Compressing and Indexing Strings and (labeled) Trees

Paolo Ferragina
Dipartimento di Informatica, Università di Pisa

Two types of data

• String = raw sequence of symbols from an alphabet \( \Sigma \)
  » Texts
  » DNA sequences
  » Executables
  » Audio files
  » ...

• Labeled tree = tree of arbitrary shape and depth whose nodes are labeled with strings drawn from an alphabet \( \Sigma \)
  » XML files
  » Parse trees
  » Tries and Suffix Trees
  » Compiler intermediate representations
  » Execution traces
  » ...
What do we mean by “Indexing”?  

- **Word-based indexes**, here a notion of “word” must be devised!  
  - Inverted files, Signature files, Bitmaps.
  
- **Full-text indexes**, no constraint on text and queries!  
  - Suffix Array, Suffix tree, ...

- **Path indexes** that also support navigational operations!  
  - see next...

**Substring searches**  
**String statistics**  
**Motif extraction**  

**Path indexes that also support navigational operations!**  
- i-th child with some label constraint  
  - Parent, or ancestor  
  - Labeled path anchored anywhere  

**Subset of XPath [W3C]**

What do we mean by “Compression”?  

- Data compression has two positive effects:  
  - Space saving (or, enlarge memory at the same cost)  
  - Performance improvement  
  - Better use of memory levels closer to CPU  
  - Increased network, disk and memory bandwidth  
  - Reduced (mechanical) seek time

**Folk tale:**  
More economical to store data in compressed form than uncompressed
Toward Ubiquitous Compression

Fred Dougliis
IBM T.J. Watson Research Center

August 2004 | DIMACS BWT Workshop

Conclusions

**Systems should automatically compress data whenever the benefits of storing or transmitting the compressed data outweigh the costs**

- It’s time to “teach” systems how to do this
Study the interplay of Compression and Indexing

Do we witness a paradoxical situation?

- An index injects redundant data, in order to speed up the pattern searches
- Compression removes redundancy, in order to squeeze the space occupancy

NO, new results proved a mutual reinforcement behaviour!

- Better indexes can be designed by exploiting compression techniques
- Better compressors can be designed by exploiting indexing techniques

More surprisingly, strings and labeled trees are closer than expected!

- Labeled-tree compression can be reduced to string compression
- Labeled-tree indexing can be reduced to “special” string indexing problems
Our journey over “string data”

Index design (Weiner ‘73)  
Suffix Array ‘87 and ‘90

Compressed Index  
Space close to gzip, bzip
Query time close to P’s length  
(Ferragina-Manzini, FOCS ’00 + JACM ’05)

Compressor design (Shannon ‘48)  
Burrows-Wheeler Transform (1994)

Compression Booster  
A combinatorial tool to transform poor compressors into better compressors  
(Ferragina, Giancarlo, Manzini, Sciortino, JACM ’05)

Improved indexes and compressors for strings  
[Ferragina-Manzini-Makinen-Navarro, ’04]  
And many other papers of many other authors...

The Suffix Array [BaezaYates-Gonnet, 87 and Manber-Myers, 90]

\[ T = \text{mississippi}\# \]

**Suffix permutation cannot be any of \{1, ..., N\}**  
\# binary texts = \( 2^N \times N! \) = \# permutations on \( 1, 2, ..., N \)  
\( \Omega(N) \) bits is the worst-case lower bound  
\( \Omega(N H(T)) \) bits for compressible texts  
Several papers on characterizing the SA’s permutation  
[Düvel et al. ’06; Hamai et al. ’05; Nassim et al. ’05; Stoye et al. ’05]
Can we compress the Suffix Array? [Ferragina-Manzini, JACM '05]

The FM-index is a data structure that mixes the best of:
- Suffix array data structure
- Burrows-Wheeler Transform

The theoretical result:
- Query complexity: $O(p + \text{occ} \log^c N)$ time
- Space occupancy: $O(N H_k(T) \log N)$ bits $\rightarrow o(N)$ if $T$ compressible

The corollary is that:
- The Suffix Array is compressible
- It is a self-index

**New concept**: The FM-index is an opportunistic data structure that takes advantage of repetitiveness in the input data to achieve compressed space occupancy, and still efficient query performance.

The Burrows-Wheeler Transform (1994)

Let us given a text $T = \text{mississippi}#$

Sort the rows

Let $F$ be the sorted version of $T$.

Paolo Ferragina, Università di Pisa
Why $L$ is so interesting for compression?

