HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI
MATEMAATTIS-LUONNONTIETEELLINEN TIEDEKUNTA
MATEMATISK-NATURVETENSKAPLIGA FAKULTETEN
FACULTY OF SCIENCE

String Sorting in Python – Comparison of Several Algorithms

Onni Koskinen, Arturs Polis, and Lari Rasku

All algorithms were written from scratch, striving for idiomatic and easily understandable Python code over low-level or implementation-specific optimizations whenever possible. Empirical measurements on the performance of these algorithms were made. Here we try to explore the relative performance of different algorithms and analyze the reasons behind strengths and weaknesses of the algorithms used.

BURROWS–WHEELER TRANSFORM

The Burrows–Wheeler transform (BWT) is an invertible text transform defined as follows.

Input: text T = BANANA#

1. Build a matrix with the text *rotations* as rows

$$\begin{array}{ccccccc} B & A & N & A & N & A & \# \\ A & N & A & N & A & \# & B \\ N & A & N & A & \# & B & A \\ A & N & A & \# & B & A & N \\ N & A & \# & B & A & N & A \\ A & \# & B & A & N & A & N \\ \# & B & A & N & A & N & A \end{array}$$

2. Sort the rows

$$\begin{array}{c|ccccc|c} F & & & & & & L \\ \# & B & A & N & A & N & A \\ A & \# & B & A & N & A & N \\ A & N & A & \# & B & A & N \\ A & N & A & N & A & \# & B \\ B & A & N & A & N & A & \# \\ N & A & \# & B & A & N & A \\ N & A & N & A & \# & B & A \end{array}$$

Output: BWT L = ANNB#AA (the last column)

The properties of the BWT make it easier to compress than the original text. It is used as the first stage in many compression programs including the widely used bzip2 (thus the b).

INVERSE BWT

Define $\text{RANK}(j) = \left|\{i \mid i < j \text{ and } L[i] = L[j]\}\right|$. The BWT can be inverted as follows.

Input: BWT L = ANNB#AA

1. Compute C and RANK arrays by scanning L

2. Starting at $L[i]$ = #, follow the permutation:

$$i \mapsto C[L[i]] + \text{RANK}(i)$$

Output $L[i]$ at each step

Output: reverse text T^R = #ANANAB

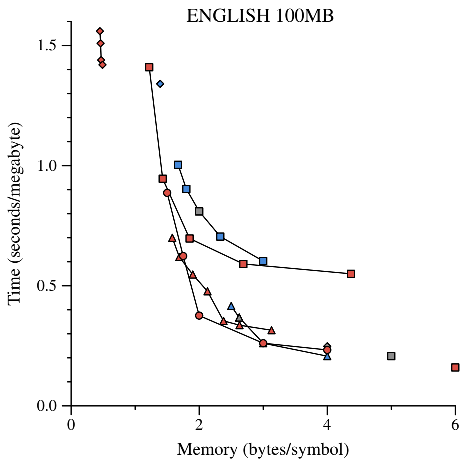
EXPERIMENTAL RESULTS

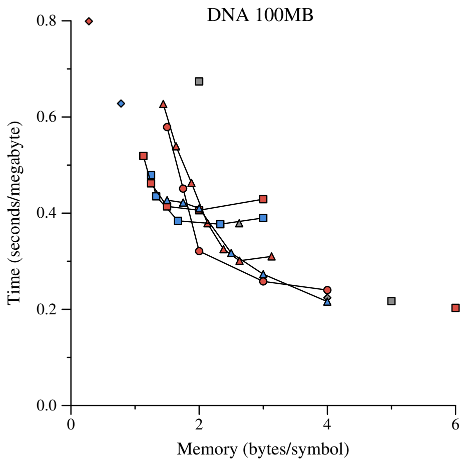
The graphs below show the time and space requirements of several algorithms on two texts. The algorithms are divided into three groups:

New algorithms based on reference point ranks, repetition shortcuts and wavelet trees

Improved implementations of wavelet trees and algorithms from [?]

Prior algorithms from [?, ?]





NEW ALGORITHMS FOR INVERSE BWT

The basic inversion algorithm described above has linear time and space complexity, but it still dominates the time and space requirements during decompression in programs like bzip2.

It is slow because each memory access during the permutation traversal is essentially random causing many cache misses.

It needs a lot of space for the RANK array:

$$|\text{RANK}| = n \log n \text{ bits} = 4n \text{ bytes}$$
$$|\text{text}| = n \log \sigma \text{ bits} = n \text{ bytes}$$

where n = text length and σ = alphabet size.

REFERENCE POINT RANKS

We reduce space by storing ranks relative to reference points, which can be placed in two ways:

Every kth position [?]   Every kth occurrence [new]

A new improvement is to use variable length encoding, where a frequent symbol uses less bits for symbol and more bits for rank.

Finally, we can trade time for space by replacing RANK with scanning from the nearest reference point [?].

REPETITION SHORTCUTS

Repetitions in the text manifest as *pairs of parallel paths* (PPP) in the inverse BWT permutation. We use this as follows.

1. On the first pass, observe the PPP (due to repeated ANA)

2. Replace the other path by shortcut and follow it on the second pass

testing here

The shortcuts reduce the number of cache misses. This is the *fastest* known algorithm.

WAVELET TREES

Wavelet tree is a text representation that can be both compressed and preprocessed for rank queries with little additional space. They are used with compressed text indexes [?] to answer *general rank queries*:

$$\text{RANK}_c(j) = \left|\{i \mid i < j \text{ and } L[i] = c\}\right|.$$

We need wavelet trees for *special rank queries*:

$$\text{RANK}(j) = \text{RANK}_{L[j]}(j).$$

We use our own wavelet tree implementations optimized for special rank queries.

We combine wavelet trees with reference point ranks, obtaining the *most space-efficient* algorithm.