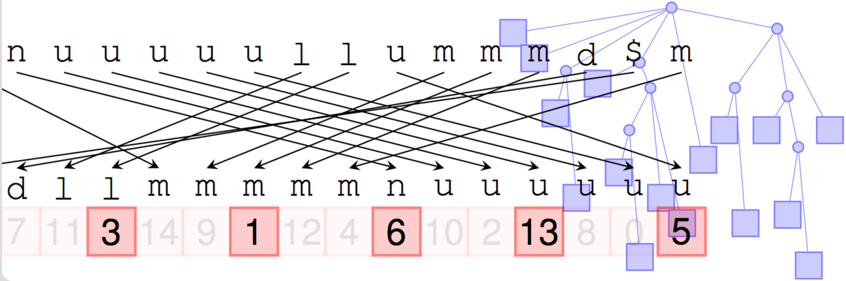
Karlsruhe Institute of Technology

Text Indexing: Lecture 7

Simon Gog – *gog@kit.edu*

Institute of Theoretical Informatics - Algorithmics



Range Querying using Wavelet Trees

Problem 2

Given a $[0, n-1] \times [0, n-1]$ grid G and a set P of n points $(i, S[i])$ with weight $w[i]$ for $0 \leq i < n$. For a pair of points (x_0, y_0) and (x_1, y_1) with $x_0 \leq x_1$ and $y_0 \leq y_1$ we define the top-k range query:

- A top-k *range report query* asks for the k points $(x, y) \in P$ such that $x \in [x_0, x_1]$ and $y \in [y_0, y_1]$ with maximum weight sorted in decreasing order of weight.

Results [5]

Top-k range queries can be answered in $O(\log^2 n + k \cdot \log n)$ time using a index of size $3n \log n + o(n \log n) + |weights|$ bits, where $|weights|$ is the space required to store the weights associated with the n points.

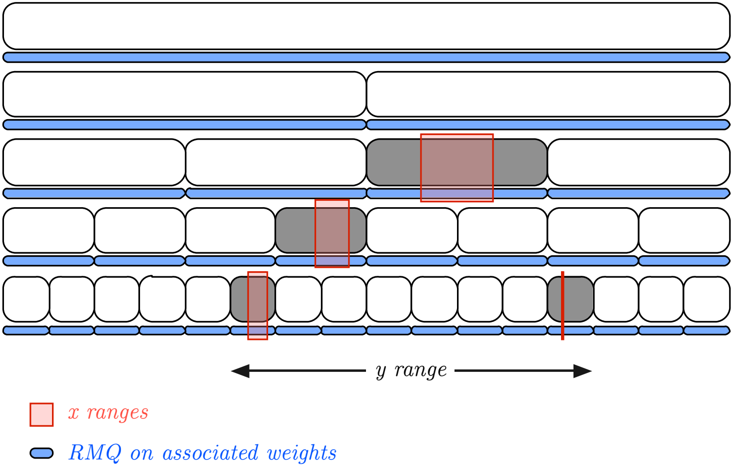
Top-k Range Report Queries

Outline of solution

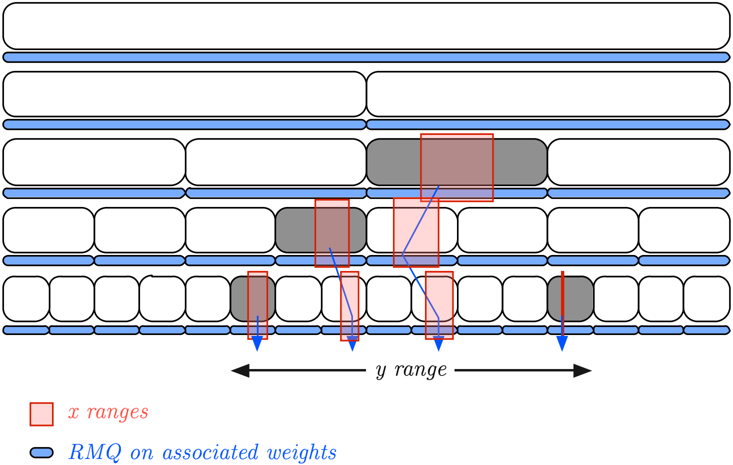
- Use the range count algorithm to get the $O(\log n)$ WT nodes C which cover the y range
- For each x range in C use the RMQ structure to navigate to the haviest point ($O(\log n)$ time per range, i.e $O(\log^2 n)$ total)
- Insert the haviest points into a max priority queue Q
- Remove maximum point from Q, report it and split its corresponding x range in C. Navigate to the haviest points in the two new ranges Q. Insert
- Repeat last step until Q is empty or k points were reported

