A Compressed Enhanced Suffix Array Supporting Fast String Matching

Enno Ohlebusch and Simon Gog

Institut of Theoretical Computer Science
Ulm University

August 26, 2009

Some Definitions and Notations

- Let $\mathcal{T}$ be a text of length n over the alphabet Σ.
- $|\Sigma|$ denotes the size of Σ.
- We denote the length of a pattern $\mathcal{P}$ with m.

Outline

- New (very simple) compressed suffix tree, called `cst_sct`, which supports the child operation in $\mathcal{O}(\log|\Sigma|)$ time (i.e. pattern matching in $m\log|\Sigma|$ time).
- `cst_sct` is based on the Super-Cartesian Tree (SCT) of the lcp-table.
- SCT takes only $2n+o(n)$ bits and replaces the child-table.
- We do not need range minimum queries.

Suffix Array

$\mathcal{T}$ = `acaaacatat`

|5 |10

i	SA	$\mathcal{T}_{SA[i]}$
1	3	`aaacatat`
2	4	`aacatat`
3	1	`acaaacatat`
4	5	`acatat`
5	9	`at`
6	7	`atat`
7	2	`caaacatat`
8	6	`catat`
9	10	`t`
10	8	`tat`
11		

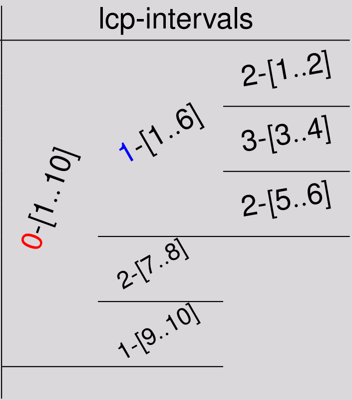
Properties

- *SA* gives the lexicographic order of the suffixes
- pattern matching takes $\mathcal{O}(m \log n)$ (binary search)

Enhanced Suffix Array

$\mathcal{T}$= `acaaacatat`

i	SA	LCP	$\mathcal{T}_{SA[i]}$
1	3	-1	`aaacatat`
2	4	2	`aacatat`
3	1	1	`acaaacatat`
4	5	3	`acatat`
5	9	1	`at`
6	7	2	`atat`
7	2	0	`caaacatat`
8	6	2	`catat`
9	10	0	`t`
10	8	1	`tat`
11		-1	



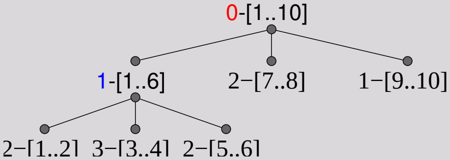
lcp-interval $\ell-[i..j]$. An ℓ-index is an index $k \in [i..j]$ with $LCP[k] = \ell$

Lcp-Interval Tree

$\mathcal{T}$ = `acaaacatat`
5 10

i	SA	LCP	$\mathcal{T}_{SA[i]}$
1	3	-1	aaacatat
2	4	2	aacatat
3	1	1	acaaacatat
4	5	3	acatat
5	9	1	at
6	7	2	atat
7	2	0	caaacatat
8	6	2	catat
9	10	0	t
10	8	1	tat
11		-1	

lcp-interval tree



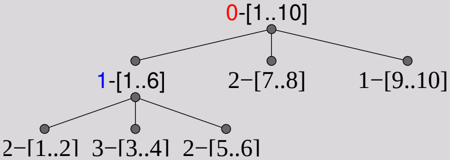
ℓ-indices split child intervals

Lcp-Interval Tree

$\mathcal{T} = \texttt{acaaacatat}$
5 10

i	SA	LCP	$\mathcal{T}_{SA[i]}$
1	3	-1	`aaacatat`
2	4	2	`aacatat`
3	1	1	`acaaacatat`
4	5	3	`acatat`
5	9	1	`at`
6	7	2	`atat`
7	2	0	`caaacatat`
8	6	2	`catat`
9	10	0	`t`
10	8	1	`tat`
11		-1	

lcp-interval tree



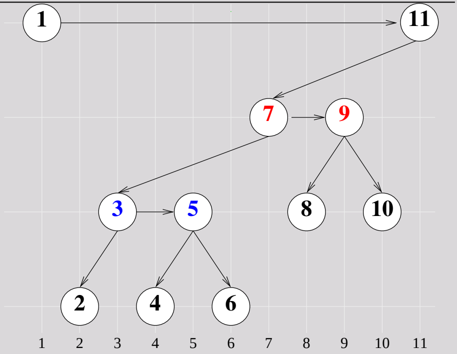
Problem: Given an lcp-interval, find child intervals.

