The suffix trie is constructed from a text T by taking each suffix $S = T[i:]$ in increasing length and searching the existing trie for each character iteratively until a branch does not exist fo the suffix, which it then creates new successive branches for each remaining character afterward. Searching the suffix trie is a simple traversal of the trie by character.

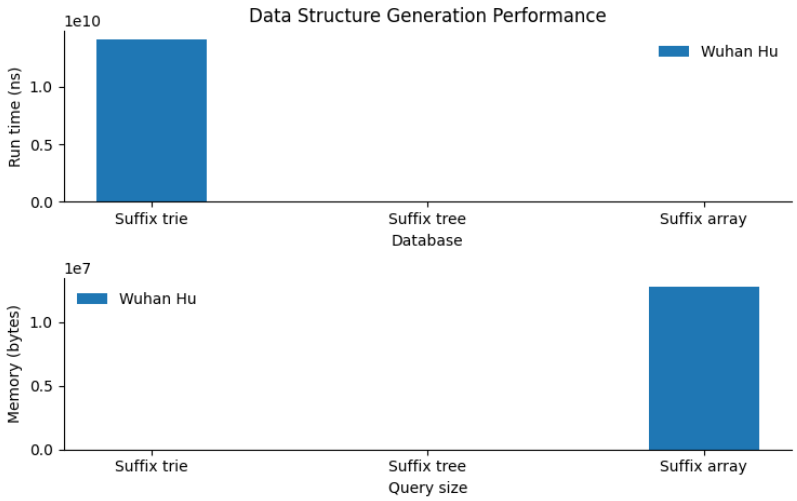

Figure 1: The empirical runtime and memory usage of generating suffix tries, trees, and arrays for the first 5000 characters of the Wuhan Hu dataset.

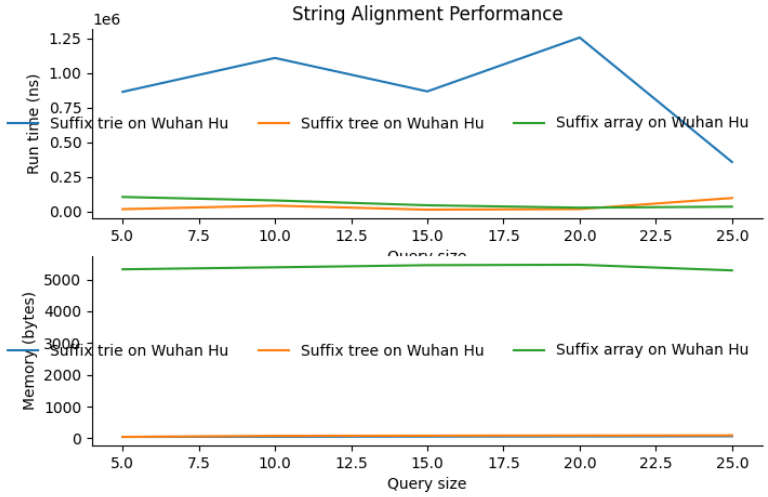

Figure 2: The empirical runtime and memory usage of aligning strings of various lengths to the first 5000 characters of the Wuhan Hu dataset with suffix tries, trees, and arrays.

3.2 Suffix tree

Like the suffix trie, the suffix tree is constructed from a text T by taking each suffix S in increasing length and searching the tree, but instead searches through the suffix left-to-right with each edge, which may be a substring, until a misalignment is found or a leaf is reached, at which point the tree truncates the preceding branch at the misalignment and gives it two children: the remainder of the existing branch and the new remaining substring. Searching the suffix tree takes the same method as its construction.

3.3 Suffix array

The suffix array is constructed by sorting the indices of each suffix by the suffix in alphabetical order. To search the array, a binary search for the lowest index before a lower-order suffix than the pattern is found, then a binary search for the highest index before a higher-order suffix than the pattern is found.

3.4 Empirical comparison

We evaluated the performance of the suffix string search algorithms considering five pattern sizes of 5, 10, 15, 20, and 25 on the first 5000 characters of the Wuhan Hu dataset. The performance metrics include runtime and memory usage. For each text size, we ran a single search where we generated a random string for T from the alphabet A, C, T, G and extracted a random substring P from T. We then recorded the runtime and memory usage of the algorithm consider that P. This was repeated for each text and pattern size five times. After the round was complete for a given text size, we calculated the average runtime and memory usage for the search.

3.5 Reproducibility

To replicate these experiments, clone the repository and then run the following commands from the root directory of the repository.

```
$ git clone https://github.com/cu-compg-spring-2025/assignment-6-suffix-index-itsTurner.git \
            suffix_index
$ cd suffix_index
$ pip install -r requirements.txt
$ python src/evaluation.py \
    -r data/wuhana-hu.fa.gz \
    --query_size 5 30 5 \
    --queries_per_size 5 \
    --out_file doc/results/wuhana_q5-30-5_n5.png
```