Total time: $O(\log^2 n + k \cdot \log n)$

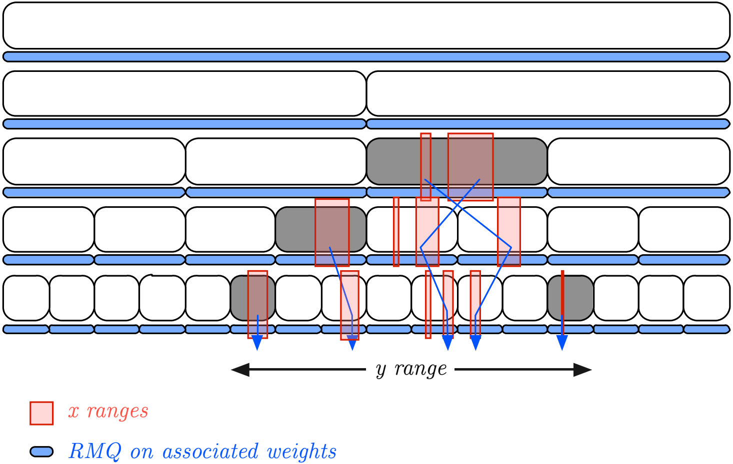
Top-k Range Report Queries



Top-k Range Report Queries



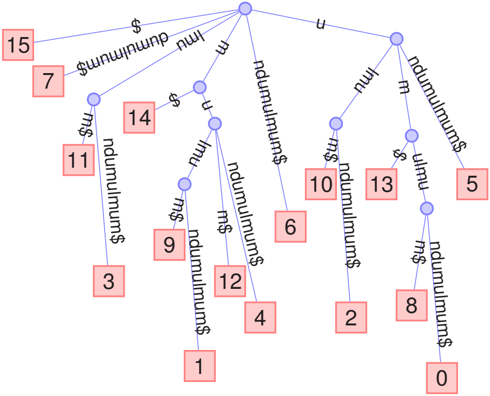
Top-k Range Report Queries



Suffix Trees

- The suffix tree (ST) extends the functionality of suffix array
- Construction in three phases (each takes linear time)
  - suffix array construction (Timo Bingmann will present this at the 21th of January 2016)
  - LCP array construction
  - tree topology construction
- As the pointer-based representation takes too much space in most application we will present a more space-efficient version: The compressed suffix tree (CST).

Suffix Trees



15	7	11	3	14	9	1	12	4	6	10	2	13	8	0	5
0	0	0	3	0	1	5	2	2	0	0	4	1	2	6	1

(()()(()())(()((()())()()))()((()())(()(()()))()))

- Versatile index structure: E.g. we can solve longest common substring queries for *k* strings efficiently.
- Exercise: Find longest palindromic substring of a string.

Space-Efficient Suffix Tree Representations

Most representations consist of three parts:

- Suffix Array (leaves of suffix tree)
- LCP Array (longest common prefix lengths/depth of inner nodes)
- Tree Topology

LCP Array

Definition

Let $lcp(U, V)$ denote the longest common prefix between two strings U and V. For a text $\mathcal{T}$ of size n the longest common prefix (LCP) array of size $n+1$ is defined as follows. $LCP[i] = |lcp(\text{SA}[i], \text{SA}[i-1])|$ for $i \geq 1$ and $LCP[0] = 0$ and $LCP[n] = -1$.

([4] introduced this array as *Hgt* array)

LCP Array – Example

i	SA	LCP	$\mathcal{T}$[SA[i],$n-1$]
0	11	0	`$`
1	10	0	`i$`
2	7	1	`ippi$`
3	4	1	`issippi$`
4	1	4	`ississippi$`
5	0	0	`mississippi$`
6	9	0	`pi$`
7	8	1	`ppi$`
8	6	0	`sippi$`
9	3	2	`sissippi$`
10	5	1	`ssippi$`
11	2	3	`ssissippi$`

- Time complexity of naive computation (for each $i > 0$, compare suffix SA$[i]$ and SA$[i-1]$): $\mathcal{O}(n^2)$.
- Comparison in SA-order.

Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

0	1	2	3	4	5	6	7	8	9	10	11
m	i	s	s	i	s	s	i	p	p	i	$

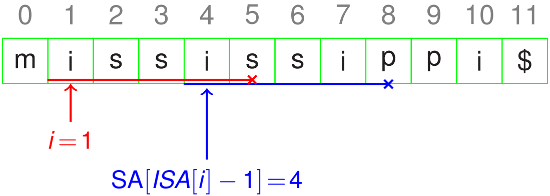
$i = 0$

$\mathrm{SA}[ISA[i] - 1] = 1$

0

Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.