Super-Cartesian Tree (1)

$\mathcal{T}$ = acaaacatat

i	SA	LCP	$\mathcal{T}_{SA[i]}$
1	3	-1	aaacatat
2	4	2	aacatat
3	1	1	acaaacatat
4	5	3	acatat
5	9	1	at
6	7	2	atat
7	2	0	caaacatat
8	6	2	catat
9	10	0	t
10	8	1	tat
11		-1	

Super-Cartesian tree of LCP

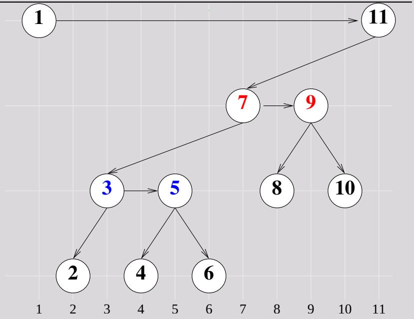


Super-Cartesian Tree (2)

$\mathcal{T}$= acaaacatat

i	LCP
1	-1
2	2
3	1
4	3
5	1
6	2
7	0
8	2
9	0
10	1
11	-1

Super-Cartesian tree of LCP



Properties

- Binary tree
- A node v corresponds to an index in the lcp array
- v has a right child or a right sibling
- $LCP[v] < LCP[v.L]$ and $LCP[v] \leq LCP[v.R]$

Calculate the child intervals of an lcp-interval

Given a lcp-interval $\ell - [i..j]$, we can compute the first ℓ-index k_1 as follows:

$$k_1 = \begin{cases} (j+1).L & \text{if } LCP[i] \leq LCP[j+1] \\ i.R & \text{if } LCP[i] > LCP[j+1] \end{cases}$$

The next ℓ-index

$$k_2 = \begin{cases} (k_1).R & \text{if } LCP[k_1] = LCP[k_1.R] \\ \perp & \text{otherwise} \end{cases}$$

Example for the lcp-interval 0-[1..10]

- $k_1 = (10+1).L = 7$ as $LCP[1] = -1 \leq -1 = LCP[10+1]$
- $k_2 = 7.R = 9, \quad k_3 = \perp$
- $\Rightarrow$ child intervals: [1..6], [7..8], [9..10]
- Apply method again to get ℓ values of the child intervals!

Remarks

So far:

- First child interval can be computed in constant time without range minimum queries
- but *i*th child takes $\mathcal{O}(|\Sigma|)$ time.

Next:

- Compress the Super-Cartesian tree
- and *i*th child takes $\mathcal{O}(\log|\Sigma|)$ time

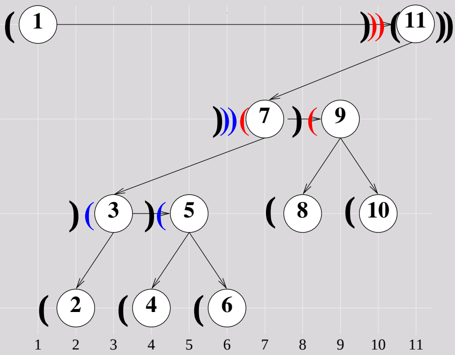
Succinct Super-Cartesian Tree

$\mathcal{T}$ = `acaaacatat`

i	*LCP*
1	-1
2	2
3	1
4	3
5	1
6	2
7	0
8	2
9	0
10	1
11	-1

Super-Cartesian tree of LCP

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



- Each node is represented by an opening and a closing parenthesis
- Size of the balanced parentheses sequence *bp* = $2n$ bits

Succinct Super-Cartesian Tree (2)

