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.

DATA SET

The timing test data consisted of the PROTEINS, DNA and ENGLISH datasets from the Pizza&Chili Corpus, in addition to a set of URLs from Ranjan Sinha's ref1 data ref2 for his original Burstsort paper.

A 100MB and a 200MB sample of each dataset was used. The ENGLISH datasets were not used as-is, but with each word split on its own line, in order to make the algorithms sort individual words and not entire lines. The statistics file documents some stringological properties of these datasets.

ref1 https://sites.google.com/site/ranjansinha/home
ref2 http://www.cs.mu.oz.au/ rsinha/resources/data/sort.data.zip

Dataset	Number of strings	Sum of lengths	Max string length	alphabet size	Sum of LCP array
dna.100MB	618	104856983	3732300	15	4501
dna.200MB	1114	209714087	3732300	15	8948
proteins.100MB	359505	104498096	36805	24	18853436
proteins.200MB	709116	209006085	36805	24	50076184
urls.100MB	3284368	101569109	372	114	94113004
urls.200MB	6576059	203139142	560	114	191545831
words.100MB	18502734	85200064	112	211	83643408
words.200MB	37003241	170395992	112	220	168115390

ALGORITHMS

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*:

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

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.

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 [?, ?]