Burrow Wheel Index and Wavelet Trees

Mathis Fituwi

May 7, 2025

1 Introduction

The Burrows–Wheeler Transform (BWT) is a data transformation algorithm widely used for compression purposes, one of its most notable uses being in the *bzip2* file compressor. In this paper, we examine its application in a succinct data structure, the wavelet tree (WT). We analyze the performance of query searching in a WT built over the BWT output and compare it to the suffix array data structure as a baseline.

The Burrows-Wheeler transform groups similar characters by rearranging the input string's cyclic rotations. Given a string s of length n, we will generate all n cyclic rotations of s, sort them lexicographically, store it in a matrix and then read off the last column. This string of 'last column' tends to cluster identical or similar characters together, but not always. In both the best and worst cases, the transformation runs in $O(n)$ time. To reverse the transform and reconstruct s, we use *LF-mapping*: by comparing the last column sorted with the first column of the matrix, we recover the original string in $O(n)$ time.

For a quick overview of two related data structures, the wavelet tree and the suffix array, we note the following. A wavelet tree, constructed over the BWT output, recursively splits the alphabet into two halves (binary partition), like merge-sort would, and builds a balanced binary tree. Building the tree takes $O(n \log \sigma)$ time (where σ is the alphabet size), and operations such as `access`, `rank`, and `select` each run in $O(\log \sigma)$ time. A suffix array is an array of integers that gives the starting positions of all suffixes of s, lexicographically ordered. It can be built in $O(n)$ time with advanced algorithms and supports pattern searches of length m in $O(m \log n)$ time via binary search.

Our findings indicate that the BWT index with wavelet tree support dramatically outperforms the suffix array in query runtime, at the expense of increased memory consumption.

1.1 Results

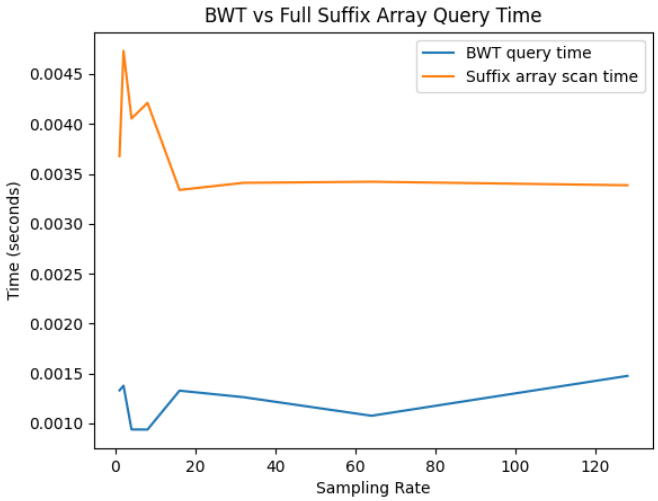

Figure 1: Query runtime for "ACTGACTG" on the Wuhan virus genome

As shown in Figure 1, the BWT combined with a WT supports pattern search with respect to the query length m. In the suffix array, the scan must examine each of the n suffixes and compare up to m characters, giving $O(nm)$ time per query. In contrast, with the backward search algorithm in the BWT, it performs m WT rank operations in constant time, with an overall $O(m)$ complexity.

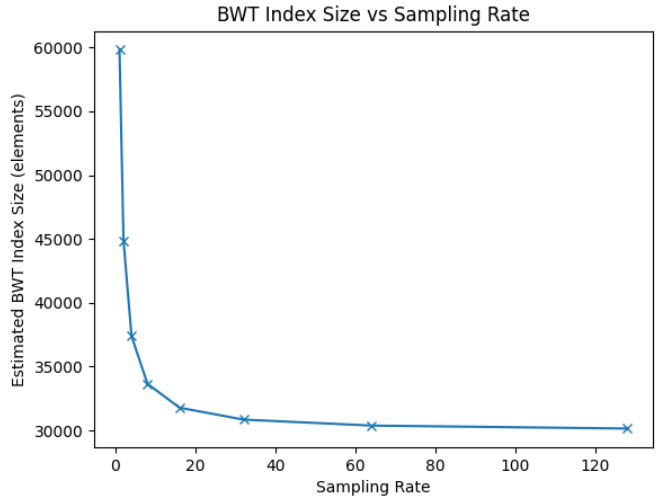

Figure 2: BWT memory consumption for "ACTGACTG" on the Wuhan virus genome

As shown in Figure 2, the BWT index consumes more memory—storing the BWT string itself, a skip list for backward searching, and a sampled suffix array—and thus peaks at about 600,000 elements when the sampling rate is small, then converges to around 300,000 as the sampling rate increases.

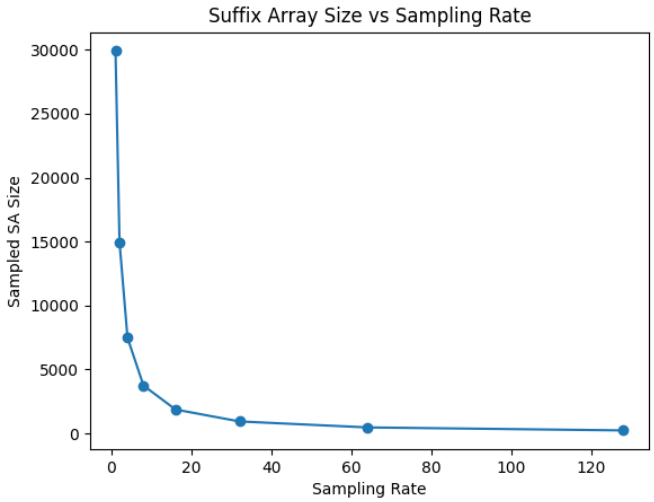

Figure 3: Suffix array memory consumption for "ACTGACTG" on the Wuhan virus genome

As shown in Figure 3, the suffix array size fares much better in memory consumption, since it only needs to store sampled suffix positions. Its size drops to almost 0 because by keeping just one suffix position every r characters such that the total count, $\lceil n/r \rceil$, falls rapidly as r increases.

1.2 Method

We implemented two string-search approaches—a BWT index with wavelet-tree support and a suffix array scan—and compared them over varying sampling rates. First, each input text was appended with a unique end marker "$" and transformed via the Burrows–Wheeler Transform to produce the last-column string L. We then built a skip list that records, for each character, its first row in the sorted rotations. Using the open-source wavelet-tree library provided to us, we indexed L into a bit-vector tree, storing both the tree nodes and the character-to-bit-path codes.

For several sampling rates $r \in \{1, 2, 4, \ldots, 128\}$, we constructed a sampled suffix array by retaining every rth suffix position, measured the backward-search query on the BWT index and the direct scan that checks every suffix for the query via character-by-character comparison. From there, we recorded the query times and the number of stored suffix positions.

1.3 Reproducibility

To replicate these experiments, follow these steps:

```
$ git clone git@github.com:cu-compg-spring-2025/assignment-7-burrows-wheeler-index-MatFit.git
$ cd assignment-7-burrows-wheeler-index-MatFit
$ python3.10 src/bwt.py \
        --q ACTGACTG \
        --f data/wuhana-hu.fa.gz
```