A key observation:
- $L$ is locally homogeneous
  \[ \Rightarrow \text{$L$ is highly compressible} \]

Algorithm Bzip:
1. Move-to-Front coding of $L$
2. Run-Length coding
3. Statistical coder: Arithmetic, Huffman

Bzip vs. Gzip: 20% vs. 33% compression ratio! [Some theory behind: Manzini, JACM ‘01]

Building the BWT = SA construction
Inverting the BWT = array visit
...overall $\Theta(N)$ time, but slower than gzip...

L is helpful for full-text searching?

L includes SA and T. Can we search within L?

Paolo Ferragina, Università di Pisa
A useful tool: \( L \to F \) mapping

How do we map \( L \)'s onto \( F \)'s chars?

... Need to distinguish equal chars...

\[
\text{occ}("i",11) = 3
\]

To implement the \( LF \)-mapping we need an oracle \( \text{occ}(\ 'c'\ ,j) = \text{Rank of char } c \text{ in } L[1,j] \)

Substring search in \( T \) (Count the pattern occurrences)

Inductive step: Given \( fr,lr \) for \( P[j+1,p] \)

1. Take \( c=P[j] \)
2. Find the first \( c \) in \( L[fr, lr] \)
3. Find the last \( c \) in \( L[fr, lr] \)
4. \( L \)-to-\( F \) mapping of these chars

\( \text{Occ()} \) oracle is enough

(i.e. Rank/Select primitives over \( L \))
Many details are missing...

😊 The column L is actually kept compressed:
  - Still guarantee $O(p)$ time to count the P’s occurrences
  - Achieves logarithmic-time to locate each pattern occurrence

😊 FM-index + LZ78 parsing
  - Achieves $O(p \cdot \text{occ})$ time
  - but it loses a sub-logarithmic factor in the front of $H_k$

😊 What about a large $\Sigma$:
  - Wavelet Tree and variations [Grossi et al., Soda 03; F.M.-Makinen-Navarro, Spire 04]
  - New approaches to Rank/Select primitives [Munro et al. Soda 06]

😊 Efficient and succinct index construction [Hon et al., Focs 03]
  - In practice, Lightweight Algorithms: $(5+\varepsilon)N$ bytes of space
  - Achieves $O(p + \text{occ})$ time...

FM-index + LZ78 parsing [see also LZ-index by Navarro]

Efficient and succinct index construction [Hon et al., Focs 03]

Five years of history...

Look at the survey by Gonzalo Navarro and Veli Makinen
What’s next?

**Interesting issues:**

- What about large $\Sigma$: fast Rank/Select in entropy-space bounds?  
  [Sadakane et al., Soda 06; Munro et al. Soda 06]

- What about disk-aware or cache-oblivious versions?  
  [Brodal et al., Soda 06]

- Applications to show that this is a technological breakthrough...

What about their practicality?
Is this a technological breakthrough?

How to turn these challenging and mature theoretical achievements into a technological breakthrough?

- Engineered implementations
- Flexible API to allow reuse and development
- Framework for extensive testing
Joint effort of Navarro’s group and mine, hence two mirrors

The best known indexes will have been implemented!! A carefully designed API which build, count, locate, extract,...

A group of var Some tools have been designed to automatically plan, execute and check the index performance over the text collections.

Where we are...

- **Data type**: Text
- **Indexing**:
  - Suffix Tree
  - Suffix Array
  - Powerful but space consuming
- **Compressed Indexing**:
  - BWT + rank/select
  - Powerful and compact

Paolo Ferragina, Università di Pisa
Why we care about labeled trees?