0 4

Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

0	1	2	3	4	5	6	7	8	9	10	11
m	i	s	s	i	s	s	i	p	p	i	$

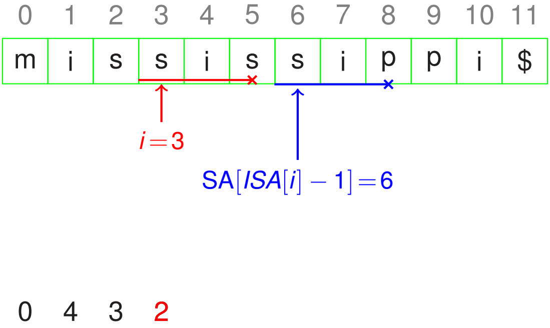
$i = 2$

$\mathrm{SA}[ISA[i] - 1] = 5$

0 4 3

Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.



Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

0	1	2	3	4	5	6	7	8	9	10	11
m	i	s	s	i	s	s	i	p	p	i	$

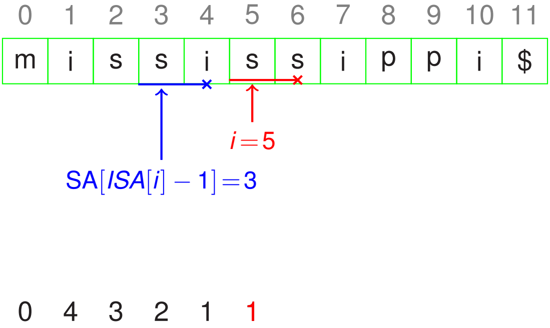
$i = 4$

$\mathrm{SA}[ISA[i]-1] = 7$

0 4 3 2 1

Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.



Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

0	1	2	3	4	5	6	7	8	9	10	11
m	i	s	s	i	s	s	i	p	p	i	$

$i = 6$

$\mathsf{SA}[ISA[i]-1] = 8$

0 4 3 2 1 1 0

Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

0	1	2	3	4	5	6	7	8	9	10	11
m	i	s	s	i	s	s	i	p	p	i	$

$i = 7$

$SA[ISA[i] - 1] = 10$

0 4 3 2 1 1 0 1

Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

0	1	2	3	4	5	6	7	8	9	10	11
m	i	s	s	i	s	s	i	p	p	i	$

$i = 8$

$\mathrm{SA}[ISA[i]-1] = 9$

0 4 3 2 1 1 0 1 1

Simon Gog:
Text Indexing: Lecture 7
Institute of Theoretical Informatics
Algorithmics

Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

0	1	2	3	4	5	6	7	8	9	10	11
m	i	s	s	i	s	s	i	p	p	i	$

$i = 9$

$\mathrm{SA}[ISA[i]-1] = 0$

0 4 3 2 1 1 0 1 1 0

Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

0	1	2	3	4	5	6	7	8	9	10	11
m	i	s	s	i	s	s	i	p	p	i	$

$i = 10$

$\mathsf{SA}[ISA[i]-1] = 11$

0	4	3	2	1	1	0	1	1	0	0

Linear Time Calculation of LCP Array

Idea of [3]: Processing in text-order.

0	1	2	3	4	5	6	7	8	9	10	11
m	i	s	s	i	s	s	i	p	p	i	$

$i = 11$

$\mathrm{SA}[ISA[i] - 1] = 5$

0	4	3	2	1	1	0	1	1	0	0	0

Linear Time Calculation of LCP Array

Lemma ([3])

For $0 < i \leq n$, we have $LCP[ISA[i]] \geq LCP[ISA[i-1]] - 1$.

Linear Time Calculation of LCP Array

$LCP[0] \leftarrow 0$
$LCP[n] \leftarrow -1$
for $i \leftarrow 0$ **to** $n-1$ **do**
  $ISA[\mathrm{SA}[i]] \leftarrow i$
$\ell \leftarrow 0$
for $i \leftarrow 0$ **to** $n-1$ **do**
  $j \leftarrow \mathrm{SA}[(ISA[i]-1) \bmod n]$
  while $\mathcal{T}[i+\ell] = \mathcal{T}[j+\ell]$ **do**
    $\ell \leftarrow \ell + 1$
  $LCP[ISA[i]] \leftarrow \ell$
  $\ell \leftarrow \max(0, \ell - 1)$

Exercise
How much memory is required during the algorithms execution?