Construct the balanced parentheses sequence

```
$push(\langle 1, -1 \rangle)$
write an opening parenthesis
for $k \leftarrow 2$ to $n+1$ do
  while lcp[k] < top().lcp do
    $pop()$
    write a closing parenthesis
  if $k \neq n+1$ then
    $push(\langle k, lcp[k] \rangle)$
    write an opening parenthesis
  else
    write a closing parenthesis
```

Succinct Super-Cartesian Tree (3)

- We use the data structures of Jacobson, Munro and Clark to support $\mathcal{O}(1)$ time *rank* and *select*.
- We use the data structure of Geary et al. to support $\mathcal{O}(1)$ time *findclose*, *findopen*, and *enclose* on the balanced parentheses sequence.
- Construction of the balanced parentheses sequence *bp* is simple and runs in linear time and space (in bits).

Observation

The closing parentheses of the at most $|\Sigma|$ ℓ-indices for an lcp-interval are all neighbours in *bp*.
$\Rightarrow$ Binary search for the *i*th child.
$\Rightarrow$ $\mathcal{O}(\log|\Sigma|)$ time.

Succinct Super-Cartesian Tree (4)

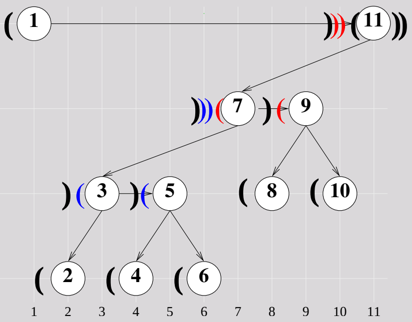
Calculate first ℓ-index of ℓ-[i..j]

$$k_1 = \begin{cases} rank_{(}(findopen(select_{(}(j+1)-1)) & \text{if } LCP[i] \leq LCP[j+1] \\ rank_{(}(findopen(findclose(select_{(}(i))-1)) & \text{if } LCP[i] > LCP[j+1] \end{cases}$$

i	*LCP*
1	-1
2	2
3	1
4	3
5	1
6	2
7	0
8	2
9	0
10	1
11	-1

Super-Cartesian tree of LCP

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



Example

- 0-[1..10]
- 1-[1..6]

Full Functionality

Other operations

- parent
- suffix link
- lowest common ancestor (LCA)

We have to compute

- *next smaller value* (NSV) and
- *previous smaller value* (PSV) queries

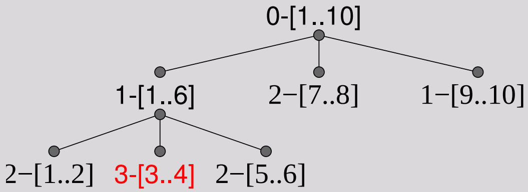
for the lcp-array for parent and suffix link.

Suffix link and LCA need a new operation on balanced parentheses sequences called *range restricted enclose* (rr-enclose).

The parent operation

parent(ℓ-[i..j])

```
if lcp[i] > lcp[j+1] then
    return lcp[i]-[PSV[i],j]
else if lcp[i] < lcp[j+1]
    return lcp[j+1]-[i,NSV[j+1]]
else
    return lcp[i]-[PSV[i],NSV[j+1]]
```



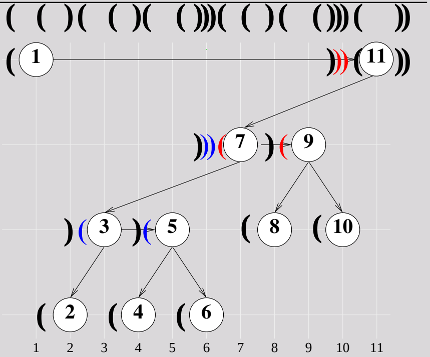
NSV and PSV

Calculate NSV[i]

$NSV[i] = rank_{(}(findclose(select_{(}(i))) + 1$

i	LCP
1	-1
2	2
3	1
4	3
5	1
6	2
7	0
8	2
9	0
10	1
11	-1

Super-Cartesian tree of LCP



Example

- NSV[2]
- NSV[3]

