Deterministic Sparse Suffix Sorting on Rewritable Texts

Johannes Fischer
Tomohiro I
Dominik Köppl
suffix sorting problem

- sort all suffixes of string $T$ of length $n$
  $\Rightarrow$ suffix array (SA)
suffix sorting problem

- sort all suffixes of string $T$ of length $n$ => suffix array (SA)
- lengths of the longest common prefix (LCP) between adjacent suffixes

both in $O(n)$ time and space
sparse suffix sorting problem

• sort suffixes starting at positions in $P$ (+ sparse LCP)
• $P$: a set of $m$ (< $n$) text-positions
sparse suffix sorting problem

- sort suffixes starting at positions in \( P \) (+ sparse LCP)

- \( P \): a set of \( m \) (< \( n \)) text-positions
  - given text in RAM and \( m = o(n) \), we want
    - \( o(n) \) time
    - \( O(m) \) additional space
previous work

• $O(n)$-time algos for special $P$
  • [Kärkkäinen and Ukkonen, ‘96]
  • [Inenaga and Takeda, ‘06]
  • [Ferragina and Fischer, ‘07]
  • [Uemura and Arimura, ‘11]

• arbitrary $P$ (difficult!)
  • [Burkhardt and Kärkkäinen, ‘03]
  • [Bille+, ‘13] randomized
  • [I+, ‘14] improved on Bille+’s work

aim on deterministic algorithms
main result

1. text is loaded into RAM
main result

1. text is loaded into RAM
2. text space: re-writeable
main result

1. text is loaded into RAM
2. text space: re-writeable

\[ c := \# \text{ positions that must be compared to sort} \]

\[ O(m \ lg m \ lg n \ lg^* n + c \ lg^{0.5} n) \] time
\[ O(m) \] additional space
main result

1. text is loaded into RAM
2. text space: re-writeable
   \[ \text{c := \# positions that must be compared to sort} \]

\[ O(m \lg m \lg n \lg^* n + c \lg^{0.5} n) \text{ time} \]
\[ O(m) \text{ additional space} \]
high-level strategy

• use suffix binary search tree (BST) [Irving+, '03]
• insert every suffix into BST one by one
high-level strategy

• use suffix binary search tree (BST) [Irving+,’03]
• insert every suffix into BST one by one
high-level strategy

- use suffix binary search tree (BST) [Irving+, ‘03]
- insert every suffix into BST one by one
high-level strategy

• use suffix binary search tree (BST) [Irving+, ‘03]
• insert every suffix into BST one by one
high-level strategy

- use suffix binary search tree (BST) [Irving+, '03]
- insert every suffix into BST one by one
a naïve approach

• takes $O(L)$ time
• $L : \#\text{char-to-char comparisons}$
• $L = \Theta(nm)$ if arrows mostly overlap
a naïve approach

- takes $O(L)$ time
- $L$ : #char-to-char comparisons
- $L = \Theta(nm)$ if arrows mostly overlap
a naïve approach

• takes $O(L)$ time
• $L$ : #char-to-char comparisons
• $L = \Theta(nm)$ if arrows mostly overlap
a naïve approach

- takes $O(L)$ time
- $L$ : #char-to-char comparisons
- $L = \Theta(nm)$ if arrows mostly overlap
LCE query

- need \( c \) naive checks
- LCE query:
  longest common prefix of two substrings (suffixes)
LCE query

- need \( c \) naive checks
- LCE query: longest common prefix of two substrings (suffixes)
LCE query

- need \( c \) naive checks
- LCE query: longest common prefix of two substrings (suffixes)
LCE query

- need $c$ naive checks
- LCE query:
  longest common prefix of two substrings (suffixes)
LCE query

- need \( c \) naive checks
- LCE query: longest common prefix of two substrings (suffixes)
LCE query

• need \( c \) naive checks
• LCE query: longest common prefix of two substrings (suffixes)
our idea

T

insert 2

p1

p2
our idea

- create an LCE data structure on **scanned** area
our idea

• create an LCE data structure on scanned area
our idea

• create an LCE data structure on scanned area
our idea

• create an LCE data structure on scanned area
⇒ issue LCE queries
LCE with ESP

ESP = Edit Sensitive Parsing [Cormode+, '07]
LCE with ESP

- partition string into blocks of size 2 or 3
LCE with ESP

- Partition string into blocks of size 2 or 3
- Give each block an ID

ESP = Edit Sensitive Parsing [Cormode+, '07]
LCE with ESP

- partition string into blocks of size 2 or 3
- give each block an ID
- blocking should be locally consistent

