Lempel-Ziv full-text indexing

Nicola Prezza

Technical University of Denmark

DTU Compute

Building 322, Office 006

Outline

Introduction

compressed indexing

- Suffix trees are very fast indexes, but they take $\Theta(n)$ space
- In practice, this translates to $20n$ Bytes in the most space-efficient implementations
- The plain text takes n Bytes.
- The LZ77-*compressed* text can be *thousands* of times smaller than the text itself (if the text is very compressible)

Million dollar question

Can we build a fast index taking $\mathcal{O}(z)$ words of space?[1]

[1] z = number of phrases in the LZ78/LZ77 parse. When clear from the context, we will simply write z instead of z_{77} or z_{78}

Indexing: definitions

A full-text index $\mathcal{I}(T)$ is a data structure that permits to efficiently (polylog time) answer these queries:

- **count**: report number *occ* of occurrences of a string P in T
- **locate**: report locations of the *occ* occurrences of P in T
- **extract**: return $T[i, ..., i + m]$

We will consider only `locate` queries in these slides

- If $\mathcal{I}(T)$'s size is proportional to that of the compressed text, $\mathcal{I}(T)$ is a compressed index
- If $\mathcal{I}(T)$ does not need T to answer queries, $\mathcal{I}(T)$ is a self-index.

LZ indexing

State of the art

index	compression	space (words)	locate time
KU-LZI[1]	LZ78	$\mathcal{O}(z) + n$	$\mathcal{O}(m^2 + occ \log^{\epsilon} n)$
NAV-LZI[3]	LZ78	$\mathcal{O}(z)$	$\mathcal{O}(m^3 \log \sigma + (m + occ) \log n)$
KN-LZI[2]	LZ77	$\mathcal{O}(z)$	$\mathcal{O}\left((m^2 h + (m + occ) \log z) \log \frac{n}{z}\right)$

m = pattern length. h = parse height (defined later)

Note 1. For simplicity, we will assume the text is stored in n words.

Note 2. KU-LZI needs the text: not a self-index.

LZ indexing

We will combine ideas from all these indexes and first describe a (LZ77/LZ78) full-text index with these bounds:

- $\mathcal{O}(z \log n) + n$ space (text needed: not a self-index)
- $\mathcal{O}(m(m + \log z) + occ \log n)$-time locate

We start describing the index with LZ78 and then extend to LZ77

To conclude, we will see how to get rid of the text and turn the index into a *self-index* (same speed with LZ78, slower with LZ77)

LZ full-text index

Overview

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

General idea (applies to both LZ78 and LZ77)

- Consider the orange occurrence of $GACAC$
- Consider the phrase-aligned split $GAC|AC$
- AC is a prefix of a phrase
- $CAG = \overleftarrow{GAC}$ is a prefix of a phrase-aligned suffix of the reversed text
- these two prefixes define two ranges among:
  - lexicographically sorted LZ phrases (AC)
  - lexicographically sorted suffixes of the reversed text (CAG)

  $\Rightarrow$ 2D range search (2D range trees)
- To find the remaining occurrences: track phrases entirely copying occurrences that span ≥ 2 phrases (again range search)

Pattern occurrences

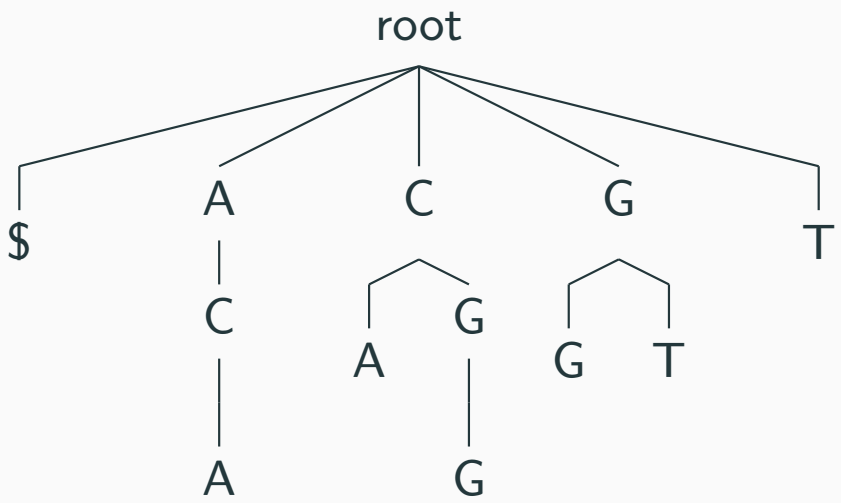
We divide pattern occurrences in 2 classes:

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)

LZ78 trie

Recall that LZ78 has an elegant trie representation:

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



Phrases are in bijection with tree nodes (root excluded) $\Rightarrow$ there are exactly $z + 1$ nodes $\Rightarrow$ we can store the trie in $\mathcal{O}(z)$ words of space [2]

[2] can be improved to $2z + o(z) + z \log \sigma$ bits using succinct trees. For simplicity, we use a pointer-based tree representation.

LZ78 trie

Consider the lexicographic order of LZ phrases:

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

0. $
1. A
2. AC
3. ACA
4. C
5. CA
6. CG
7. CGG
8. G
9. GG
10. GT
11. T

LZ78 trie

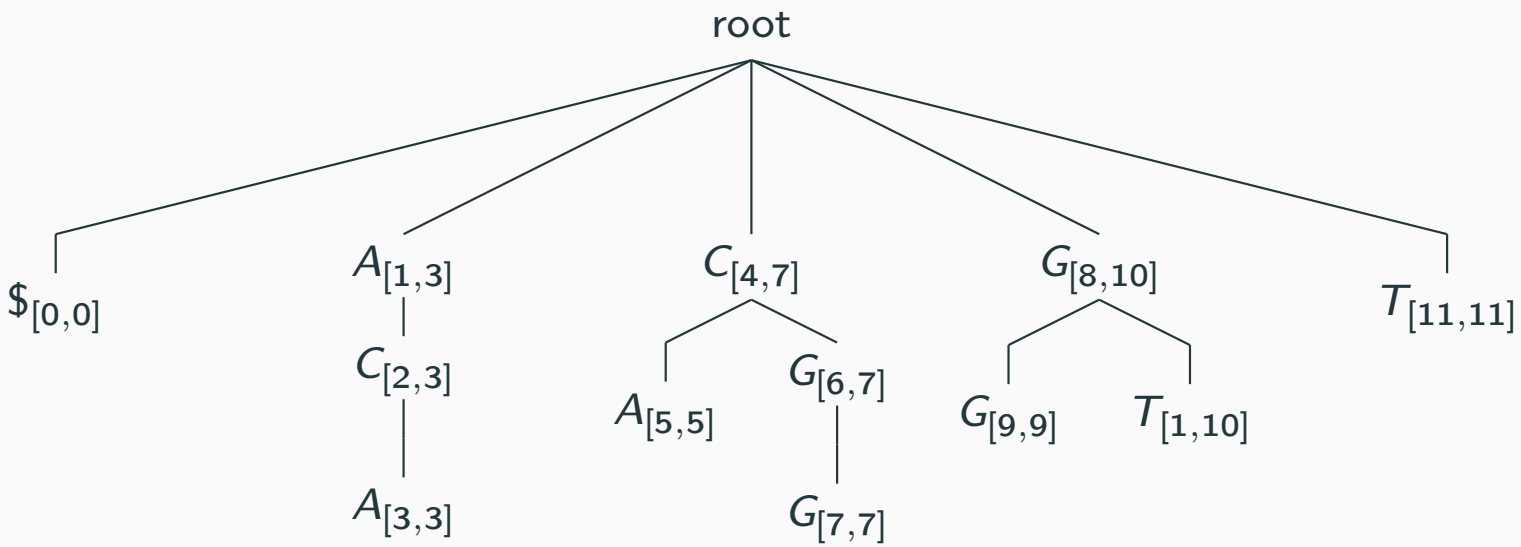
Lexicographic order of phrases

We can augment the trie writing, on each node N, the range $[l^{fw}, r^{fw}]$ of lexicographic ranks of phrases in the subtree rooted in N.

