Compressed indexing: when compression equals search

Part IV: LZ-based self-indexing

Nicola Prezza

University of Udine, department of mathematics, physics, and computer science (DIMA)
Algorithmica A.A. 2015/2016
Outline

1. **Introduction**
2. **LZ full-text index**
   - LZ78 trie
   - LZ78 sparse suffix tree
   - Lexicographic range search
   - Handling secondary occurrences
   - Putting it all together
3. **LZ78 self-index**
4. **Combining BWT and LZ**
5. **References**
Why LZ?

**Repetitive text collections**

Consider a text \( T \), and a collection of texts \( T_1, \ldots, T_q \) that are almost identical to \( T \) \(^a\). What is the size of the compressed concatenation \( T_1 \ldots T_q \)?

\(^a\)e.g. the collection of all human genomes

**Simplification**

To simplify analysis, let \( T_1 = \ldots = T_q = T \)
Why LZ?

**Entropy compression**
- Not surprisingly, \( H_k(TT) \approx H_k(T) \): symbol frequencies do not change!
- As a result, \( qnH_k(T_1...T_q) \approx qnH_k(T) \) (for any \( k \leq n \))
- Entropy compression is **insensitive to long repetitions**

**LZ compression**

Let \( z_X(T) \) (\( X=77,78 \)) be the number of LZX factors of \( T \)
- On the other hand, \( z_{77}(TT) = z_{77}(T) + 1 \)
- More in general: \( z_{77}(T_1...T_q) = z_{77}(T) + 1 \)
- LZ77 compression is **sensitive to long repetitions**
- LZ78 is less powerful, but easier to index (LZ77 indexing uses similar techniques).

From now on, \( z \) will indicate \( z_X(T) \) (\( X=77,78 \) will be clear from the context)
LZ indexing

Main results:

**Background**

  \( O(z \log n) + (1 + o(1))n \log \sigma \) bits of space, \( O(m^2 + occ \log^\epsilon n) \) locate. Not a self index.

- **FM-LZI** (LZ78, see lecture 3). Ferragina and Manzini, 2005

- **NAV-LZI**[3] (LZ78). Navarro, 2007. \( O(z \log n) \) bits of space, \( O(m^3 \log \sigma + (m + occ) \log n) \) locate.

- **KN-LZI**[2] (LZ77). Kreft and Navarro, 2011. \( O(z \log n) \) bits, \( O((m^2 + occ) \log^2 n) \) locate.

- ... many more ...

Note: NAV-LZI and KN-LZI reach \( O(z \log n) \) bits of space. All other indexes have a \( o(n) \) space term
LZ indexing

We will study:

A simplified variant of the KU-LZI index

- $O(z \log n) + n \log \sigma$ bits of space
- $O(m(m + \log z) + \text{occ} \log z)$-time locate
- **Not a self index!** (we need the text)
- We will describe the structure using **LZ78**, but with small modifications the technique works also with **LZ77**

By using a technique from NAV-LZI, we will get rid of the text in the above index, turning it into a **self-index**

- $O(z \log n)$ bits of space and same query times as above
- This technique works only with **LZ78**
Overview

\[ \text{LZ78}(T$) = AC|CG|AC|AC|CA|CGG|T|GG|GT|\$ \]

Main idea

- Consider a pattern that spans at least 2 phrases (red in the example)
- Split the pattern in 2 parts: the one contained in the rightmost phrase (AC) and the rest (GAC)
- Find the interval \([l_{fw}, r_{fw}]\) of AC among the lexicographically sorted LZ phrases
- Find the interval \([l_{rev}, r_{rev}]\) of CAG (GAC reversed) among the lexicographically sorted reversed text prefixes ending at phrase boundaries
- 4-sided range search \([l_{fw}, r_{fw}] \times [l_{rev}, r_{rev}] \rightarrow \text{text position of the split}\)
- Different strategy if a pattern is contained in a phrase (we will use 2-sided range search) ...
Pattern occurrences

We divide pattern occurrences in 2 classes:

1. **Primary occurrences**: those spanning at least 2 LZ phrases or that end a phrase

2. **Secondary occurrences**: those contained in a single LZ phrase (and that do not end a phrase)
The LZ78 trie is the main reason why LZ78 is easier to index than LZ77: in the latter, phrases cannot be organized in a (small) trie.