An XML excerpt

```xml
<dblp>
  <book>
    <author>Donald E. Knuth</author>
    <title>The TeXbook</title>
    <publisher>Addison-Wesley</publisher>
    <year>1986</year>
  </book>
  <article>
    <author>Donald E. Knuth</author>
    <author>Ronald W. Moore</author>
    <title>An Analysis of Alpha-Beta Pruning</title>
    <pages>293-326</pages>
    <year>1975</year>
    <volume>6</volume>
    <journal>Artificial Intelligence</journal>
  </article>
</dblp>

Paolo Ferragina, Università di Pisa
A tree interpretation...

XML document exploration = Tree navigation
XML document search = Labeled subpath searches

Our problem

Consider a rooted, ordered, static tree $T$ of arbitrary shape, whose nodes are labeled with symbols from an alphabet $\Sigma$.

We wish to devise a succinct representation for $T$ that efficiently supports some operations over $T$'s structure:

- Navigational operations: parent($u$), child($u$, $i$), child($u$, $i$, $c$)
- Subpath searches over a sequence of $k$ labels

Seminal work by Jacobson [Focs '90] dealt with binary unlabeled trees, achieving $O(1)$ time per navigational operation and $2t + o(t)$ bits.

Munro-Raman [Focs '97], then many others, extended to unlabeled trees of arbitrary degree and a richer set of navigational ops: subtree size, ancestor,...

Geary et al [Soda '04] were the first to deal with labeled trees and navigational operations, but the space is $\Theta(t |\Sigma|)$ bits.

Yet, subpath searches are unexplored
We propose the XBW-transform that mimics on trees the nice structural properties of the BW-transform on strings.

The XBW-transform linearizes the tree $T$ in such a way that:

- the indexing of $T$ reduces to implement simple rank/select operations over a string of symbols from $\Sigma$.
- the compression of $T$ reduces to use any $k$-th order entropy compressor ($gzip$, $bzip$, ...) over a string of symbols from $\Sigma$.

---

**The XBW-Transform**

1. Visit the tree in pre-order. For each node, write down its label and the labels on its upward path.

Paolo Ferragina, Università di Pisa
The XBW-Transform

Step 2. Stably sort according to $S_\pi$

Key facts
- Nodes correspond to items in $<S_{\text{last}}, S_\pi>$.
- Node numbering has useful properties for compression and indexing.

XBW can be built and inverted in $\text{optimal}/O(t)$ time.

XBW takes $\text{optimal}/t \log |\Sigma| + 2t$ bits.
The XBW-Transform is highly compressible

- $S_{\text{last}}$ is locally homogeneous (like BWT for strings)
- $S_{\text{aux}}$ has some structure (because of T9's structure)

XML Compression: XBW + PPMdi!

String compressors are not so bad!?
Structural properties of XBW

Properties:
- Relative order among nodes having same leading path reflects the pre-order visit of T
- Children are contiguous in XBW (delimited by 1s)
- Children reflect the order of their parents

The XBW is searchable

XBW indexing [reduction to string indexing]:
- Store succinct and efficient Rank and Select data structures over these three arrays

Paolo Ferragina, Università di Pisa
Subpath search in XBW

Inductive step:
1. Pick the next char in $\Pi[i+1]$, i.e., 'D'
2. Search for the first and last 'D' in $S_\alpha[fr,lr]$ of $S_\pi$
3. Jump to their children

Paolo Ferragina, Università di Pisa
XML Compressed Indexing

What about XPress and XGrind?
- XPress ~ 30% (dblp 50%), XGrind ~ 50% → no software running

In summary [Ferragina et al, Focs ’05]

- The XBW-transform takes optimal space: $2t + t \log |\Sigma|$, and can be computed in optimal linear time.

- We can compress and index the XBW-transform so that:
  - its space occupancy is the optimal $t H_0(T) + 2t + o(t)$ bits
  - navigational operations take $O(\log |\Sigma|)$ time
  - subpath searches take $O(p \log |\Sigma|)$ time

- It is possible to extend these ideas to other XPath queries, like:
  - `//path[text()="substring"]`
  - `//path1//path2`
  - ...

New bread for Rank/Select people!!

If $|\Sigma| = \text{polylog}(t)$, no log$|\Sigma|$-factor
(loglog $|\Sigma|$ for general $\Sigma$ [Munro et al, Soda 06])
The overall picture on Compressed Indexing...

Data type
- Text
- Labeled Tree

Indexing
- Suffix Tree
- Suffix Array
- Powerful but space consuming

Compressed Indexing
- BWT + rank/select
- Strong connection
- XBW + rank/select

This is a powerful paradigm to design compressed indexes for both strings and labeled trees based on first transforming the input, and then using simple rank/select primitives over compressed strings.

Mutual reinforcement relationship...

We investigated the reinforcement relation:

Compression ideas \(\Rightarrow\) Index design

Let’s now turn to the other direction

Indexing ideas \(\Rightarrow\) Compressor design

Booster
Compression Boosting for strings [Ferragina et al., J.ACM 2005]

Technically, we go from the bound
\[ |c| \leq \lambda |s| H_k(s) + \mu \]
...to the new performance bound
\[ |c'| \leq \lambda |s| H_k(s) + \log_2 |s| + \mu' \quad \forall k \]

Qualitatively, the booster offers various properties:
1. The more compressible is \( s \), the shorter is \( c' \) wrt \( c \)
2. It deploys compressor A as a black-box, hence no change to A’s structure is needed
3. No loss in time efficiency, actually it is optimal
4. Its performance holds for any string \( s \), it results better than Gzip and Bzip
5. It is fully combinatorial, hence it does not require any parameter estimations

Researchers may now concentrate on the “apparently” simpler task of designing 0-th order compressors
[see e.g. Landau-Vekrin, 05]

An interesting compression paradigm...

PPC paradigm (Permutation, Partition, Compression)

1) Given a string \( S \), compute a permutation \( \Pi(S) \)
2) Partition \( \Pi(S) \) into substrings
3) Compress each substring, and concatenate the results

\[ \checkmark \] Problem 1. Fix a permutation \( \Pi \). Find a partitioning strategy and a compressor that minimize the number of compressed bits.

\[ \checkmark \] If \( \Pi = \text{Id} \), this is classic data compression!

\[ \checkmark \] Problem 2. Fix a compressor \( C \). Find a permutation \( \Pi \) and partitioning strategy that minimize the number of compressed bits.