Time complexity

- NSV in $\mathcal{O}(1)$
- PSV in $\mathcal{O}(\log|\Sigma|)$

Implementation and experimental results

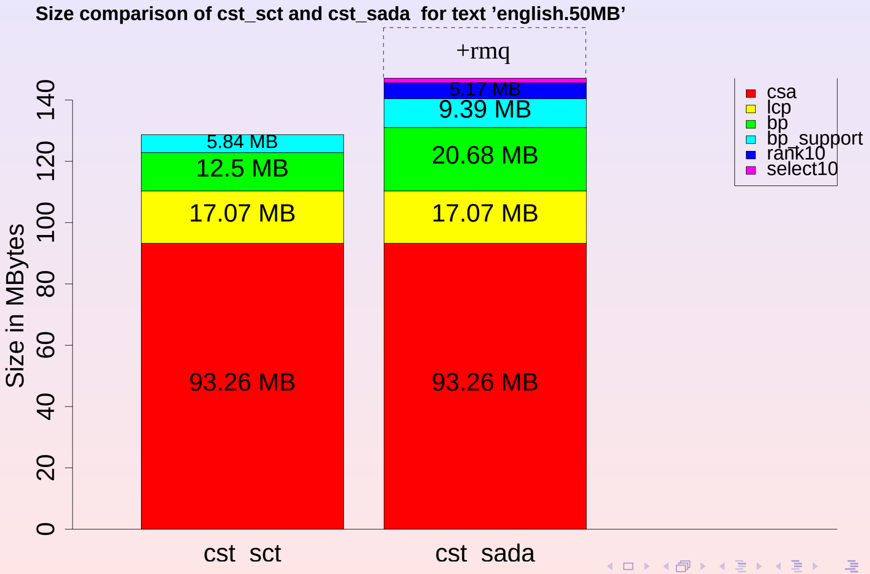
Implementation (www.uni-ulm.de/theo/in/research/sdsl)

- Template C++ library *sdsl* contains data structures for
  - bit vector, integer vector, rank support, select support
  - coders: Fibonacci coder, Elias-δ coder,...
  - balanced parentheses support
  - compressed suffix arrays
  - compressed suffix trees
  - ...

Results

- We used test cases from *Pizza&Chili* website.
- Child operation of `cst_sct` is 2-3 times faster on alphabets of size 90-230 than child operation on `cst_sada`
- `cst_sct` uses less space than `cst_sada`
- `cst_sada` is faster on parent operation

Comparison `cst_sada` vs. `cst_sct`(1)



Comparison `cst_sada` vs. `cst_sct` (2)

Operation	`cst_sada`	`cst_sct`
child(v,c)	$\mathcal{O}((t_{SA} + t_{SA^{-1}}) \cdot \lvert\Sigma\rvert)$	$\mathcal{O}((t_{SA} + t_{SA^{-1}} + t_{LCP}) \cdot \log \lvert\Sigma\rvert)$
parent(v)	$\mathcal{O}(1)$	$\mathcal{O}(t_{LCP} \cdot \log \lvert\Sigma\rvert)$
depth(v)	$\mathcal{O}(t_{LCP} \vee t_{SA})$	$\mathcal{O}(1)$
edge(v,d)	$\mathcal{O}(\log \lvert\Sigma\rvert \cdot (t_{SA} + t_{SA^{-1}}))$	$\mathcal{O}(\log \lvert\Sigma\rvert \cdot (t_{SA} + t_{SA^{-1}}))$
leaves_in_the_subtree(v)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
ith_child(v,i)	$\mathcal{O}(i)$	$\mathcal{O}(t_{LCP})$
ith_leaf(i)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
children(v)	$\mathcal{O}(\lvert\Sigma\rvert)$	$\mathcal{O}(\log \lvert\Sigma\rvert \cdot t_{LCP})$
sibling(v)	$\mathcal{O}(1)$	$\mathcal{O}(t_{LCP})$
sl(v)	$\mathcal{O}(1)$	$\mathcal{O}(\log \lvert\Sigma\rvert)$
lca(v)	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Summary

- `cst_sct` consists of only four components: *csa*, *lcp*, balanced parentheses sequence of sct (*bp*), support structure for balanced parentheses sequence.
- Construction of *bp* simple (linear time, only linear bits extra space).
- Size of `cst_sct`$= |csa| + |lcp| + 2n + o(n)$.
- Fast child operation for large alphabets.
- Provides full functionality.