Augmented LZ78 trie

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

Notation: $c_{[l^{fw}, r^{fw}]}$, where $c \in \Sigma$ is the node label



Still $\mathcal{O}(z)$ space

Now do the same with phrase-aligned suffixes of the reversed text (lexicographically sorted)

Space-efficient solution: *sparse suffix tree*

Sparse suffix tree

$LZ78(T\$) = A|C|G|CG|AC|ACA|CA|CGG|T|GG|GT|\$$
$reverse(LZ78(T\$)) = \$|TG|GG|T|GGC|AC|ACA|CA|GC|G|C|A$

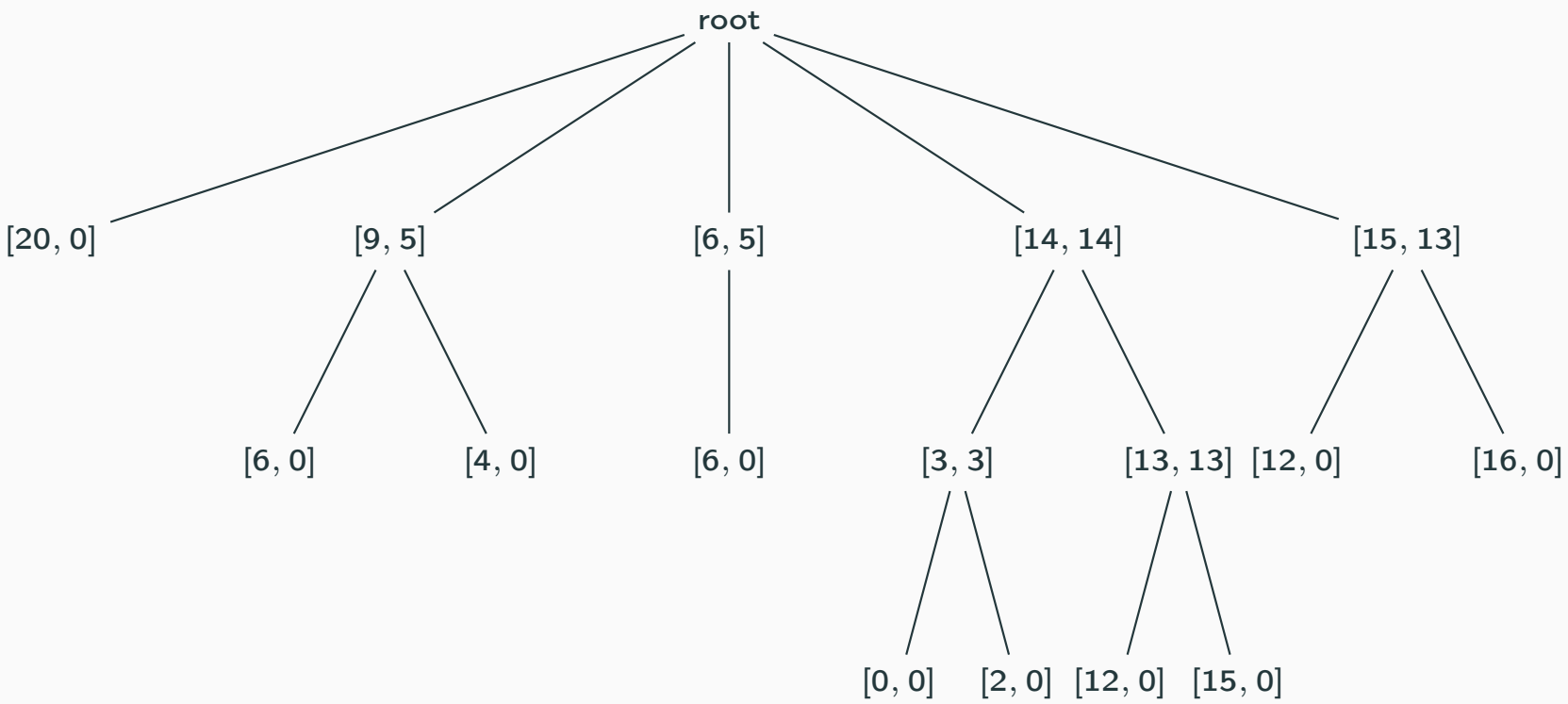
Lexicographic order of phrase-aligned suffixes of the reversed text