\[ \checkmark \] Taking \( \Pi = \text{Id} \), PPC cannot be worse than compressor \( C \) alone.
\[ \checkmark \] Our booster showed that a “good” \( \Pi \) can make PPC far better.
\[ \checkmark \] Other contexts: Tables [AT&T people], Graphs [Bondi-Vigna, WWW 04]

Paolo Ferragina, Università di Pisa
Compression of labeled trees [Ferragina et al., Focs ’05]

Extend the definition of $H_k$ to labeled trees by taking as $k$-context of a node its leading path of $k$-length (related to Markov random fields over trees)

A new paradigm for compressing the tree $T$

$\text{XBW}(T) + \text{Booster} + \text{any string compressor}$

The compression performance with Arithmetic is: $t \cdot H_k(T) + 2.01t + o(t)$ bits

This is a powerful paradigm for compressing both strings and labeled trees based on first transforming the input, then using the Booster over any known string compressor

Thanks!!
We investigated the reinforcement relation:

**Compression ideas** ⇨ **Index design**

Let’s now turn to the other direction

**Indexing ideas** ⇨ **Compressor design**

---

**What do we mean by “boosting”?**

It is a technique that takes a *poor* compressor $A$ and turns it into a compressor with *better performance guarantee*.

A memoryless compressor is *poor* in that it assigns codewords to symbols according only to their frequencies (e.g., Huffman).

It incurs in some obvious limitations:

- $T = a^n b^n$ (highly compressible)
- $T = \text{random string of } n \text{ 'a's and n 'b's}$ (uncompressible)
The empirical entropy $H_k$

For any $k$-long context

$H_k(T) = \frac{1}{|T|} \sum_{|\omega|=k} |T[\omega]| H_0(T[\omega])$

- $T[\omega]$ = string of symbols that precede $\omega$ in $T$

Example: Given $T =$ "mississippi", we have $T[i]$ = mssp, $T[is]$ = ms

Problems with this approach:
- How to go from all $T[\omega]$ back to the string $T$?
- How do we choose efficiently the best $k$?

Use BWT to approximate $H_k$

$H_k(T) = \frac{1}{|T|} \sum_{|\omega|=k} |T[\omega]| H_0(T[\omega])$

Remember that...

$T[\omega]$’s permutation

We have a workable way to approximate $H_k$
Finding the “best pieces” to compress...

Goal: find the best BWT-partition induced by a Leaf Cover!!

Some leaf covers are “related” to $H_k$!!!

A compression booster [Ferragina et al., JACM ’05]

- Let $Compr$ be the compressor we wish to boost
- Let $LC_1, \ldots, LC_r$ be the partition of $BWT(T)$ induced by a leaf cover $LC$, and let us define cost of $LC$ as $cost(LC, Compr)=\sum_j |Compr(LC_j)|$

Goal: Find the leaf cover $LC^*$ of minimum cost

- It suffices a post-order visit of the suffix tree (suffix array), optimal time
- We have: $Cost(LC^*, Compr) \leq Cost(H_k, Compr) = H_k(T), \forall k$

Technically, we show that

$$|k'| \leq \lambda, |s| H_{\lambda}(s) + f(|s|) + \log_2 |s| + \gamma, \forall k$$

Researchers may now concentrate on the “apparently” simpler task of designing 0-th order compressors [see e.g. Landau-Verbin, 05]

This is purely combinatorial. We do not need any knowledge of the statistical properties of the source, no parameter estimation, no training...
IBM Memory Expansion Technology (MXT) - Microsoft Internet Explorer

IBM Journal of Research and Development
Volume 45, Number 2, 2001
Technology for Server Systems
The article: 471.513.33 00 43 30 06 49 72 026 702 95113 90 14 44 39 75 41 78 8 8 8 8

IBM Memory Expansion Technology (MXT)


Several technologies are leveraged to establish an architecture for a low-cost, high-performance memory controller and memory system that more than doubles the effective size of the installed main memory without added cost. This architecture is the first of its kind to employ real-time main-memory content compression at a performance competitive with the best the market has to offer. A large low-latency shared cache exists between the processor bus and a content-compressed main memory. High-speed, low-latency hardware performs real-time compression and decompression of data traffic between the shared cache and the main memory. Sophisticated memory management hardware dynamically allocates main-memory storage in small sectors to accommodate storing the variable-sized compressed data without the need for "garbage" collection or significant wasted space due to fragmentation. Though the main-memory compression ratio is limited to the range 1:1.4:1, typical ratios range between 2:1 and 5:1, as measured in "real-world" system applications.

Introduction

Memory costs dominate both large-memory server and enterprise-computing server environments such as those employed in today's "data centers" and "computer farms." These costs are both fiscal and physical (e.g., volume, power, and performance associated with the memory system implementation), and often aggregate to a significant cost constraint for the information technology (IT) professional must trade off against computing goals.
Locate the pattern occurrences in T

\[ P = \#m\text{ississippi} \]

\[ T = \text{mississippi}\# \]

For this, we need to go backward (L-to-F):

- From s's position we get 4 + 3 = 7, ok !!
- This occurrence is listed immediately !

In theory, we set to \((\log N)^{1+\varepsilon}\) to balance space and locating time.

What about their practicality?

We have a library that currently offers:

- The FM-index: build, search, display, ...
- The Suffix Array: construction in space \((5+\varepsilon)\) n bytes
- The LCP Array: construction in space \((6+\varepsilon)\) n bytes
What about word-based searches?

T = "...bzip...bzip...unbzip...unbzip..."

...the post-processing phase can be time consuming!

The FM-index can be adapted to support word-based searches:

- Preprocess T and transform it into a "digested" text DT

  Word-search in T ⇔ Substring-search in DT

- Use the FM-index over the "digested" DT

The WFM-index

Digested-T derived from a Huffman variant: [Moura et al, 98]

- Symbols of the huffman tree are the words of T
- The Huffman tree has fan-out 128
- Codewords are byte-aligned and tagged

Any word