ESP = Edit Sensitive Parsing [Cormode+, '07]
LCE with ESP

- partition string into blocks of size 2 or 3
- give each block an ID
- blocking should be locally consistent
ESP tree

- each block gets an ID (here: number)
- same content ⇔ same ID

\texttt{ababababababababababababababababababababab}
ESP tree

- each block gets an ID (here: number)
- same content $\iff$ same ID

a b a b b a b a b a b a b a b a b a b a b a b
ESP tree

- each block gets an ID (here: number)
- same content $\iff$ same ID
ESP tree

- each block gets an ID (here: number)
- same content ⇔ same ID

<table>
<thead>
<tr>
<th>ID</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a b a</td>
</tr>
<tr>
<td>2</td>
<td>b b a</td>
</tr>
<tr>
<td>3</td>
<td>b a b</td>
</tr>
<tr>
<td>4</td>
<td>a b</td>
</tr>
<tr>
<td>5</td>
<td>a a b</td>
</tr>
</tbody>
</table>

ababbababababbababaab

1 | 2 | 3 | 4 | 1 | 2 | 3 | 5
a b a b b a b a b a b a a b
ESP tree

- each block gets an ID (here: number)
- same content $\iff$ same ID
- recursion spans up ESP tree
# ESP tree

- each block gets an ID (here: number)
- same content ⇔ same ID
- recursion spans up ESP tree

<table>
<thead>
<tr>
<th>ID</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a b a</td>
</tr>
<tr>
<td>2</td>
<td>b b a</td>
</tr>
<tr>
<td>3</td>
<td>b a b</td>
</tr>
<tr>
<td>4</td>
<td>a b</td>
</tr>
<tr>
<td>5</td>
<td>a a b</td>
</tr>
</tbody>
</table>

```
ababbabababababbababaab
```

```
1  2  3  4  1  2  3  5
a b a b b a b a b a b a b b a b a b a b a b
```
ESP tree

- each block gets an ID (here: number)
- same content \(\Leftrightarrow\) same ID
- recursion spans up ESP tree

<table>
<thead>
<tr>
<th>ID</th>
<th>string</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a b a</td>
</tr>
<tr>
<td>2</td>
<td>b b a</td>
</tr>
<tr>
<td>3</td>
<td>b a b</td>
</tr>
<tr>
<td>4</td>
<td>a b</td>
</tr>
<tr>
<td>5</td>
<td>a a b</td>
</tr>
</tbody>
</table>
ESP tree

- each block gets an ID (here: number)
- same content $\leftrightarrow$ same ID
- recursion spans up ESP tree

<table>
<thead>
<tr>
<th>ID</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a b a</td>
</tr>
<tr>
<td>2</td>
<td>b b a</td>
</tr>
<tr>
<td>3</td>
<td>b a b</td>
</tr>
<tr>
<td>4</td>
<td>a b</td>
</tr>
<tr>
<td>5</td>
<td>a a b</td>
</tr>
<tr>
<td>6</td>
<td>1 2</td>
</tr>
<tr>
<td>7</td>
<td>3 4</td>
</tr>
<tr>
<td>8</td>
<td>3 5</td>
</tr>
</tbody>
</table>
### ESP tree

- each block gets an ID (here: number)
- same content ⇔ same ID
- recursion spans up ESP tree

<table>
<thead>
<tr>
<th>ID</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a b a</td>
</tr>
<tr>
<td>2</td>
<td>b b a</td>
</tr>
<tr>
<td>3</td>
<td>b a b</td>
</tr>
<tr>
<td>4</td>
<td>a b</td>
</tr>
<tr>
<td>5</td>
<td>a a b</td>
</tr>
<tr>
<td>6</td>
<td>1 2</td>
</tr>
<tr>
<td>7</td>
<td>3 4</td>
</tr>
<tr>
<td>8</td>
<td>3 5</td>
</tr>
<tr>
<td>9</td>
<td>6 7</td>
</tr>
<tr>
<td>10</td>
<td>6 8</td>
</tr>
<tr>
<td>11</td>
<td>9 10</td>
</tr>
</tbody>
</table>

height = $O(\lg n)$
construction time

• \(O(\lg n)\) time lookup/store IDs in dictionary
• \(O(\lg^* n)\) time for blocking [Cormode+, '07]

\[\Rightarrow O(|S| (\lg n + \lg^* n)) = O(|S| \lg n)\] time for ESP tree on substr \(S\)
local surrounding