0. $\$TGGGTGGCACACACAGCGCA$
1. A
2. $ACACACAGCGCA$
3. $ACACAGCGCA$
4. CA
5. $CAGCGCA$
6. GCA
7. $GCGCA$
8. $GGCACACACAGCGCA$
9. $GGTGGCACACACAGCGCA$
10. $TGGCACACACAGCGCA$
11. $TGGGTGGCACACACAGCGCA$

Sparse suffix tree

- For clarity, the picture shows only [*begin*, *end*] labels.
- **exercise:** add $[l^{rev}, r^{rev}]$ labels in each explicit node

```
0   1   2   3 4   5 6   7 8 9   10 11  12 13 14  15  16 17  18 19  20
A | C | G | C G | A C | A C A | C  A | C  G  G | T | G  G | G  T | $
```



LZ77 trie

... what about LZ77?

If we wish to use **LZ77**, the trie of phrases can have more than $\mathcal{O}(z)$ nodes (up to $\mathcal{O}(n)$)

First exercise session (ex. 1 and 2): find a space-efficient solution to represent the LZ77 trie. Build the LZ77 sparse suffix tree of the previous example.

Primary occurrences

```
0   1   2   3 4   5 6   7 8 9   10 11  12 13 14  15  16 17  18 19  20
A | C | G | C G | A C | A C A | C  A | C  G  G | T | G  G | G  T | $
```