```
0 1
```

Tagging

```
\{0,1\}
```

Word "bzip or not bzip"

T = "bzip or not bzip"

DT

```
[1]
```

WFM-index

1. Dictionary of words
2. Huffman tree
3. FM-index built on DT

Space ~ 22 %
Word search ~ 4 ms
A historical perspective

Shannon showed a “narrower” result for a stationary ergodic $S$

- Idea: Compress groups of $k$ chars in the string $T$
- Result: Compress ratio $\to$ the entropy of $S$, for $k \to \infty$

- Various limitations
  - It works for a source $S$
  - It must modify $A$’s structure, because of the alphabet change
  - For a given string $T$, the best $k$ is found by trying $k=0,1,...,|T|$
    - $\Omega(|T|^3)$ time slowdown
    - $k$ is eventually fixed and this is not an optimal choice!

Two Key Components: Burrows-Wheeler Transform and Suffix Tree

How do we find the “best” partition (i.e. $k$)

- “Approximate” via MTF $\quad$ [Burrows-Wheeler, ’94]
  - MTF is efficient in practice [bzip2]
  - Theory and practice showed that we can aim for more!

- Use Dynamic Programming $\quad$ [Giancarlo-Sciortino, CPM ’03]
  - It finds the optimal partition
  - Very slow, the time complexity is cubic in $|T|

Surprisingly, full-text indexes help in finding the optimal partition in optimal linear time!!
Example: not one $k$

\[ s = (\text{bad})^n (\text{cad})^n (\text{xy})^n (\text{xz})^n \]

1-long contexts 2-long contexts

\[
\begin{align*}
    a_s &= \text{d}^{2n} < b_a = \text{d}^n, c_a = \text{d}^n \\
    x_s &= y^n z^n > y z \Rightarrow y x s = y^{n-1}, z x s = z^{n-1}
\end{align*}
\]