Technical Report

iZENElib Data Compression

Ian Yang

March 25, 2009

iZENEsoft(Shanghai) Co., Ltd

Abstract

This is a report for the sub project data compression in *iZENElib*. Project data compression focuses on the search of large dictionaries if the data contained can be compressed.

Contents

1 Introduction

The world is drowning in data. Classic algorithms are greedy in terms of their space usage and often cannot perform computations on all of the data. Nowadays, moderate scale information retrieval system must handle huge scale date. The entire dataset can be compressed in such occasion, but many problems still require the compressed dataset to be queried quickly.

The paper *An algorithmic framework for compression and text indexing*[1] presented a unified algorithmic framework to obtain nearly optimal space bounds for text compression and compressed text indexing, apart from lower-order terms. The authors provide a new tight analysis of text compression based on the Burrows-Wheeler transform (BWT). They also provide a new implementation of compressed text indexing based on the compressed suffix array (CSA). A key point of the unified approach is the use of the finite set model instead of the empirical probability model. This new perspective reveals an alternative way to model an arbitrary partition of the bwt. The encoding adopts an algorithm that stores t items out of a universe of size n in the information theoretic minimum space $\lg \binom{n}{t}$ bits (since there are $\binom{n}{t}$ subsets of t items out of n).

This project intends to provide an implementation of the unified framework, as well as the applications to compressed suffix array and text indexing. Chapter 2 briefly introduces the algorithms and data structures used in the framework. Chapter 3 represents the design outline.

2 Algorithm and Data Structures

2.1 The BWT

Loosely speaking, the BWT produces a permutation $\texttt{BWT}(T)$ of the input text T such that from $\texttt{BWT}(T)$ we can retrieve T but at the same time $\texttt{BWT}(T)$ is much easier to compress.

The BWT is a very powerful tool and even the simplest algorithms that use it have surprisingly good performances. More advanced BWT-based compressors, such as `bzip` and `szip`, are among the best compressors currently available.

Another remarkable property of the BWT is that it can be used to build a data structure which is a sort of compressed suffix array for the input text T. Such data structure consists of a compressed version of $\texttt{BWT}(T)$ plus $o(|T|)$ bits of auxiliary information. This data structure can be used to compute the number of occurrences of an arbitrary pattern p in T in $O(|p|)$ time, and compute the position in T of each one of such occurrences in $O(\log^{\epsilon}|T|)$ time.

The BWT consists of a reversible transformation of the input string T. The transformed string denoted by $\texttt{BWT}(T)$, contains the same characters as T but it is usually easier to compress. The string $\texttt{BWT}(T)$ is obtained as follows (see Table 2.1). First, a special character # larger than any other character is appended to end of T. Then, the cyclic shifts of $T\#$ are sorted in lexical order. The output $\texttt{BWT}(T)$ is the last column of the sorted matrix. In Table 2.1, $\texttt{BWT}(T) = \texttt{ssmp\#pissiii}$.

Clearly, the BWT is related to suffix sorting, since the comparison of any two circular shifts must stop when the end marker # is encountered. The corresponding suffix array is a simple way to store the sorted suffix (see column '$SuffixArray$' in Table 2.1).

TODO: Introduce Context Partitions of BWT

TODO: Introduce Wavelet Tree

TODO: Introduce how to implement CSA

TODO: Introduce how to use the framework in text indexing

Original	*Sorted*			*Suffix Array*
Q	F		L	
mississippi#	i	ppi#missis	s	ippi#
#mississippi	i	ssippi#mis	s	issippi#
i#mississipp	i	ssissippi#	m	ississippi#
pi#mississip	i	#mississip	p	i#
ppi#mississi	m	ississippi	#	mississippi#
ippi#mississ	p	i#mississi	p	pi#
sippi#missis	p	pi#mississ	i	ppi#
ssippi#missi	s	ippi#missi	s	sippi#
issippi#miss	s	issippi#mi	s	sissippi#
sissippi#mis	s	sippi#miss	i	ssippi#
ssissippi#mi	s	sissippi#m	i	ssissippi#
ississippi#m	#	mississipp	i	#

Table 2.1: Matrix Q for the BWT containing the cyclic shifts of text $T = \texttt{mississippi}\#$ (column '*Original*'). Sorting of the rows of Q, in which the first (F) and last (L) symbols in each row are separated (column '*Sorted*'). Suffix array SA for T (column '$SuffixArray$').

3 Design

3.1 Filter Stream

Concept stream is widely used in IO operations. A stream is an abstraction that represents a device on which input and ouput operations are performed. A stream can basically be represented as a source or destination of characters of indefinite length. Clients may read data from stream (input stream), or/and write data into stream (output stream).

The compression and decompression can be viewed as stream `Filter`s. `Filter`s are used to modified character sequences. For example, somebody might use a filter to replace all instances of one word with another, to convert all alphabetic characters to lower case or to compress, decompress a document.

`Filter`s can compose a chain (or pipeline), and data are transferred from one to another. There exists two basic categories of `Filter`s, which are named `InputFilter`s and `OutputFilter`s. `InputFilter`s represent a "pull" model of filtering: a source of unfiltered data is provided, and the `Filter` is expected to generate a certain number characters of the filtered sequence. The filtered sequence is generated incrementally, meaning that to filter a given character sequence the Filter typically must be invoked several times. `OutputFilter`s represent a "push" model of filtering: a sequence of unfiltered characters and a sink are provided, and the `Filter` is expected to filter the characters and write them to the sink. Like `InputFilter`s, `OutputFilter`s also process data incrementally.

TODO: Diagram to demonstrates filters

3.2 The BWT

TODO: design for BWT

TODO: Diagram of design for BWT

3.3 The Wavelet Tree

TODO: design for Wavelet Tree

TODO: Diagram of design for Wavelet Tree

Appendix

A Schedule

Table A.1: *iZENElib* Data Compression Schedule

Milestone	Start	End	In Charge	Description	Status
1 Read the existing tr of *iZENElib*	2009-03-05	2009-03-05	Ian	Read tr to understand the objectives of *iZENElib* and the architecture, interface conventions, and *etc.*	Finished
2 Go though existing source codes of *iZENElib*	2009-03-06	2009-03-06	Ian	Go though codes to get familiar with the coding style and the layout of the whole project.	Finished
3 Requirements Analysis	2009-03-09	2009-03-10	Ian	Background research and confirm the objectives and outcomes of this project.	Finished
4 Research	2009-03-11	2009-03-17	Ian	Survey existing solutions, papers as reference and have a brainstorm.	Finished
5 Design	2009-03-18	2009-03-24	Ian	Architecture design. Detailed design for core components. Describe the design in TR.	Ongoing
6 Implementation	2009-03-25	2009-03-27	Ian	Implement all functionalities.	Not Started
7 Test	2009-03-30	2009-04-1	Ian	Unit test and integrated test.	Not Started
8 Report	2009-04-02	2009-04-03	Ian	Make conclusion and refine TR.	Not Started

Bibliography

[1] R. Grossi, A. Gupta, and J.S. Vitter. An algorithmic framework for compression and text indexing. *submitted for publication*, 2007.

Index