range search structure

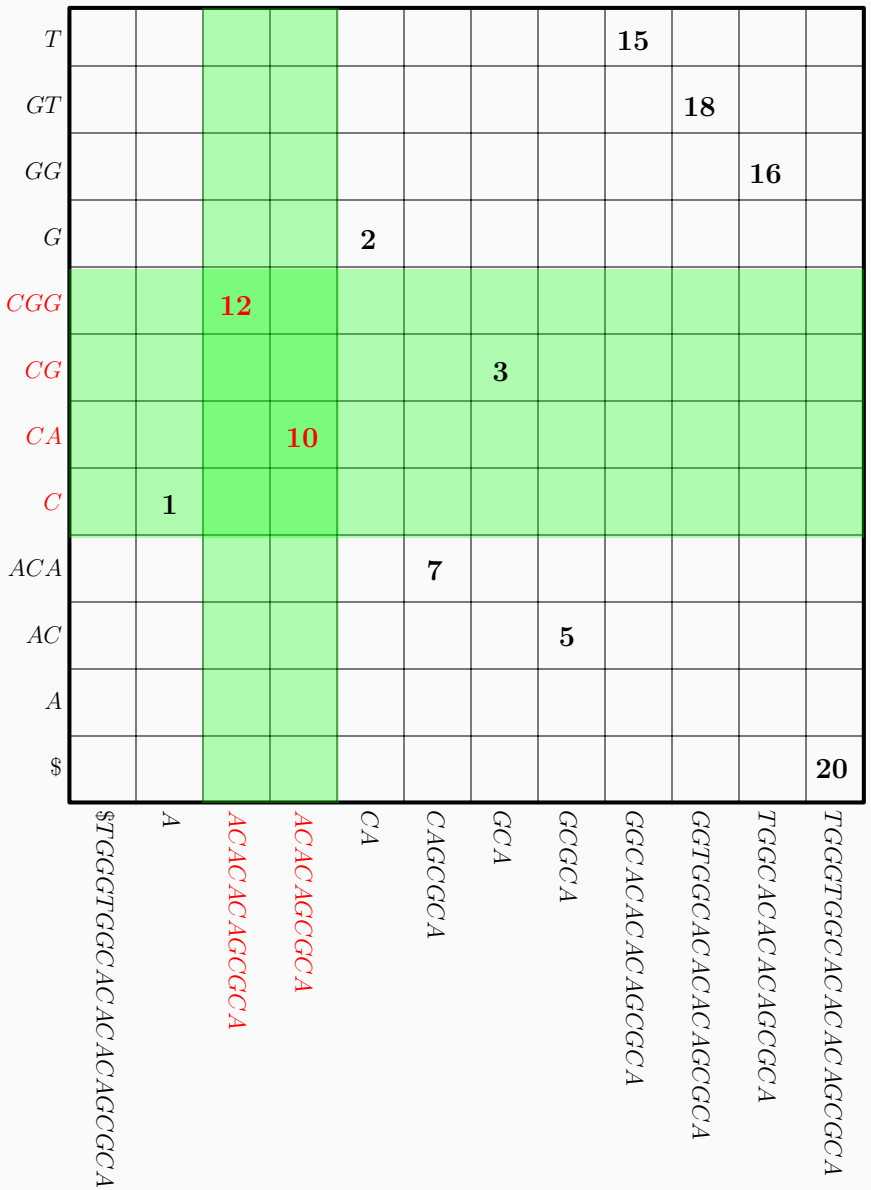
For each phrase starting at position $t > 0$ in T, let:

- i be the lexicographic rank of the $\overleftarrow{T}$ 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 $\langle\langle i, j\rangle, t\rangle$ in a range data structure

Example: search splitted-pattern $\overleftarrow{CA}|\overrightarrow{C}$ (to find *all* primary occurrences, we have to try all possible splits)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	|	C	|	G	|	C	G	|	A	C	|	A	C	A	|	C	A	|	C	G	G	|	T	|	G	G	|	G	T	|	$



Primary occurrences

We store $z-1$ 2D points from $[0, z-1]^2$ with labels from $[0, n-1]$. Using 2D range trees:

- $\mathcal{O}(z \log z) + z - 1 \subseteq \mathcal{O}(z \log n)$ space[3]
- A range reporting query is answered in time $\mathcal{O}(k + \log z)$, where k is the number of returned points

[3] Note: this solution is not space-optimal. Wavelet trees [4] use only $\mathcal{O}(z)$ words.

Primary occurrences: locate

Let $P = p_1 p_2 \ldots p_m$ be the pattern. *For each split* $p_1 \ldots p_d / p_{d+1} \ldots p_m$:

1. use the sparse suffix tree and the trie to find the range $[l^{rev}, r^{rev}]$ of $p_d \ldots p_1$ among the sorted $\overleftarrow{T}$ suffixes and the range $[l^{fw}, r^{fw}]$ of $p_{d+1} \ldots p_m$ among the sorted phrases[4]
2. Query the range structure on the rectangle $[l^{rev}, r^{rev}] \times [l^{fw}, r^{fw}]$. For every retrieved label t, return occurrence $t - d$

[4] Note: we also consider the split $p_1 \ldots p_m /$. In this case, $[l^{fw}, r^{fw}]$ is the full range.

Primary occurrences: locate

Let occ_1 be the number of primary occurrences of P in T. We can retrieve these occurrences in $\mathcal{O}(m(m + \log z) + occ_1)$ time.