LZ78 trie

Note: each phrase adds a character to a previous phrase, so all phrases can be organized in a trie with \( z + 1 \) nodes.

\[
LZ78( T$ ) = A | C | G | CG | AC | ACA | CA | CGG | T | GG | GT | $ 
\]
**LZ78 trie**

$LZ78(T$) = $A|C|G|CG|AC|ACA|CA|CGG|T|GG|GT|$

Phrases are in bijection with tree nodes (excluded root) $\Rightarrow$ there are exactly $z + 1$ nodes $\Rightarrow$ we can store the trie in $O(z \log z) \subseteq O(z \log n)$ bits$^a$

$^a$can be improved to $2z + o(z) + z \log \sigma$ bits by using succinct trees. For simplicity, we use a standard tree representation.
**Lexicographic order of the phrases**

Consider the **lexicographic order of the LZ78 phrases**. We can easily augment the LZ78 trie so that, at each node $N$, we know the range $[l^{fw}, r^{fw}]$ of lexicographic ranks of phrases in the subtree rooted in $N$. 
LZ78 trie

\[ LZ78(T$) = A|C|G|CG|AC|ACA|CA|CGG|T|GG|GT|$ \]

Lexicographic order of LZ phrases

0. $  
1. A  
2. AC  
3. ACA  
4. C  
5. CA  
6. CG  
7. CGG  
8. G  
9. GG  
10. GT  
11. T
\[ LZ78(T) = A|C|G|CG|AC|ACA|CA|CG|T|GG|GT|S \]

Notation: \( c_{[l_{fw}, r_{fw}]} \), where \( c \in \Sigma \) is the node label

Still \( \mathcal{O}(z \log n) \) bits of space
\[ \text{LZ78}(T\$) = A|C|G|CG|AC|ACA|CA|CGG|T|GG|GT\$ \]

Now do the same with the reversed text prefixes ending at phrase boundaries
Sparse suffix tree

\[ LZ78(T$) = A | C | G | CG | AC | ACA | CA | CGG | T | GG | GT | $ \]

Lexicographic order of the reversed \( T \) prefixes ending at phrase boundaries

0. \( TGGGTGGCACACACACAGCGCA \)
1. \( A \)
2. \( ACACACAGCGCA \)
3. \( ACACAGCGCA \)
4. \( CA \)
5. \( CAGCGCA \)
6. \( GCA \)
7. \( GCGCA \)
8. \( GGCACACACACAGCGCA \)
9. \( GGTGACACACACACAGCGCA \)
10. \( TGGCACACACAGCGCA \)
11. \( TGGGTGGCACACACACAGCGCA \)

Let \( T \) be the reversed text

Note: exactly \( z \) suffixes (of \( T \)), but if we organize them in a trie we end up with \( O(zn) \) nodes!
However, the trie has $z$ leaves, so the number of internal nodes with at least 2 children must be $O(z)$.

We use the same technique used in suffix trees: **path compression**

Instead of storing explicitly unary paths, we store 2 pointers $[\text{begin}, \text{end}]$ to the text (where $\text{begin} > \text{end}$: we read the text backwards).

For each explicit node $N$, we also store the interval $[l^{rev}, r^{rev}]$ of lexicographic ranks of $\prefix{T}$ suffixes in the subtree rooted in $N$.