- two equal substrings $S$
- sufficiently enclosed nodes have same ID

$3 \quad 1 \quad 2 \quad 1 \quad 4$

$5 \quad 1 \quad 2 \quad 1 \quad 6$

$O(\lg^* n)$ $O(\lg^* n)$ $O(\lg^* n)$ $O(\lg^* n)$
local surrounding

• two equal substrings $S$
• sufficiently enclosed ESP subtree have same root

$O(3^k \lg^* n)$
$S$

$O(3^k \lg^* n)$

height $k$
LCE queries
LCE queries

- compare $O(\lg^* n)$ chars
LCE queries

- compare $O(\lg^* n)$ chars
- move upwards
- compare $O(\lg^* n)$ IDs
LCE queries

- compare $O(\lg^* n)$ chars
- move upwards
- compare $O(\lg^* n)$ IDs
- (recursion)
LCE queries

- compare $O(\lg^* n)$ chars
- move upwards
- compare $O(\lg^* n)$ IDs
- (recursion)
- on mismatch:
LCE queries

- compare $O(\lg^* n)$ chars
- move upwards
- compare $O(\lg^* n)$ IDs
- (recursion)
- on mismatch:
  - move downwards
  - compare children
LCE queries

• compare $O(\lg^* n)$ chars
• move upwards
• compare $O(\lg^* n)$ IDs
• (recursion)
• on mismatch:
  • move downwards
  • compare children

$\Rightarrow O(\lg n \ lg^* n)$ time
build LCE DS

insert 2

T

insert 3

LCE query

p1

p2

p3
how to update?

build LCE DS

insert 3

LCE query

update
update

• blue arrow exceeds tree‘s range not much
  ⇒ do nothing
update

- blue arrow exceeds tree‘s range far
update

• blue arrow exceeds tree‘s range far

⇒ enlarge
update

- blue arrow traverses two trees
update

• blue arrow traverses two trees
  ⇒ merge trees
merge

• recalculate local surrounding of nodes near merge
merge

• recalculate local surrounding of nodes near merge
merge

- recalculate local surrounding of nodes near merge
- look at $O(\lg^* n)$ nodes on each height
- dictionary: $O(\lg n)$
  $\Rightarrow$ takes $O(\lg n \lg^* n)$ time
total time

• create suffix binary search tree (BST)
  • $O(m \lg m)$ time: put $m$ suffixes in BST
    • $O(\lg n \lg^* n)$ time: LCE query
  • $O(c \lg n)$ time: building ESP trees
  • $O(m \lg m \lg n \lg^* n)$ time for merging
• total:

\[ O(m \lg m \lg n \lg^* n + c \lg n) \]

• if $m, c = o(n) \Rightarrow o(n)$ overall time!
space

- $O(m)$ words for
  - BST sorting suffixes
  - locating ESP trees
- $O(c \lg n)$ bits for
  - ESP trees
  - grammar dictionary
space

• $O(m)$ words for
  • BST sorting suffixes
  • locating ESP trees
• $O(c \lg n)$ bits for
  • ESP trees
  • grammar dictionary

how to shrink ESP tree to $O(c \lg \sigma)$ bits?
overwriting text space
overwriting text space

- on finding two equal substrings
overwriting text space

- on finding two equal substrings
overwriting text space

- on finding two equal substrings
  - build ESP tree in redundant space
overwriting text space

• on finding two equal substrings
  • build ESP tree in redundant space
• truncate tree to fit into text space
overwriting text space

• on finding two equal substrings
  • build ESP tree in redundant space
• truncate tree to fit into text space
overwriting text space

• on finding two equal substrings
  • build ESP tree in redundant space

• truncate tree to fit into text space
time

• before: \( O(m \lg m \lg n \lg^* n + c \lg n) \) time

• now: \( O(c \lg^{0.5} n) \) time: construction

\[ \Rightarrow O(m \lg m \lg n \lg^* n + c \lg^{0.5} n) \] time overall
summary

• sparse suffix sorting problem
  • $n$: $|T|$
  • $m$: #positions
• solved with

\[ O(m \ lg \ m \ lg \ n \ lg^* n + c \ lg^{0.5} n) \] time
\[ O(m) \] additional space

• by
  • LCE queries on ESP trees
  • truncation of ESP trees
  • sophisticated merging
summary

- sparse suffix sorting problem
  - \( n: |T| \)
  - \( m: \) \#positions
- solved with
  - \( O(m \lg m \lg n \lg^* n + c \lg^{0.5} n) \) time
  - \( O(m) \) additional space
- by
  - LCE queries on ESP trees
  - truncation of ESP trees
  - sophisticated merging

that is all - any questions?