Secondary occurrences

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)

This problem can be solved using a range search data structure[5]

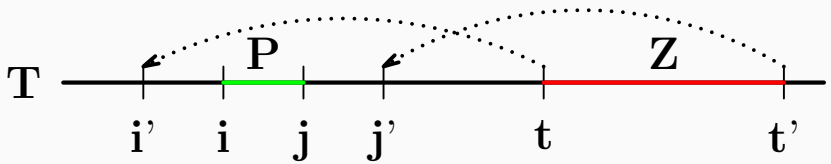
[5] Actually, with LZ78 we can again use the LZ78 trie [3]. However, by using range search our index works also with LZ77 and is therefore more general.

Secondary occurrences

range data structure

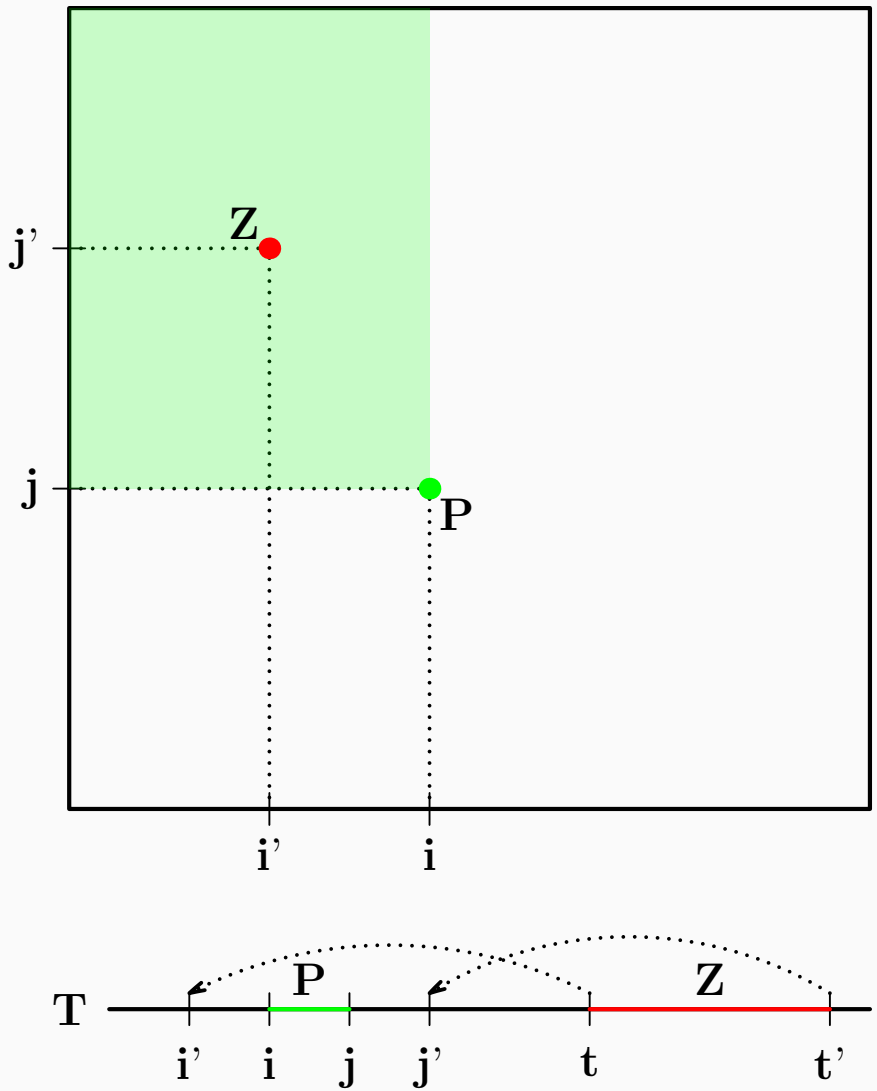
For each phrase $Z = T[t, ..., t']$ copied from $T[i', ..., j']$ $(t' - t = j' - i')$, we add a labeled 2D point $\langle\langle i', j'\rangle, t\rangle$ in a range data structure