**Space:** $O(z \log n)$ bits

We **need access to the text** so we store it in plain format: $n \log \sigma$ bits.
For clarity, we write only \([\text{begin}, \text{end}]\) labels. \([r^{\text{rev}}, r^{\text{rev}}]\) labels can be added easily in the picture.
Observation

If we wish to use **LZ77**, the trie of phrases can have more than $O(z)$ nodes (up to $O(n)$). However, if we have access to the text, we can use **path compression** as seen in the previous slide. The index described in this section therefore works also with LZ77.
Lexicographic range search

Handling primary occurrences

4-sided range search structure

For each phrase starting at position $t > 0$ in $T$, let $i$ be the lexicographic rank of the $T$ suffix ending in position $t - 1$, and $j$ be the lexicographic rank of the phrase starting in position $t$. We add a labeled 2D point $\langle \langle i, j \rangle, t \rangle$.
Lexicographic range search

Search splitted-pattern $CA \mid C$ (to find all primary occurrences, we will have to try all possible splits)
Lexicographic range search

Handling primary occurrences

**Space**

\[ z - 1 \text{ labeled points on the coordinate space } [0, z - 1] \times [0, z - 1] \implies z \log z(1 + o(1)) + z \log n \in O(z \log n) \text{ bits.} \]

**Time**

A range search query is answered in time \( O((\text{occ} + 1) \log z) \)
Lexicographic range search

Search algorithm

(1) Let \( P = p_1p_2...p_m \) be the pattern. For each split \( p_1...p_k/p_{k+1}...p_m \), we use the sparse suffix tree and the trie to find the range \([l^{rev}, r^{rev}]\) of \( p_k...p_1 \) among the sorted \( T \) suffixes and the range \([l^{fw}, r^{fw}]\) of \( p_{k+1}...p_m \) among the sorted phrases\(^a\)

\(^a\)Note: for things to work properly, we also consider the split \( p_1...p_m/ \). In this case, \([l^{fw}, r^{fw}]\) is the full range.

(2) We then query the 4-sided range structure on the rectangle \([l^{rev}, r^{rev}] \times [l^{fw}, r^{fw}]\) and retrieve all primary pattern occurrences (i.e. \( t - k \), where \( t \) is the label of each returned point)
Search algorithm

**Locate time**

\[ O(m(m + \log z) + occ_1 \log z) \] time, where \( occ_1 \) is the number of primary occurrences of \( P \) in \( T \).

**Count time**

\[ O(m(m + \log z)) \]. In the wavelet tree, we can stop as soon as we find a node whose range is contained in \([l^{fw}, r^{fw}]\) (no need to backtrack for each result point). Remember that the node cover has size \( O(\log z) \).

**Note**

If \( T \) is highly **compressible** (small \( z \)), search is **faster**!
### Secondary occurrences

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>G</td>
<td>C G</td>
<td>A C</td>
<td>A C A</td>
<td>C A</td>
<td>C G G</td>
<td>T</td>
<td>G G</td>
<td>G T</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Theorem

Each text substring $P$ has at least one primary occurrence.

#### Proof

Consider the leftmost secondary occurrence of $P$, at position $i$. Then, by definition, this occurrence is contained in a phrase $Z$. $Z$ is copied from a previous text position, so $P$ appears in the text also in another position $j < i$. 2 cases:

1. The occurrence of $P$ in position $j$ is secondary. Absurd, because we assumed that the occurrence in $i$ was the leftmost.
2. The occurrence of $P$ in position $j$ is primary.
Handling secondary occurrences

Secondary occurrences

**Corollary**

After finding primary occurrences, secondary occurrences can be found by recursively **following the chain of phrase copies**: if $P$ occurs in $T[i, ..., j]$, then retrieve all phrases that **entirely copy** $T[i, ..., j]$ (and repeat recursively)

**2-sided range search**

This problem can be solved by using a 2-sided range search data structure\(^a\)

\(^a\)Actually, with LZ78 we can again use the LZ78 trie\(^3\). However, by using 2-sided range our index works also with LZ77 and is therefore more general.
Handling secondary occurrences