Linear Time Calculation of LCP Array

Engineered version of [2]:

```
for i ← 0 to n − 1 do
   Φ[SA[i]] ← SA[(i − 1) mod n]
ℓ ← 0
for i ← 0 to n − 1 do
   j ← Φ[i]
   while T[i + ℓ] = T[j + ℓ] do
      ℓ ← ℓ + 1
   PLCP[i] ← ℓ
   ℓ ← max(0, ℓ − 1)
for i ← 0 to n − 1 do
   LCP[i] ← PLCP[SA[i]]
LCP[n] ← −1
```

(Explain why this algorithm is faster in practice)

Space-Efficient LCP Representation

[6] proposed to use a CSA and the permuted LCP array (LCP values in text-order) PLCP. We know

- $PLCP[i+1] \geq PLCP[i] - 1$
- $PCLP[i] \leq n - 1 - i$ (for $0 \leq i < n$)

$$
\begin{array}{rcccccccccccc}
i = & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
PCLP[i] = & 0 & 4 & 3 & 2 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0
\end{array}
$$

Space-Efficient LCP Representation

[6] proposed to use a CSA and the permuted LCP array (LCP values in text-order) PLCP. We know

- $PLCP[i+1] \geq PLCP[i] - 1$
- $PCLP[i] \leq n - 1 - i$ (for $0 \leq i < n$)

$i =$	0	1	2	3	4	5	6	7	8	9	10	11
$PCLP[i] =$	0	4	3	2	1	1	0	1	1	0	0	0
$PCLP[i] + i =$	0	5	5	5	5	6	6	8	9	9	10	11

Space-Efficient LCP Representation

[6] proposed to use a CSA and the permuted LCP array (LCP values in text-order) PLCP. We know

- $PLCP[i+1] \geq PLCP[i] - 1$
- $PCLP[i] \leq n - 1 - i$ (for $0 \leq i < n$)

$i =$	0	1	2	3	4	5	6	7	8	9	10	11
$PCLP[i] =$	0	4	3	2	1	1	0	1	1	0	0	0
$PCLP[i] + i =$	0	5	5	5	5	6	6	8	9	9	10	11

- Encode gaps of $PLCP[i] + i$ with unary code (results in bitvector H of length $2n$)
- In this example: $H = 1000001111011001011010 1$
- What additional structure is required to calculated $LCP[i]$?

Space-Efficient LCP Representation

With a $o(n)$-space select structure (arguments starting from 1) and a CSA we get:

```
access_lcp(i)
    x ← SA[i]
    return select(x + 1, 1, H) + 1 − 2x
```

Summary:

- Time complexity depends on CSA access
- Space: $2n + o(n)$ bits (for bitvector H + select) + |CSA|

Note: The LCP between arbitrary suffixes can be calulated in constant time using a RMQ structure.

The LCP-Interval Tree

Definition of an LCP-interval ([1])

An interval $[i, j]$, where $0 \leq i \leq n-1$ is called LCP-interval of LCP value ℓ (denoted by $\ell - [i, j]$) if

- $LCP[i] < \ell$ or $i = 0$
- $LCP[k] \geq \ell$ for all $k \in [i+1, j]$
- $LCP[k] = \ell$ for at least one $k \in [i+1, j]$
- $LCP[j+1] < \ell$

Every index k with $i < k \leq j$ and $LCP[k] = \ell$ is called ℓ-index. There are at most $\sigma - 1$ ℓ-indices in an LCP-interval.

Note: Each LCP-interval corresponds to a node in the suffix tree.

The LCP-Interval Tree – Example

i	SA	LCP	$\mathcal{T}[\mathrm{SA}[i], n-1]$
0	11	0	$
1	10	0	i$
2	7	1	ippi$
3	4	1	issippi$
4	1	4	ississippi$
5	0	0	mississippi$
6	9	0	pi$
7	8	1	ppi$
8	6	0	sippi$
9	3	2	sissippi$
10	5	1	ssippi$
11	2	3	ssissippi$

- Singleton intervals $\ell - [i, i]$ are omitted

Bibliography

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Disc. Algo.*, 2(1):53–86, 2004.

[2] Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In *Proc. CPM*, pages 181–192, 2009.

[3] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. CPM*, pages 181–192, 2001.

[4] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proc. SODA*, pages 319–327, 1990.

[5] Gonzalo Navarro and Luís Russo. Space-efficient data-analysis queries on grids. In *Proc. ISAAC*, pages 323–332, 2011.

[6] Kunihiko Sadakane. Succinct representations of LCP information and improvements in the compressed suffix arrays. In *Proc. SODA*, pages 225–232, 2002.