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



Secondary occurrences: 2-sided range search

Example: after finding P (i.e. pattern occurrence), use it to query the structure and find other occurrences



Secondary occurrences: 2-sided range search

We store z 2D points from $[0, n-1]^2$ with labels from $[0, n-1]$. Using 2D range trees:

- $\mathcal{O}(z \log n)$ space[6]
- A range reporting query is answered in time $\mathcal{O}(k + \log n)$, where k is the number of returned points

[6] with wavelet trees [4] this space becomes $\mathcal{O}(z)$

Full locate algorithm

Locate primary occurrences. For each primary occurrence $T[i, ..., j]$ of P, call the following procedure:

RetrieveSecondaryOcc(i, j)

1. Query the 2-sided structure on the rectangle $[0, i] \times [j, n-1]$
2. For all secondary occurrences $T[i', ..., j']$ found, call *RetrieveSecondaryOcc*(i', j')

Second exercise session (ex. 3): prove the correctness and completeness of the search algorithm

The LZ full-text index

Theorem

The LZ index we described is a **full-text index** based on LZ78/LZ77 that:

- takes $\mathcal{O}(z \log n) + n$ words of space
- supports locate in $\mathcal{O}(m(m + \log z) + occ \log n)$ time[7]

[7] in the worst case, $occ \in \Theta(occ_2)$ and we perform a separate 2D range reporting for each secondary occurrence (e.g. search A in the LZ77 index for $A^{n-1}\$$)

LZ self-index

LZ78 self-index

- Note: the text is needed only to support path-compression in the tries/suffix trees
- Recall that we can extract any character from LZ78 in $\mathcal{O}(\log \log n)$ time
- This implies we can delete the text and obtain a *LZ78 self-index*. Times are multiplied by $\mathcal{O}(\log \log n)$ w.r.t. the full-text index.

Exercise Show that, on LZ78, the $\mathcal{O}(\log \log n)$ slowdown can be avoided (i.e. show a LZ78 self-index as fast as the full-text index)

LZ77 self-index

LZ77 is harder to treat: we need access to the text to compress both the LZ77 trie and the LZ77 sparse suffix tree

Parse height

Definition

The height h_i of character $T[i]$ is the number of times we have to "jump back" from position i (following phrase copies) until we find an explicitly stored character

Definition

The parse height h of the LZ77 parse is defined as $h = \max_{i=0,\ldots,n-1} h_i$

In real-case texts, h is very small [2]

LZ77 self index

Exercise: show how to extract any text character in $\mathcal{O}(h \log \log n)$ time using a data structure of size $\mathcal{O}(z_{77})$ words. We obtain:

Theorem

The LZ index we described is a **self-index** based on the LZ77 parsing that:

- takes $\mathcal{O}(z \log n)$ words of space
- supports locate in $\mathcal{O}(m(m \cdot h \cdot \log \log n + \log z) + occ \log n)$ time

NB: using wavelet trees we can reach the optimal $\mathcal{O}(z)$ words of space

References

References I

Juha Kärkkäinen and Esko Ukkonen.
Lempel-Ziv parsing and sublinear-size index structures for string matching.
In *Proc. 3rd South American Workshop on String Processing (WSP'96)*. Citeseer, 1996.

Sebastian Kreft and Gonzalo Navarro.
On compressing and indexing repetitive sequences.
Theoretical Computer Science, 483:115–133, 2013.

Gonzalo Navarro.
Indexing text using the ziv–lempel trie.
Journal of Discrete Algorithms, 2(1):87–114, 2004.

References II


Gonzalo Navarro.
Wavelet trees for all.
Journal of Discrete Algorithms, 25:2–20, 2014.


Gonzalo Navarro and Veli Mäkinen.
Compressed full-text indexes.
ACM Computing Surveys (CSUR), 39(1):2, 2007.


Nicola Prezza.
***Compressed computation for text indexing*.**
PhD thesis, Universitá degli studi di Udine, 2016.