Secondary occurrences

2-sided range data structure

For each phrase $Z = T[t, \ldots, t']$ copied from $T[i', \ldots, j']$ ($t' - t = j' - i'$), we add a labeled 2D point $\langle\langle i', j'\rangle, t\rangle$

In the figure below, pattern $P$ occurs in $T[i, \ldots, j]$ and $Z$ entirely copies it (hence $P$ occurs also in $T[t + (i - i'), \ldots, t + (j - i')]$)
Handling secondary occurrences

2-sided range data structure

note: point P is not inserted in the range structure (P is the query)
Handling secondary occurrences

Re-map coordinates to \([0, z - 1]\)

Re-mapping coordinates

Note that now point coordinates span on \([0, n - 1]\) and can be repeated. We can remove duplicates and re-map all coordinates to \([0, z - 1]\) by using two **sparse bitvectors** of length \(n\) with \(\leq z\) 1-bits each and one **succinct bitvector** of length \(z\) (exercise).

Sparse bitvector

A simple implementation of a sparse bitvector \(B[0, ..., n - 1]\) with \(b\) 1-bits consists in the sorted array of 1-bits positions.

- **Space:** \(b \log n\) bits
- **Access/rank:** \(O(\log b)\) (binary search)
- **Select:** \(O(1)\) (1 access)

Example

- \(B = 000000100000010001000000000010000\)
- structure: \(\langle 6, 13, 17, 29 \rangle\)

\(O(z \log n)\) additional bits
Handling secondary occurrences

Search algorithm

For each primary occurrence $T[i, ..., j]$ of $P$, call $\text{RetrieveSecondaryOcc}(i, j)$

$\text{RetrieveSecondaryOcc}(i, j)$

1. Query the 2-sided structure on the rectangle $[-\infty, i] \times [j, \infty]$.
2. For all secondary occurrences $T[i', ..., j']$ found, call $\text{RetrieveSecondaryOcc}(i', j')$. 


Search algorithm

Locate time

$O(occ_2 \log z)$ time, where $occ_2$ is the number of secondary occurrences of $P$ in $T$.

Count time

We perform count and locate simultaneously.
The LZ full-text index

Main result

The LZ index we described is a **full-text index** based on LZ78/LZ77 parsing that offers the following space/time tradeoffs:

- $\mathcal{O}(z \log n) + n \log \sigma$ bits of space

- $\mathcal{O}(m(m + \log z) + occ \log z)$ count/locate time, where $occ$ is the number of occurrences of $P$ in $T$. 
Getting rid of the text

Actually, if we restrict our attention to LZ78 it is quite easy to turn the LZ full-text index into a self-index:

Fact

The LZ78 trie can be augmented with $2z$ pointers to nodes ($2z \log z$ bits) so that it supports the extraction of any length-$L$ substring of the reversed text in optimal $O(L)$ time. Moreover, each additional character extraction takes $O(1)$ time.

Corollary

Instead of using the text, we can use the LZ78 trie itself to support path compression in the sparse suffix tree.
A|C|G|CG|AC|AC|CA|CA|CGG|T|GG|GT|$
LZ78 trie additional edges

Augmenting the LZ78 trie

Let $X$ and $Y$ be two LZ78 phrases (= two nodes of the trie). we add the following edges:

- **Parent**: if $Y$ is child of $X$ in the trie, then we add the edge $\pi(Y) = X$

- **Previous phrase**: if $X$ immediately precedes $Y$ in the text then we add the edge $pr(Y) = X$

Particular cases

- $\pi(root) = root$
- If $X$ is the first text phrase, $pr(X) = NULL$
- To simplify the description, in the next slides we assume that we do not reach $T[0]$ during extraction
Extraction algorithm

**Text extraction**

Suppose we want to extract (backwards) $L$ text characters starting from $T[i]$. Let:

- $X$ be the phrase containing position $i$ (note: $X$ is a trie node)
- $N$ be the trie node corresponding to $T[i]$

To start extraction, we just need the coordinate pair $\langle X, N \rangle$. 
Extraction algorithm

Algorithm 1: extract(⟨X, N⟩, L)

if L = 0 then
  return ε; /* return empty string */
c ← char(N); /* character stored in node N */

if π(N) = root then
  X ← pr(X); /* jump to previous phrase */
  N ← X; /* next character is the last of previous phrase */
  return extract(⟨X, N⟩, L − 1) · c; /* next L − 1 chars + this char */
else
  return extract(⟨X, π(N)⟩, L − 1) · c;
Getting rid of the text

Now, simply substitute intervals $[\text{begin}, \text{end}]$ in the sparse suffix tree with pairs $\langle \langle X, N \rangle, L \rangle$, where:

- $X$ is the phrase (trie node) containing text position $\text{begin}$
- $N$ is the trie node corresponding to $T[\text{begin}]$
- $L = \text{begin} - \text{end} + 1$

**Space**

- Additional $2z \log z \in \mathcal{O}(z \log n)$ bits
- We can discard the text
The LZ78 self-index

Main result
The LZ index we described is a **self-index** based on LZ78 parsing that offers the following space/time tradeoffs:

- $O(z \log n)$ bits of space
- $O(m(m + \log z) + occ \log z)$ count/locate time, where $occ$ is the number of occurrences of $P$ in $T$.

The space can also be upper-bounded by $O(nH_k) + o(n \log \sigma)$ bits, with $k \in o(\log_{\sigma} n)$ (see lecture 1)
Fast and small: is it possible?

- LZ77-based self-indexes can be *exponentially* smaller than the text, but (until now) they are slower than BWT-based self-indexes ($O(m^2)$ vs $O(m)$).
- **BWT+LZ77** can reach both goals *simultaneously*. 
BWT+LZ77

BWT runs

Consider a text $T$. Let $r(T)$ be the number of equal-letters runs in $BWT(T)$.

Example

$T=$mississippi. $BWT(T$$)=$ipssm$pissii$. $r(T) = 9$

Repetitive texts

- Let $T_1 = \ldots = T_q = T$. Then, $r(T_1 \ldots T_q) \approx r(T)$
- $r(T)$ can be exponentially smaller than $|T|$!

Notation: we just write $r$ instead of $r(T)$
Run-length encoded BWT

We can build a **run-length encoded FM index** on \( T \) (RLBWT) that takes \( \mathcal{O}(r \log n) \) bits of space and that supports finding the BWT range of a pattern \( P \in \Sigma^m \) in \( \mathcal{O}(m \log \log n) \) time.

A C++ implementation: RLCSA (run-length compressed suffix array). 
github.com/adamnovak/rlcsa
LZ-RLCSA = LZ77+RLBWT

**LZ77+RLBWT: main idea**

- Compute the **LZ77 parsing** of $T$
- **Replace the trie and sparse suffix tree** of the LZ index (see prev. slides) with **2 RLBWT** (on forward and reverse text)
- Instead of using SA samples ($o(n)$ bits) to support `locate`, use components of the LZ index ($O(z \log n)$ bits)

**Advantage**

If we know the BWT range of $p_k p_{k+1} ... p_m$, with the forward FM index we can find the range of $p_{k-1} p_k p_{k+1} ... p_m$ with $O(1)$ rank queries (similar for the reverse): **time becomes linear in $m$!**
LZ-RLCSA: a fast repetition-aware data structure

Data structure: lz-rlcisa[1]

- Space: $O((r + z) \log n)$ bits
- Locate time: $O((m \log \log n + occ) \log z)$

$a$ can be $O(\log n)$ for extremely repetitive texts!

C++ implementation

http://github.com/nicolaprezza/lz-rlcsa
References


